杜毅仁　李慕靖
王建中　陈启帆　编

16位微型计算机

GHI LIU WEI WEI
XING JI SUAN JI

——上册

上海交通大学出版社

十六位微型计算机

（上　册）

杜毅仁 李慕靖 王建中 陈启帆 编

上海交通大学出版社

内 容 简 介

本书从微型计算机的硬件、软件、系统结构等方面出发，通过 Intel 公司的 8086 系列以及 IBM 公司的个人计算机作为典型，进行系统的介绍和剖析，使读者获得有关微机的基础和应用知识。

全书共分上、中、下三册。上册通俗易懂地介绍了 8086 的硬件结构、指令系统及其程序设计语言，是全书的基础和核心部分；中册就整个 8086 系列，分别介绍了输入/输出处理器 8089、专用数值处理器 8087、操作系统固件 80130、8086 的改进型 80186 和 80286，同时，还对 MULTIBUS 这一标准的系统总线以及 8086 的其它配套系列芯片作了介绍；下册比较系统地分析了IBM公司的个人计算机，内容包括 PC 的硬件结构、系统固件及操作系统等。

本书具有硬件、软件相结合的特点，而且既能了解 8086、又能由此了解十六位微型计算机的一般原理及应用。因此，既可作为高等院校的教学参考书，也可作为研究院所、厂矿有关研究人员、工程技术人员乃至一般用户的工作手册。

十六位微型计算机

（上册）

杜毅仁 李慕靖 王建中 陈启帆 编

上海交通大学出版社出版

上海淮海中路 1984 弄 19 号

新华书店上海发行所发行

立信会计专科学校常熟市梅李印刷联营厂印装

开本：787×1092 毫米 1/16 印张：25.5 字数：583,000

1984年7月第一版 1986年1月第2次印

印数10,000—20,000

统一书号：15324·33 科技书目：114—234

定 价：4.70

目　　录

第一篇　8086 的体系结构，系统设计和程序编制

中 册 目 录

第二篇　iAPX 86/88 微处理器系列

下 册 目 录

第三篇 IBM 个人计算机系统

第 一 篇

8086 的体系结构，系统设计和程序编制

本篇内容是对 8086 微处理机的一个全面介绍。它描述了 8086 的系统结构，以及怎样设计某一个 8086 的系统。论述尽可能详细，并且着重于举例和图解，希望它对于微型机的初学者和专业人员都有用。

本篇由三个主要部分组成——8086 的体系结构，8086 的系统设计和 8086 的程序编制。体系结构被细分成第二章 8086 的机器组成（寄存器、存贮器结构和寻址方式）和第三章 8086 的指令系统。8086 的系统设计在第四章中讨论，在这一章中介绍了如何把 8086 微处理器和其他部件组合起来以构成一个完整的微处理机系统。程序设计部分分成 8086 汇编语言程序设计（第五章和第六章）和 8086 高级语言设计（第七章）。

通过第一章的介绍，希望把水平不同的读者对于计算机和微型计算机的认识提高到一个共同的水平。如果读者已经具有那种知识并且迫切地想了解 8086 的话，则可以跳过第一章而直接进入第二章。

第一章 引 言

这一章叙述了微型计算机，特别是 8086 的技术特点和历史沿革。微型计算机除了在微型和价廉之外同任何计算机没有什么本质的不同，因此，本章将先从计算机的基本原理出发，然后叙述微型计算机的发展进程，最后阐述 8086 的特点。

§1.1 计算机概况

在讨论微型计算机之前，先让我们简短地介绍一下计算机。除了作为一个回顾以外，这一节还要引进一些在本书中使用的术语和概念。

组成一个计算机系统的基本部件表示在图 1.1 中。图 1.2 用非人物化的方法表示了同一个系统。下面我们集中在每一个方框的功能上来说明这样一个系统的特性。

计算机从一个输入设备取得数据，在计算和处理数据后，把最终结果送到输出设备。所要做的计算和处理可由一张被称为程序的指令表来说明。这个程序通常放在存贮器的程序区中。

计算机的操作由一个称为控制部件的设备控制。它检查、判断和控制机内各部件重复做下列三步：

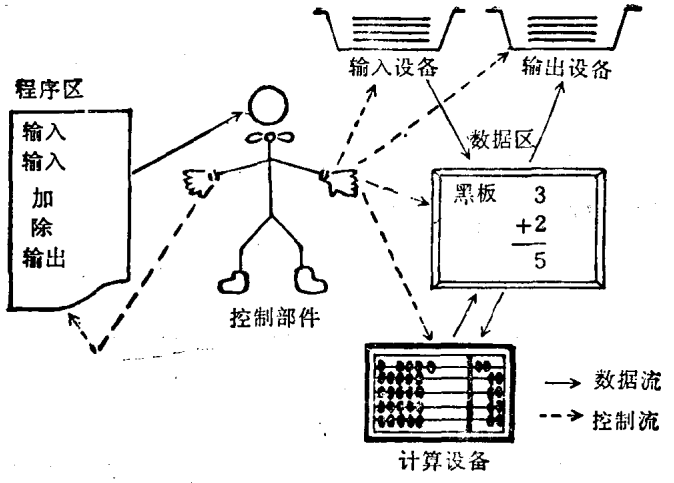


图 1.1　原始的计算系统

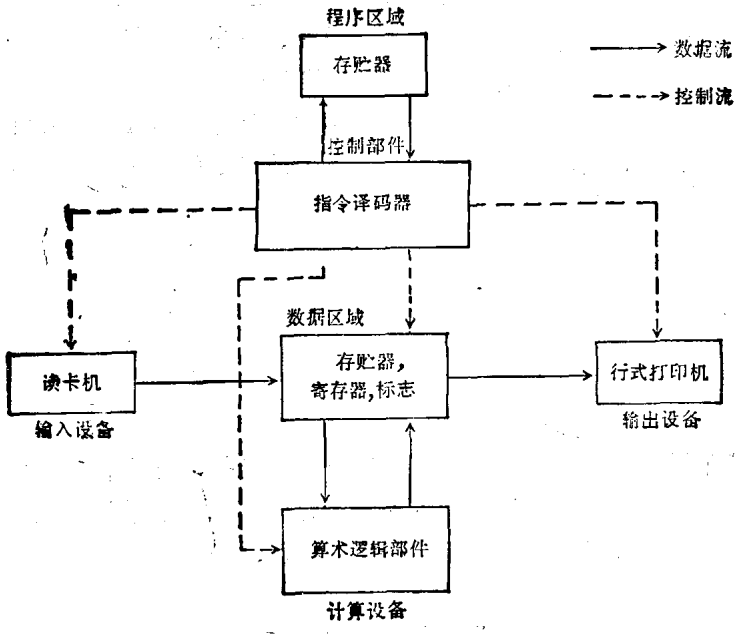


图 1.2　近代通用计算系统

1．从程序区取指令。

2．将指令译码以决定要执行何种操作。

3．把控制信号发送到执行操作的设备去执行指令。

在指令执行期间实现的操作，大体上是由在设备之间传送数据和在设备内部对数据实行计算等部分组成。计算是由运算设备实现的。数据区域是指用来为计算提供数据和存放中间结果的那一部分存贮器区域，以及通用寄存器和特征位寄存器。

让我们通过分析一条“加法”指令的执行过程，来了解这个系统是怎样相互连接在一起的。控制部件发出一个控制信号，并把它送到程序区，请求下一条指令。程序区把一条指令发送到控制部件作为响应。控制部件对该指令译码后，发现是一条“加法”指令，于是控制部件就把

控制信号发送到：

(1) 数据区，要求它传送两个数值到运算设备。

(2) 要求运算设备把取得的两个值相加。

(3) 通知指令中指定的数据区单元接收相加的结果。

程序区和数据区虽然通常都是存放信息的存贮器的组成部分，但在每个区中所放的信息种类则有很大的不同。数据区放中间结果，它在程序执行期间是经常改变的。程序区放程序，它在执行某个既定程序时通常是不改变的（近年来，在执行过程中修改自身程序的工作方式，已经不怎么流行了）。在一些系统中，程序事实上是“固化”在存贮器中的，所以它们只能被读出，而不能被改变。具有这种性质的存贮器被称为只读存贮器（简写成 ROM）。很明显，ROM 不适合用于数据区。数据区由读-写存贮器组成，一般简称为 RAM（RAM 是随机访问存贮器的缩写）。

存贮器是一个连续存贮单元的集合，每个单元具有一个唯一的地址。每个单元含有一串连续的 2 进制数的位，这些位或是 0 或是 1，从而构成一个 2 进制数。关于 2 进制数在这一章的后面还要加以介绍。

前面已经说过数据区由寄存器和标志以及存贮器组成。象存贮器一样，寄存器也是用来存贮中间结果的。通常在寄存器中存取数据比在存贮器中更容易，速度也快得多。计算机使用标志作为指示器来决定下一步要做什么。标志可以分成二类，即记录前面执行指令所产生的结果的特征的状态标志和控制计算机操作的控制标志。例如，溢出标志就是一个状态标志，它指出操作的结果是否超过了计算机所能保持的精度。中断允许标志则是一个控制标志，它表示处理器是否能接受外部的中断。

在计算机系统中的另一个设备是输入输出转接口（也称为 I/O 转接口）。一个输入输出转接口就象一扇通往外部世界的门，通过它才能与输入设备或输出设备传送信息。为了使原理图简化，I/O 转接口没有在图 1.1 和 1.2 中画出。

§1.2 数 据 格 式

一个存贮器单元的内容可以是程序中的一条指令，也可以是一个数据。指令在存贮单元里存放的方法，称为指令格式，它对各种计算机来说都是不相同的。8086 的指令格式将在第三章中详加描述。这里只介绍在 8086 中使用的数据格式。

由计算机处理的数据可以是数值信息也可以是非数值信息。一个工资程序可能大量地使用数值数据，而一个文本编辑程序却要进行大量的非数值操作。用来表示非数值数据的格式被称为 ASCII 码。

一、数字系统

我们已经习惯于把数表达为 10 进制数字的序列，例如 365。这个数可以理解成为 3个100，6 个 10 和 5 个 1。数的这种表达法常常称为基数为 10 的表达法。人有 10 个手指，用 10 为基数的表达法来表示数，我们感到是天经地义的。但计算机是用电平高低来计数的。为了可靠，只用 2 个电平值，要么是高电平，要么是低电平（或电平等于 0）。于是，对于计算机来说，用以 2 为基数的表达法来表示它们的数，也是天经地义的。所以，在计算机中数是以 2 进

制数字 (bits) 的序列来表示的.例如，11010，这个数字可以理解为 1 个 16，1 个 8，0 个 4，1 个 2 和 0 个 1 所组成的数的以 2 为基数的表达法．由于 2 进制数字只有 0 和 1，它逢 2 就进 1，所以我们只要记住 1+1 等于 10 而不是 2，不需要先把它们转换成 10 进制数，2 进制数也能直接进行＋、－、×、÷运算.

例如：

```
   1001      9 的 2 进制表示法
 + 0101      5 的 2 进制表示法
 ------
   1110     14 的 2 进制表示法
```

尽管计算机擅长于处理 2 进制表示的数，但这种很长的 2 进制数字序列人们却常常容易搞混淆．例如，181 的 2 进制表示是 10110101．为了压缩数字的长度，使它更便于记忆，人们根据 2 进制数的特点引出了数的 16 进制表示法，不言而喻，这种数的表示法的基数是 16．由于 2^4 等于 16，16 进制数必然和 2 进制数有着十分密切的联系．事实正是这样，16 进制数的每一个数字，恰好能用一个 2 进制的 4 位组来表示，它们的对应关系在表 1.1 中表示．这样 10110101 可以缩写成 B5，这样的数称为 16 进制数．如果人们生来就有 16 个手指的话，无疑这就是我们所要使用的数字系统．现实世界中也有应用 16 进制的，中国的老称，不是 1 斤等于 16 两吗！

表 1.1　16 进制数字表达法

2 进制数	16 进制数	10 进制数	2 进制数	16 进制数	10 进制数
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	A	10
0011	3	3	1011	B	11
0100	4	4	1100	C	12
0101	5	5	1101	D	13
0110	6	6	1110	E	14
0111	7	7	1111	F	15

二、带符号数

2 进制数的概念对于描述正数和 0 是很完美的，但当我们要把它用于负数时，就需要有一个附加的机构来指明该数的符号．这样做最简单的办法是把该数的最高位（最左位）作为符号位，例如：

```
0000 0100      是 +4
1000 0100      是 −4
0111 1111      是 +127
1111 1111      是 −127
```

这样的表达法称为带符号的表达法，但它有一个严重的缺点：即它需要一个新的运算规则．这一点在要从 0 中减去 +1 而希望得到 −1 时是很明显的，例如：

```
   0000 0000      带符号表示的 0
 − 0000 0001      带符号表示的 +1
 ------------
   1111 1111      带符号表示的 −127
```

若我们想要把在无符号数上用的2进制运算规则运用到带符号数上去，我们就需要另一种符号数表示法，用这种方法 1111 1111 代表 −1 而不是 −127。同样，从 −1 减去 +1 应该得到 −2。让我们实行这个减法来看 −2 应该是怎么样的。

```
  1111 1111        −1
− 0000 0001        减去 +1
-----------
  1111 1110        把这个称为 −2
```

这样，正数和负数似乎应该表示成这样：

```
     ⋮
0000 0011        +3
0000 0010        +2
0000 0001        +1
0000 0000         0
1111 1111        −1
1111 1110        −2
1111 1101        −3
     ⋮
```

事实上也正是这样，这种表达法称为2的补码表达法，它具有可以运用2进制加法和减法规则，并给出正确的2进制补码结果的性质，例如：

```
  0000 0011      2的补码表示的 +3
+ 1111 1110      2的补码表示的 −2
-----------
  0000 0001      2的补码表示的 +1
```

它也具有每个非负数（正数或0）的最高位是0和每个负数的最高位是1的性质。这样，就和带符号的表达法一样，最高一位起符号作用。

一个2的补码数的相反数可以通过改变每一位的值和末位加 +1 来获得。例如，可以从 +3 的2的补码表达法中取得 −3 的2的补码表达法如下：

```
  0000 0011      2的补码表示的 +3
  1111 1100      每一位都改变了的 +3
+ 0000 0001      末位加 +1
-----------
  1111 1101      2的补码表示的 −3
```

关于2的补码数有一点要引起警惕，即当要把一个8位的2的补码数扩展成16位（例如，使它能和一个16位的2的补码数相加）时，怎样正确地处置前面附加的8位。请先看下面的例子。

假如要给 0000 0000 0000 0011（2的补码表示的 +3），加上 0000 0001（2的补码表示的 +1），在这种情况下毫无疑问，我们会在 +1 的左边简单地加上8个0，然后再把两个数相加。

```
  0000 0000 0000 0011      (用2的补码表示的 +3)
+ 0000 0000 0000 0001      (用2的补码表示的 +1)
---------------------
  0000 0000 0000 0100      (用2的补码表示的 +4)
```

但是，如果我们要把 1111 1111（2的补码表示的 −1），加到 0000 0000 0000 0011（2的补码表示的 +3），却必须在 −1 的左边加上8个1（加0会使它变成正数）。这个加法就是：

```
  0000 0000 0000 0011      (用2的补码表示的 +3)
+ 1111 1111 1111 1111      (用2的补码表示的 -1)
-------------------
  0000 0000 0000 0010      (用2的补码表示的 +2)
```

这样，把一个8位数扩展成16位时要这样进行：

数值	8位表示	16位表示
+1	0000 0001	0000 0000 0000 0001
-1	1111 1111	1111 1111 1111 1111

扩展一个2的补码数的规则，是把原来符号位的值扩展到附加的8位上去。这种操作称为符号扩展，只有这样，才能正确执行补码运算。8086中就应用这种方法。

三、字　符

字符可以用一个位的序列来表示。我们要表示的最低限度的字符集是26个字母和10个数字，共36个字符。若要能区别大写和小写（另外26个字符）字母，并且能表示一些特殊的字符（例如+和*），则超过了64个字符，这样至少需要7位才能表示全部所需要的字符（一个6位的数所能表达的最大值是64）。一种通用的7位编码称为ASCII码（American Standard Code for Information Interchange），它被表示在附录C中。一个8位的存贮器单元称为一个字节，因为它能很方便地用来存放一个ASCII编码的字符（第8有时用来作为正确性检查位）。

§1.3 堆　栈

堆栈是在微处理机中经常出现的概念。堆栈的另一个名称是“下推表”或“后进先出队列”。这些名称可以从堆积自助食堂碟子的设备中想象出来。当一个洗干净的碟子被放在碟子堆的顶上时，它把下面的碟子往下压一层，当这个顶上的碟子被人取去时，所有的碟子就往上弹一层。很明显，最后放在堆栈顶上的碟子将是最先被人取走的那一个碟子，而最先放在堆栈中的碟子将是最后一个被取走的碟子，这就是后进先出名称的来源。

值得注意的是，堆栈在计算机中是很有用的机构，为了说明这一点，我们有必要先看看子程序。子程序（有时也称为过程）是一个程序的一部分，它是被主程序调用来执行特殊任务的，它提供了把要解的全部问题分为较小和较简单的很多模块的方便。一个子程序本身还可以调用其它子程序，去进行更细分的工作。当一个子程序完成任务以后，它就返回到调用它的主程序，这样就形成了一个子程序调用序列，后一个子程序压在前一个子程序的上面，直到最后一个被调用的子程序为止。这最后一个被调用的子程序正是第一个要返回的子程序。换一句话说，子程序的重迭调用具有后进先出的特点。因此，堆栈就在此时发挥了作用。实际上，当一个子程序被调用时，某些信息必须保护起来。这些信息可能包括一些寄存器中的内容和当前设置的标志，当然也包括子程序最终要返回的地址。当子程序完成任务以后，它要恢复这些被保护的信息，把它们送回到原来的寄存器中去，其中包括把标志位恢复成原来的值，并且使用“返回地址”使之回到主程序去。由于最后一个被调用的子程序是第一个要返回的子程序，所以最后被保护的信息将是第一个要被恢复的。这样信息就必须要被堆放得如同自助食堂的碟子一样。至此，我们已经描述了堆栈是怎样动作的和为什么在计算机中是一种有用的机构。现在我们来看计算机的堆栈是如何实现的。由于堆栈必须保存信息，所以它肯定是某种存贮

器，事实上，存贮器的任何可用部分（只读存贮器除外）均可被用来作为一个堆栈。所需要的只是一个指向该堆栈最末一个单元的指示器。这个指示器常常称为栈顶指示器，而被该指示器指着的存贮单元通常被称为堆栈顶。当一块新的信息被推入堆栈中时（或称为压入堆栈），堆栈指示器被递减以使它指向下一个存贮器单元，而被推入的信息就存放在该存贮器单元内。当一个信息从堆栈中推出时（或称为弹出），信息从堆栈指示器指着的单元中弹出来，而堆栈指示器将被递增以指示当前堆栈顶的位置。

§1.4 8086 存贮器的分段

前面的章节举例说明了存贮器可以用来存放程序（代码）、存放数据（数和字符）和作为堆栈。这样，8086 事实上把它的存贮器分成代码段、数据段和堆栈段就不足为奇了。关于存贮器将在第二章中讨论。

§1.5 微型计算机的发展简史

在概述了计算机的基本概念之后，了解一下计算机的发展历史，并且看看它的演化到微型计算机的过程是有好处的。

一、从大型计算机到微型计算机

在 50 年代，所有的电子设备（收音机、电视机以及计算机）都是笨重的电子管设备。那个时代的计算机被称为第一代计算机。例如：IBM 的 650 和 704。这些计算机包含有若干个电子设备的机架，安装在很大的房间里。50 年代末期，晶体管和其它固态电路开始登上历史舞台并开始取代电子管。利用这种技术的计算机被称为第二代计算机。例如IBM 7090和Burroughs B5500。

在 60 年代，许多分立的电子器件（电阻、电容、晶体管等）被组合在一个单一的集成电路内（简称 IC）。IC 的面积比一张邮票还小，一般封装在象蜈蚣一样的管壳里。它能方便地插入一个系统。这种可插可拔的集成电路常称为中小规模集成电路。用 IC 制造的计算机是第三代计算机。例如 IBM360，GE635 和 Burroughs B6700。由于集成电路技术的飞速发展，到了 70 年代初期，在图 1.2 中的许多部件可以集成到一块很小的硅晶片上，例如，Intel 4004 和 8008，从而开始了晶片上的计算机时代。

到了这个时候，不仅是计算机的体积已经急剧地减小，而且价格也大大下降。电子管计算机的价格是以百万美元来计算的，而晶片上的计算机的最初价格是 300 美元左右，在几年之内竞争使得价格下降到 10 美元以下。

晶片上的计算机被称为微处理机或微处理器。虽然这些术语有时可以互换使用，但是却有不同的含义。一个微处理器是指单一的片子，它通常由控制部件、算术逻辑部件、寄存器、标志、输入/输出转接口等部件组成，也就是通常称为 CPU（中央处理部件）的部分，故微处理器就是微型计算机中的 CPU。程序和数据存贮器以及输入/输出设备通常不在微处理器片子上。微处理机相当于一台计算机的主机部分。微处理机通常由一片微处理器，许多存贮器片子和输入/输出接口片及外围电路组成。当然，也有整个微处理机包含在一个片子上的，例

如，Intel 的 8048，这被称为整单片微型计算机，也可称为整单片微处理机。由于微型计算机发展极快，其结构也日新月异，就是主要部件的名称也很不统一，有时令人捉摸不定。就目前情况来看，按微处理器、微处理机和微型计算机的层次命名还比较合理。它们的关系如图 1.3 所示。

微处理器：是指把运算器和控制器看成一个整体，将其做在一片大规模集成电路 (LSI) 上的器件。对该器件的称呼很混乱，有的称为中央处理器 (CPU)，有的称为微处理器 (MPU)，有的称为微处理机 (μP)，也有称为通用微处理机的。看来称之为微处理器比较合理，称之为中央处理器 (CPU) 也可以，因为 CPU 在计算机范围内用得比较久了，不会引起人们的误解。

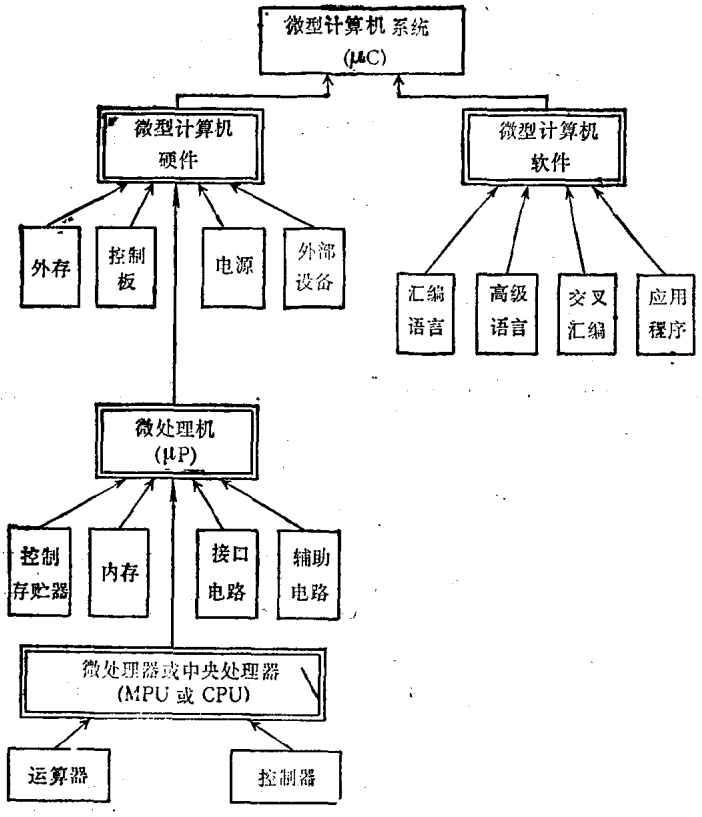


图 1.3 微型计算机的组成

微处理机：由图 1.3 可知，微处理器和其他部件，即存贮器片、输入/输出接口片以及辅助电路用某种方法结合在一起以后，就构成微处理机，它相当于一台计算机的主机部分，而且在某些专用场合，已经可以作为一台专用微型机使用了。把上述各种功能的片子组合在一起的方法主要有两种：一种是常规的装配方法；另一种是大规模集成法，即把上述功能片都集成在一片 LSI 或 VLSI (超大规模集成电路) 上，从而构成整单片微处理机，也有人称它为整单片计算机的。事实上，它们只能是一些专用机。

微型计算机：狭义地说，只要将上述微处理机加上电源部件、控制板、配上外部设备就构成一台微型计算机了，但这只是微型计算机的硬件部分，完整的微型计算机还应包括软件。有时把上述硬件系统和软件系统的总和称为微型计算机系统。这才是人们通常所指的微型计算机。

二、从8008 到 8086

微处理器时代是从 1971 年 Intel 推出 4004 和 8008 微处理器开始的。这是第一代微处理器。这两种片子都是为专门的应用而设计的，4004 主要用于计算器，8008 则主要用于计算机终端设备。这些微处理器虽然比较新奇，但并没有得到足够的重视。直到 1974 年，当 8008 成长为 8080 (第二代微处理器) 时，才引起了计算机工业的震动。8080 是第一个精心设计，可以广泛地在各方面应用的微处理器。它很快就风靡世界，变成了“标准”的微处理器工业产品。

微处理机现在已经能够实现老式的计算机能实现的各种任务，并且价格已经便宜到能够

到达业余爱好者手中的程度。许多 Intel 以外的公司也开始制造 8080 片子，而且有一些公司(著名的 Zilog 公司) 造出了 8080 的增强型产品(即 Z-80)。Intel 公司本身也在 1976 年推出了增强型产品 8085。但是直到 1978 年 Intel 公司生产出 8086 为止，微处理器的基本特点并没有重大的改变。8086 与 8080/8085 是向上兼容的(以前为 8080/8085 开发的软件能够方便地转化为 8086 的软件)，而 8086 的先进性足以称为第三代微处理器。

三、8086 成功的奥秘

8086 究竟提供了什么，使它能立即获得成功？要了解答案，我们必须先看看 8080/8085 的不足之处。

8080/8085 早期的成功，鼓励着用户把它使用在越来越大的系统中。这些系统变得如此之大，以致 8080/8085 的 64KB (其中 K＝1024，B 表示字节) 的寻址范围远远不能满足他们的需要了。8086 的寻址范围超过一百万存贮器字节单元。除此之外，8080/8085 也被越来越多地用在需要比 8 位长的快速处理领域中。8080/8085 的 8 位字长使得长的数据必须要由几个 8 位组来构成，而每一组又必须分开操作，这样就增加了处理时间。8086 虽然是基于 16 位字长操作的，但它也保留了处理 8 位数据的能力，从而使得短的数据仍能被 8086 有效地处理。考虑到 8080/8085 用于通用计算机系统时缺少乘法和除法指令，还缺少带符号数的操作，因而在使用上很不方便，8086 就提供了以前缺少的这些运算指令。由于越来越多的 8080/8085 程序采用高级语言编写，它们要先被翻译成 8080/8085 的机器语言，然后才能被执行。鉴于 8080/8085 的寻址方式不能支持把用高级语言写的程序转换成有效的 8080/8085 代码，8086 的寻址方式则设计得能适应高级语言的处理。从大量的应用中发现 8080/8085 在变换数据串时能力很差，8086 则设计得能有效地处理数据串。最后，随着系统变得日益复杂，单靠一个处理器来执行系统的全部功能的方法，越来越行不通。但是，8080/8085 没有能力与其他处理器合作，而 8086 则设计成能在多处理器环境下运行。特别是 8086 与两个出色的协处理器，即数字数据处理器 8087 和 I/O 处理器 8089 天衣无缝的联合，加上 80186 和性能更强的 8086 的增强型 80286 (详见本书第二篇) 的出现，更把 8086 的性能推到了前所未有的高度。所有这些就是 8086 成功的奥秘所在。

习　　题

1. 简述输入/输出部件，控制部件和算术逻辑部件在计算机中的作用。
2. ROM、RAM 有什么区别，哪一种适合于作为数据区？
3. 计算机一般有几类标志，它们的作用各是什么？
4. 控制部件的主要作用是什么？试以一条"减法"指令来分析控制的过程。
5. 计算机的所谓数据区由哪些部分组成？
6. 试把 10 进制数 37, 58, 103, 196, 512, 1024 化为 2 进制数和 16 进制数，计算机中还有一种经常使用的 8 进制数(数字是 0, 1, 2, 3, 4, 5, 6, 7)，你能把这些数化成 8 进制数吗？
7. 有四个 8 位的带符号数 0100 1001, 1000 0001, 0001 0101, 1100 1000 试把它们符号扩展为 16 位数，若要扩展为 32 位数，怎么扩展？
8. 堆栈的工作原理是什么？试用一迭盘子体会一下放入和取出的操作。
9. 什么是子程序，它的主要作用是什么？
10. 什么是微处理器，微处理机，微型计算机，微型计算机系统，它们的区别和联系怎样？
11. 8086 和 8080 相比有哪些主要优点？

第二章　8086 的组成

§2.1　概　　述

描述一台计算机的一种方法是描述组成该计算机的功能部件。这些部件和它们之间的相互作用称为该计算机的结构。例如，在计算机中有多少寄存器，这些寄存器起什么作用，可以连接多少存贮器，存贮器是怎么被寻址的，以及可以使用哪种输入/输出机构等等。

8086 是一块含有构成一台计算机的绝大多数部件的大规模集成电路片子。其中包括控制计算机所有功能的控制电路，所有寄存器和标志位以及算术逻辑操作部件。存贮器和输入/输出转接口虽然没有包含在片上，但可以很方便地与其他外围支持片子相连接从而形成一台计算机。这些片子的集合有时称为微处理机，其中央处理器 (CPU) 则称为微处理器。

如果想简要地描述 8086 CPU 的结构，它应当是这样的：8086 有 4 个寄存器集。其中第1个集含有 4 个用来保存中间结果的通用寄存器；第 2 个集含有 4 个用来在存贮器中寻找信息时，作为指示器和变址器的寄存器；第 3 个集含有 4 个用来支持存贮器分段的段寄存器；第 4 个集含有指令指示器。8086 中还有 9 个标志位。这些标志用来记录处理器的状态和控制它的操作。8086 可以访问多至一百万个存贮器字节和多至 65,000 个输入和输出转接口。这一章的前半部分将详细地推敲这些特点，至于8086片子的引脚功能和内部结构则放在本章末尾来讨论。

一般的计算机指令都涉及到寻找指定的操作数 (要处理的数据)，在这些操作数上执行一个操作，并把结果放回到指定的单元。操作数和结果单元由指令指定，可以在存贮器中也可以在寄存器中。用来指定这些单元的机构称为计算机的操作数寻址方式。8086 的操作数寻址方式将在这一章的下半部分详加描述。在指定的操作数上进行操作的真实指令在第三章中描述。

§2.2　存贮器结构

8086 系统中的存贮器是一个可以多至 2^{20} (大约 1,000,000) 个 8 位数量的字节序列。每一个字节单元分配一个唯一的地址 (无符号数，2 进制从 0000 0000 0000 0000 0000 到 1111 1111 1111 1111 1111；16 进制从 00000 到 FFFFF)，这一点在图 2.1 中说明

16 进制数地址	2 进 制 数 地 址
00000	0000 0000 0000 0000 0000
00001	0000 0000 0000 0000 0001
00002	0000 0000 0000 0000 0010
00003	0000 0000 0000 0000 0011
⋮	⋮
FFFFE	1111 1111 1111 1111 1110
FFFFF	1111 1111 1111 1111 1111

存 贮 器

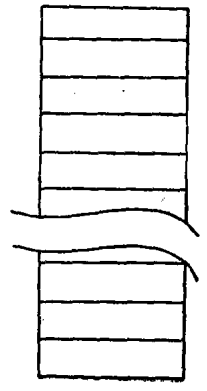


图 2.1　存贮器地址

在存贮器中任何两个相邻的字节被定义为一个字．在一个字中的每一个字节有一个字节地址，并且这 2 个地址中的较小的一个被用来作为该字的地址。字的例子在图 2.2 中表示。

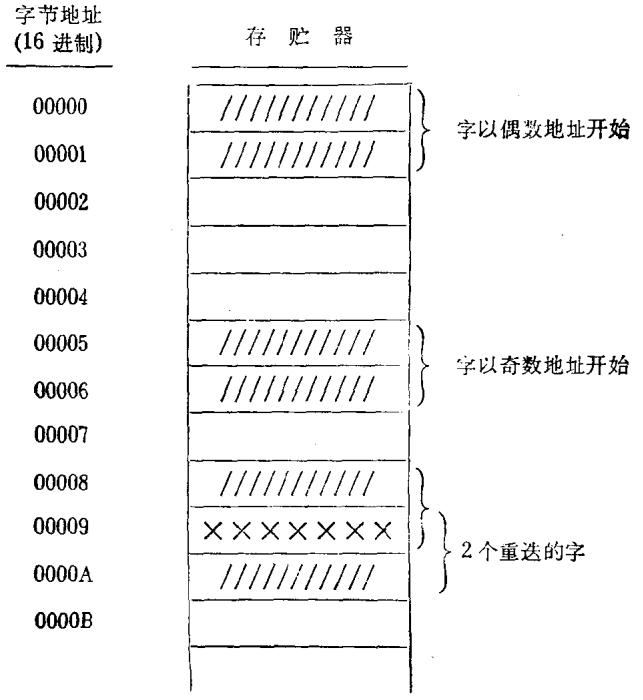


图 2.2 字在存贮器中的例子

一个字含有 16 位。具有较高存贮器地址的字节含有该字的高 8 位。具有较低存贮器地址的字节含有该字的低 8 位。初看起来，这一点似乎很自然。高字节当然应该具有较高的存贮器地址。但是，当考虑到存贮器是一个从最低地址开始到最高地址为止的字节序列时，很明显，8086 是反向地存放它的字的(或许它们应当称作反字)。这种情况可以用图 2.3 (a) 中的例子来说明。

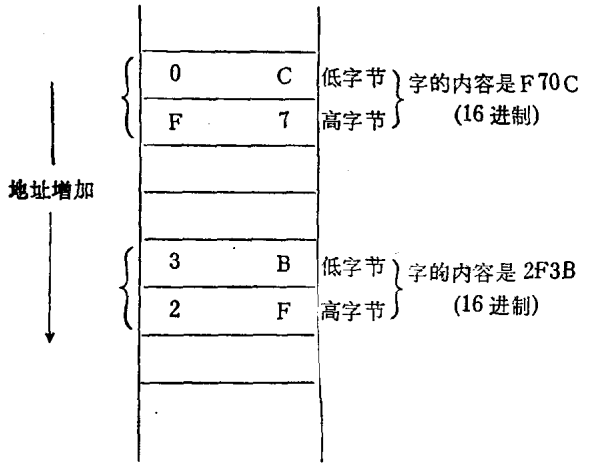


图 2.3(a) 在存贮器中"反字"存放的例子

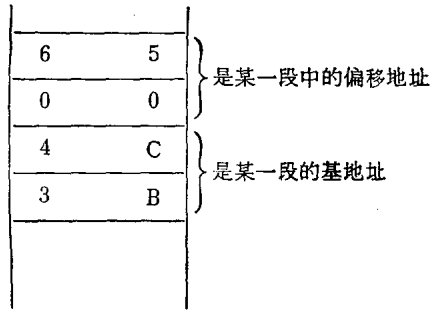


图 2.3(b) 一个指示器的例子

在 8086 中还有一类特殊的数据，它们是按双字 (即相邻的两个字) 进行存贮的，通常被称为指示器。它主要用于指示当前可寻址段以外的数据和代码的地址。指示器的较低地址字是

一个在某段中的偏移地址，而较高地址字则是该段的基地址。图 2.3(b)是一个指示器的例子，其中 3B4CH (H 表示是 16 进制数)是某一段的基地址，而 0065H 则是在该段中的偏移地址。

8086 有些指令是访问(读或写)字节的，有些指令是访问字的。在同一时间里，8086 传送到存贮器或从存贮器中取出的信息数量总是 16 位，并且该 16 位总是在存贮器中以偶地址开头的两个字节的内容。在字节指令的情况下，这些位中只有 8 位是有用的，其余 8 位均被忽略。这就是说，一条开始于偶地址的读写一个字的指令，可以用一个存贮器访问周期来实现。然而，对开始于奇地址的字操作指令就变得复杂了，必须对两个连续的偶地址字做 2 次存贮器访问，忽略各自的不需要的一半，对剩余的一半做一些字节变换才能得到，各种字节和字的读出例子在图 2.4 中表示。实际上，8086 的程序并不涉及这些细节，一条指令只是请求访问一个特定的字节或字，实现这样一个访问所必须要做的一切，都是在处理器控制下自动实现的。

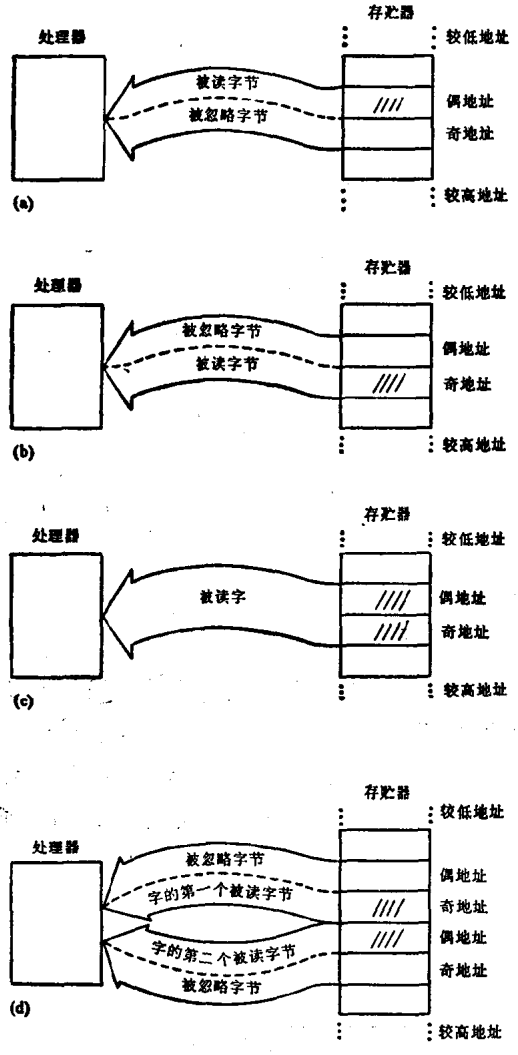


图 2.4 读偶地址和奇地址的字节和字：(a) 读偶地址字节；(b) 读奇地址字节；(c)读偶地址字；(d)读奇地址字需要 2 次存贮器访问.

§2.3 存贮器分段

由于 8086 可以寻址多至 2^{20} 个存贮器字节，所以，字节和字的地址必须表示成 20 位的数量。但是，8086 是设计来执行 16 位运算的，它能处理的地址目标只能是 16 位长的，这样，就需要一个附加的机构来建立地址。

我们可以把 1 兆字节的存贮器想象为任意数量的段，其中每一段最多可含有 2^{16} (大约 65000) 个字节。这样，每一段就必须开始于 1 个能被 16 整除的字节地址 (即该字节地址的最低 4 位全是 0)。在任意给定的时刻，程序可以立即访问 4 个段中的内容。这 4 个段被称为当前代码段、当前数据段、当前堆栈段和当前附加段(附加段是通用区域，经常作为一个附加的数据段)。8086 用把这些段的第 1 个字节的高 16 位地址放到 4 个专用寄存器中的方法，来访问这些段中的代码和数据。这些寄存器称为段寄存器。存贮器的分段并不是唯一的，它们可以相互重迭，对于 1 个具体的存贮单元来说，它可以属于 1 个逻辑段，也可以同时属于几个逻辑段。图 2.5 是 4 个段寄存器分别指示着存贮器中 4 个当前段的例子。在这个例子中假设每 1 个逻辑段的长度都是 64 KB。例如，假定 16 位代码段寄存器含有 16 进制数值 C018，这就使得该代码段在字节地址 0C0180H (在表示 16 进制数时，若第 1 个数

字是表示 10 到 15 的 A, B, C, D, E, F 时常在它们之前放上一个 0，以便把数字和字符区别开来，数字末尾的 H 表示是 1 个 16 进制数.)开始并可以一直扩充到总数 2^{16} (10000H) 个字节. 这样，在该代码段中可能的最后 1 个字节地址将是 0D017FH (0C0180H+0FFFFH=0D017FH).

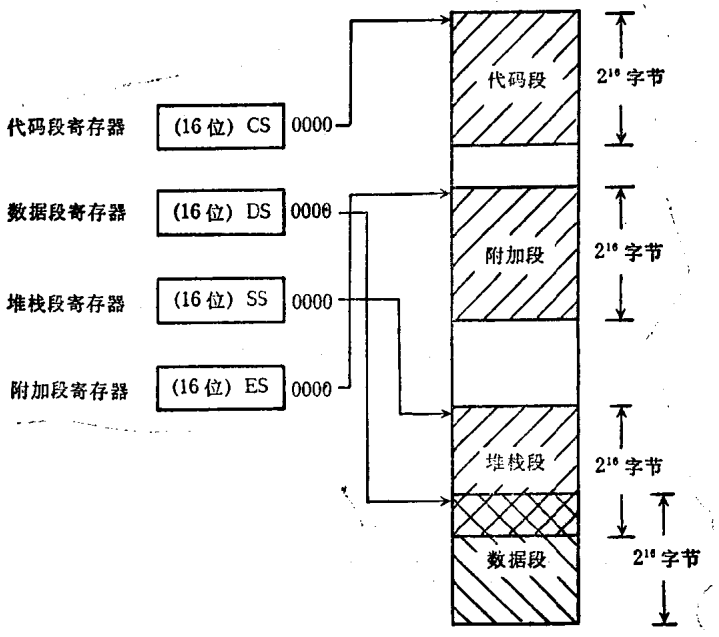


图 2.5 存贮器分段的例子，注意堆栈段和数据段在这个例子中是重迭的

某一段中的 16 位偏移地址用来寻找该段里的字节或字. 8086 处理器实际上是把 16 位偏移地址加上附加有最低 4 位 0 的 16 位段寄存器的内容来产生 20 位字节或字的地址，这个操作的原理在图 2.6 中表示. 这样，在前面的例子中，地址为 0CFFFFH 的字节在当前的代码段中，特别是它在该段中的偏移地址为 0FE7FH (0CFFFFH−0C0180H=0FE7FH) 这一点在图 2.7 中得到了说明.

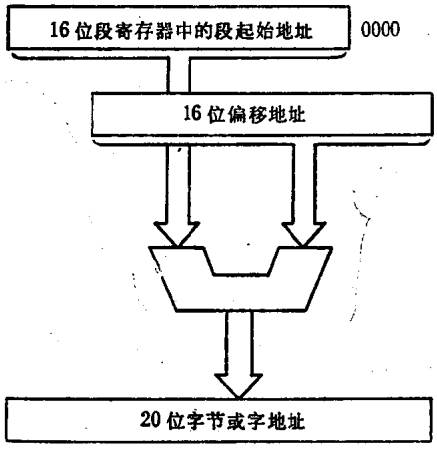


图 2.6 产生字节或字地址

由于段寄存器指向着 4 个当前可寻址段 (见图 2.5)，所以通过改变该寄存器的内容使其指向所要求的段，程序就可以对其它段中的代码和数据进行访问. 8086 的每一种应用都可以定义並使用不同的段. 当前可寻址段已提供了宽广的工作空间：64K 字节代码、64K 字节堆栈和 128K字节数据存贮. 很多应用程序只要预置段寄存器就可以了，但对于一些比较大的应用程序，则需要仔细地考虑给出段的定义.

8086 存贮空间的分段结构，有力地支持了模块化的软件设计，以避免搞巨大的、单块的程序. 在许多程序设计情况下，利用分段技术是有益的. 因为这样程序涉及的将只是逻辑地址而

不是实际地址，它不需要事先知道要产生的代码在存贮器的具体位置，这样可以简化存贮器资源的动态管理。所以说 8086 的存贮器分段为模块化、结构化的程序设计提供了物质基础。

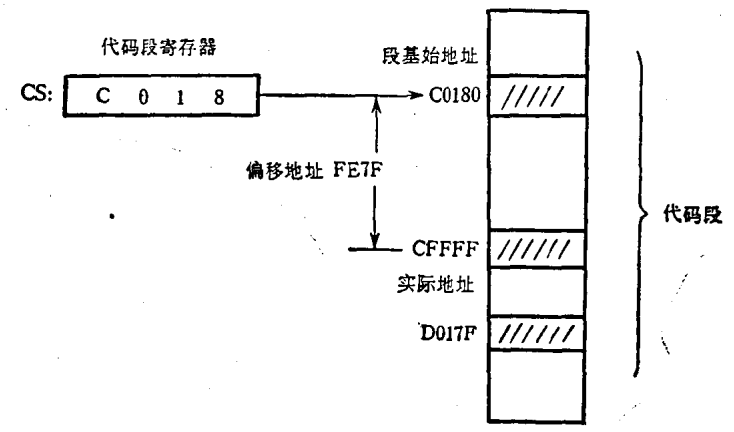


图 2.7 产生字节地址的例子

§2.4 输入/输出结构

8086 系统与外部设备相联系的机构称为转接口。正是通过这些转接口，8086 才能接收外部的信息（例如，旅客的座位皮带是否已经扣好）和发送控制信号（例如，防止汽车发动而伤害司机）以及输出各种信息。

8086 可以访问多至 2^{16}（约 65000）个类似于存贮器字节的 8 位转接口。每个转接口分配范围从 0 到 $2^{16}-1$ 的唯一的地址。类似于存贮器字的构成，任何 2 个相邻的 8 位转接口，可以组合成一个 16 位的转接口，并且象存贮器字一样，对位于奇地址的 16 位转接口的访问要进行 2 次才能完成。事实上，转接口的寻址方法，除了没有转接口段寄存器以外与存贮器的字节或字的寻址是一样的。换句话说，所有的转接口是在一个段中。

8086 具有从输入转接口中，读取信息和把信息写到输出转接口的专用指令。输入/输出转接口也可以放在存贮空间中，以便充分利用 8086 整个指令系统和寻址方式的能力，去进行输入/输出处理。

当 8086 采用最小模式构成系统时，8086 会提供 HOLD（保持）和 HLDA（保持响应）信号（关于这两个信号在 §2.8 中介绍），这些信号同 8257 和 8237 等系统的 DMA 控制器是相容的（DMA 是直接存贮器访问的英文缩写）。DMA 控制器通过发送 HOLD 信号请求使用总线，以便在输入/输出设备和存贮器之间直接传送数据。如果总线周期正在进行中，则CPU 将在完成现行的总线周期后发出 HLDA 信号，批准 DMA 控制器使用总线，等到 HOLD 信号变成无效以后，CPU 才试图使用总线。

在高性能的输入/输出应用中，8086 与 8089 一起运用。8089 在原理上象是带有两个 DMA 通道的处理器，其指令系统是特为输入/输出操作而设计的。8089 与简单的 DMA 控制器不同，它可以直接为输入/输出设备服务，而 8086 就不再承担这项任务。此外，它能在本身的总线或系统总线上传送数据，使 8 位或 16 位的外围设备与 8 位或 16 位的总线匹配，并且执行从存贮器到存贮器或从输入/输出设备到输入/输出设备传送数据的任务（详见第二篇）。

§2.5 寄存器结构

8086处理器共含有13个16位寄存器和9个标志位。为了便于描述，把这些寄存器分成4个集。这些集中的3个每个含有4个寄存器。第13个寄存器，即指令指示器，是不能被程序员所直接访问的，所以它自成1集。8086寄存器和标志表示在图2.8(a)中。

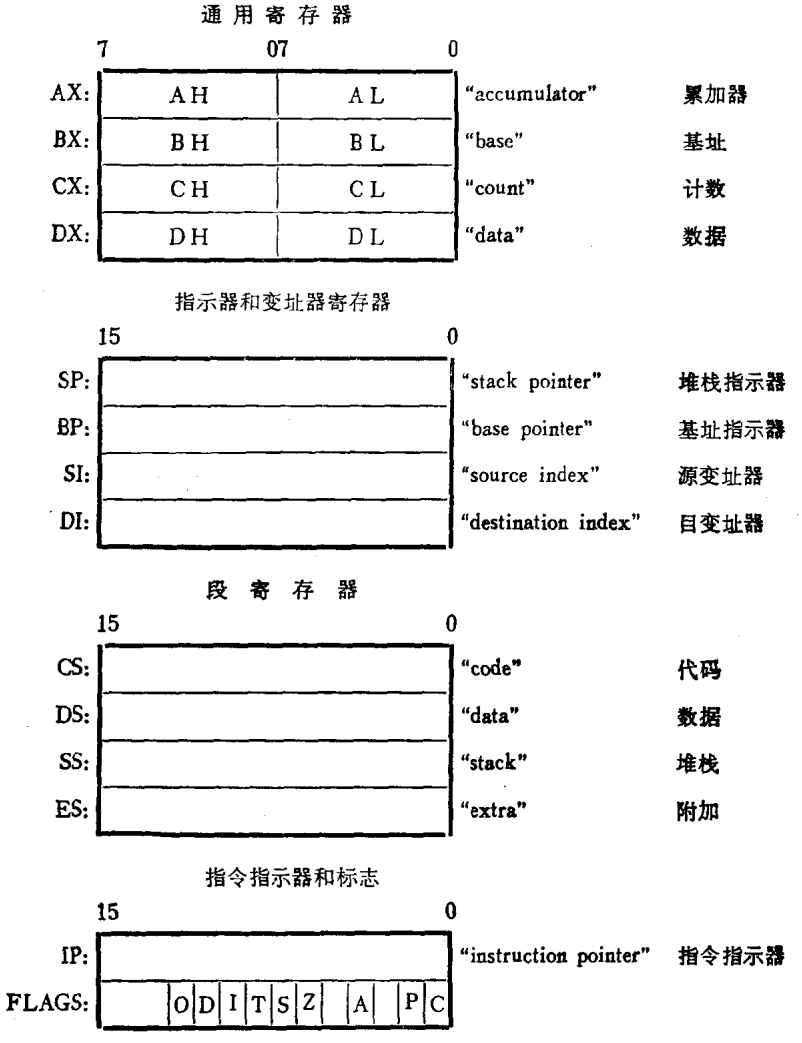


图2.8(a) 8086寄存器和标志

由于8086对于8080/8085是向上兼容的，所以8080/8085的寄存器集必然是8086寄存器集的一个子集。8080/8085中的寄存器、标志位和指令指示器在8086中都有其对应物。8080/8085中的累加器对应于8086中的AL寄存器；8080/8085中的H和L，B和C，D和E寄存器分别对应于8086中的BH，BL，CH，CL，DH和DL寄存器；8080/8085中的SP（堆栈指示器）和PC（程序计数器），在8086中对应为SP和IP。AF，CF，PF，SF和ZF标志在两个CPU系列中是相同的，其余的标志位和寄存器在8086中是独有的。这种

8086 对 8080/8085 的映象设计，使大多数现有的 8080/8085 的程序代码，可以直接翻译成 8086 的代码。它们的对应关系在图 2.8(b) 中表示，其中涂有阴影的寄存器是 8080/8085 寄存器的对应物。

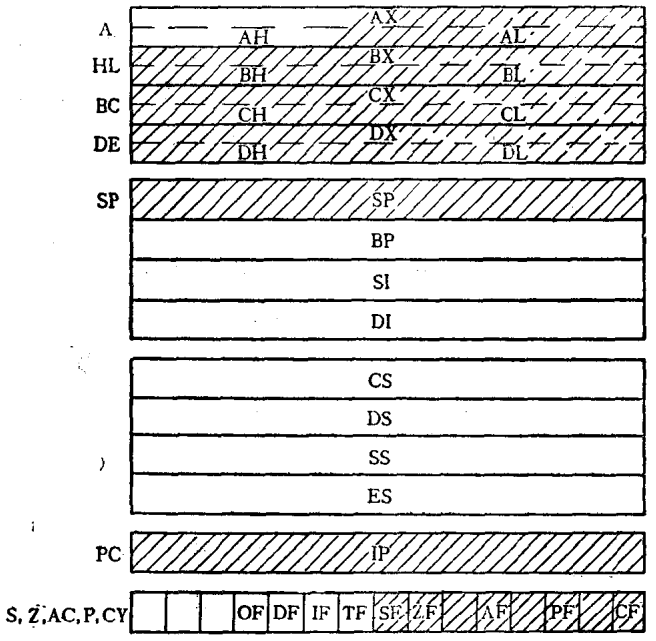


图 2.8(b)　8080/8085 的寄存器集是 8086 寄存器集的一个子集(阴影部分)

可访问的寄存器的 3 个集是通用寄存器、指示器、变址器寄存器和段寄存器。通用寄存器主要用来存放算术和逻辑运算的操作数；指示器、变址器寄存器用来存放段内的偏移地址；段寄存器用来指定段的起始地址。

一、通用寄存器

在一个没有通用寄存器的处理器中，每一条指令要从存贮器中取出它的操作数，并把操作后的结果放回存贮器。由于存贮器访问是比较费时间的，所以若能把那些经常使用的操作数和结果暂时地保存在一些快速的单元中，就可以减少所需要的时间。8086 处理器中的通用寄存器集，就是这样一些快速访问单元。

8086 的通用寄存器是 16 位的寄存器 AX，BX，CX 和 DX。每个通用寄存器的高半部和低半部可以分开作为两个 8 位寄存器或在一起作为 16 位寄存器使用。因此通用寄存器的每个半部都有自己的名字。低半部被命名为 AL，BL，CL 和 DL；高半部则被命名为 AH，BH，CH 和 DH。这些寄存器的双重性，使它们在处理字节量时和处理字量一样方便。

在大多数情况下，这些通用寄存器的内容，可以互换地参与 8086 的算术和逻辑操作。例如，ADD (加法) 指令可以把任何 8 位或 16 位通用寄存器的内容同其它通用寄存器的内容相加，并把结果存入这两个寄存器中的任意一个。但是，有少量的 8086 指令，它们把某些通用寄存器作为专用。例如，串指令需要 CX 寄存器含有串中元素个数的计数，而 AX，BX 和 DX 却不能用于这个目的。由于 CX 寄存器的这个特殊用法，我们就把 COUNT (计数) 的缩写，作为该寄存器的名字。同理，关于 AX，BX 和 DX 寄存器 (在后面叙述) 则用 ACCUMULATOR (累加器)，BASE (基址) 和 DATA (数据) 的缩写来命名这些寄存器。

通用寄存器的某些特殊用法，似乎会使处理器更难于被掌握，因为在编制程序时，要记住很多特殊的规则。从表面上看，程序可能会变得更长，因为需要在执行某些指令前把数据从一个通用寄存器中转移到专用的那个寄存器中去。然而，有经验的程序员，决不会为处理器写一个在任何时刻把所有的通用寄存器都等同看待的程序。为了掌握程序运行的进程，他们一定会这样来组织它们的程序：使特殊种类的数据总是驻留在特殊的寄存器中。若我们总是选择 CX 寄存器来记录串中的元素数目，就再也用不到把串的长度移入到 CX 寄存器中去了。若串指令可以从任何通用寄存器中取得串的长度，则每一条串指令必须要说明到哪一个通用寄存

器中去找串的长度。当然，这可以用增加每条串指令的长度（用两个字节代替一个字节），或是用更多的一字节串指令来实现。第一种解决办法会使程序代码长度增加；第二种解决办法，实际上也使代码长度增加，因为只有很少数量的一字节指令（256 条）。有更多的一字节串指令，就意味着一些其他的一字节指令将增长为 2 字节指令。由此可见，8086 采用使某些寄存器专用于某些指令的做法，实际上可使程序的代码总长度缩短。

二、指示器和变址器寄存器

一条访问存贮器的指令，可以直接说明该单元的地址。这个地址要占用指令空间，因而会增加指令代码的长度。如果把经常使用的单元的地址，存放在特定的寄存器中，访问这些单元的指令，就再也用不到含有地址，而只要指定该寄存器就可以了。这种寄存器有时称为指示器或变址器寄存器。

寄存器的这种用法，在生活中也可以找到例子，它很象压缩的电话拨号。假设拨 7 位电话号码可以同所在城市里的任何人通话，这相当于直接说明地址；在某些地方电话局提供这样的服务，即用户可以把经常要使用的电话号码事先存入一个“寄存器”集中，然后，需要时只要拨一两个指定寄存器的数字，电话就可以接通了，显然，这相当于通过寄存器说明地址。

8086 的指示器和变址器寄存器由 16 位寄存器 SP、BP、SI 和 DI 所组成。这些寄存器通常含有在某一段内寻址的偏移地址。例如，一条 ADD 指令可以在当前数据段中指定它的一个操作数，办法之一，就是把该操作数的偏移量，放在一个指示器或变址器寄存器（假定是 SI）中去。

使用指示器和变址器寄存器，除了有减少指令长度的好处之外，还有更重要的作用，即它们使指令可以访问这样的单元：其偏移地址是前面程序执行运算的结果。在高级语言程序中，为了建立变量的偏移地址，常常要执行这样的运算。这些运算可以在一个通用寄存器中执行，然后把结果传送到一个作为偏移量用的指示器或变址器中去，但这样做的直接后果是增加了代码的长度。为了消除不必要的传送和缩短代码，8086 的指示器和变址器所含有的值，可以与 16 位的通用寄存器一起参加算术和逻辑运算。这样，前面提到的 ADD 指令可以指定它的另一个操作数是 DI 寄存器的内容。

这些寄存器在使用中并不是完全一致的，它们相互之间有一些差别，这些差别把这一集寄存器分为指示器寄存器 SP 和 BP，以及变址器寄存器 SI 和 DI。指示器在运算中，为访问当前堆栈数据提供了方便。把堆栈段用作一个“数据区”，对于执行高级语言很有用处（关于这一点将在本章的 § 2.7 中讨论）。这样，除非特别指定，存放在指示器中的偏移量，总是假定为引用当前的堆栈段，而存放在变址器中的偏移量，则一般是假定为引用当前的数据段（在这里用了“一般”这个词就意味着还有例外的情况）。例如，一条 ADD 指令指定 SI 含有它的一个操作数的偏移量，除非加法指令直接指定一个其他的段，否则操作数是在当前的数据段中。

有一些指令是把两个指示器 SP 和 BP 加以区分的，PUSH 和 POP 指令从 SP 寄存器中取得关于堆栈顶单元的偏移量，所以，很自然地，这个寄存器的名字就是 STACK PIONTER（堆栈指示器）的缩写 SP。BP 寄存器不能用于这个目的，但却可以用它来存放在堆栈段中的一个数据区的“基址”的偏移量，所以给它起的名字就是 BASE POINTER（基址指示器）的缩写 BP。

此外，串指令对两个变址寄存器 SI 和 DI 的使用也作出区分。串指令需要一个偏移量放

在 SI 寄存器中的源操作数和一个偏移量放在 DI 中的目操作数。这就是用 SOURCE INDEX (源变址器) 和DESTINATION INDEX (目变址器) 的缩写 SI 和 DI 给它们命名的原因。对于这些串指令，SI 和 DI 所扮演的角色，是不能互换的。例如，串传送指令把在当前数据段中开始于在 SI 中的偏移量的串，传送到当前的附加段 (这就是前面提到过的例外)，而该段中的偏移量是存放在 DI 中的。应当指出的是，SI 和 DI 寄存器是隐含地指定的，串指令不能直接地指定它们。目的串在附加段中，而不是在数据段中，使每个串可以有它自己的一个段，并且最多能达到 2^{16} 字节的长度。在这里，我们又一次看到，对某些寄存器规定某些特殊的用途是有好处的。

三、段寄存器

8086 具有 1 兆字节的存贮器，但放在指令指示器和变址器寄存器中的地址只有 16 位长，这些地址不能在 1 兆存贮器中寻址，而只能是在一个特定的 64K 字节的段中的偏移量寻址。问题是，这种偏移量寻址在哪一段中进行呢？这个问题就涉及到 8086 的分段技术。

8086 有 4 个 16 位段寄存器 CS, DS, SS 和 ES，这些段寄存器，用来识别当前可寻址的 4 个段，它们不可互换地使用：CS 识别当前的代码段；DS 识别当前的数据段；SS 识别当前的堆栈段；而 ES 则识别当前的附加段。它们的对应关系如表 2.1 所示。

表 2.1 段寄存器和当前段的对应关系

寄存器	含　义	隐含识别的当前段
CS	代码段寄存器	当前代码段
DS	数据段寄存器	当前数据段
SS	堆栈段寄存器	当前堆栈段
ES	附加段寄存器	当前附加段

一条指令指定了一个段内的偏移量，并且段寄存器指定了可供使用的 4 个段，而选择哪一段，则依赖于这些偏移量是如何被使用的。一个偏移量可能是指定下一条要执行的指令或是一个操作数，下面就这两种情况分别加以说明。

所有的指令都是从当前的代码段中取出的，因此就需要一个含有下一条要执行的指令在当前代码段中的偏移量的寄存器。这个寄存器就是 IP，即 INSTRUCTION POINTER (指令指示器) 的缩写。例如，若 CS 含有 16 进制数 1FF7 并且 IP 含有 16 进制数 003A，则被访问的下 1 条指令将在地址为 1FFAA 的单元中，因为

```
  1FF70      代码段起始地址
+ 003A       在 IP 中的偏移量
  1FFAA      下 1 条指令的存贮器地址
```

(可以回想图 2.6，在产生存贮器地址的时候，16 进制数字 "0" 被加在段寄存器中的值的后面)。

取操作数的段，一般地可以用在指令前放一个 1 字节前缀的方法来指定。这个前缀指定从4个段中的哪一个段中取操作数。如果没有这样一个前缀 (一般情况)，操作数取自当前的数据段，除非：

(1) 偏移地址是从一个指示器的内容中计算得到的，在这种情况下，使用当前的堆栈段；

(2) 操作数是一条串指令的目的操作数，在这种情况下，使用当前的附加段。

例如，考虑一条 ADD 指令，它的一个操作数在数据段中并且偏移量在 SI 中。该指令在它的操作数字段中只指定 SI，而不指定 DS。当执行该指令时，处理器会自动地用 DS 的内容

和 SI 的内容,--起去共同确定该操作数的位置。接着,考虑一条 ADD 指令,它的操作数在代码段中(可能是在 ROM 中的常数)并且偏移量在 SI 中。这条指令将会象前面一条指令一样,在操作数字段中指定 SI,但是,另外将有一个前缀字节放在指令的前面以指定 CS (关于这一点,在§2.6 的寻址方式中还要加以讨论)。

四、标志位

8086 含有 9 个标志位,它们用来记录处理器的状态信息 (状态标志),或控制处理器操作 (控制标志)。状态标志通常在算术或逻辑指令执行之后设置,以反映这种操作的结果的某种性质。这些标志是进位标志 (CF),指出该指令是否在最高位产生 1 个进位;辅助进位标志 (AF),指出该指令是否在低 4 位产生 1 个进位;溢出标志 (OF),指出该指令的执行是否产生 1 个越出范围的带符号结果;0 标志 (ZF),指出该指令是否产生全 0 结果;符号标志 (SF),指出该指令是否产生 1 个负的结果;校验标志 (PF),指出该指令是否产生 1 个具有偶数个 "1" 的结果。

控制标志是方向标志 (DF),它控制串处理指令的方向;中断允许标志 (IF),它允许或不允许外部可屏蔽中断;自陷标志 (TF),它为了方便程序的调试,而使处理器的执行进入单步方式。图 2.9 说明了各个标志位的含义。

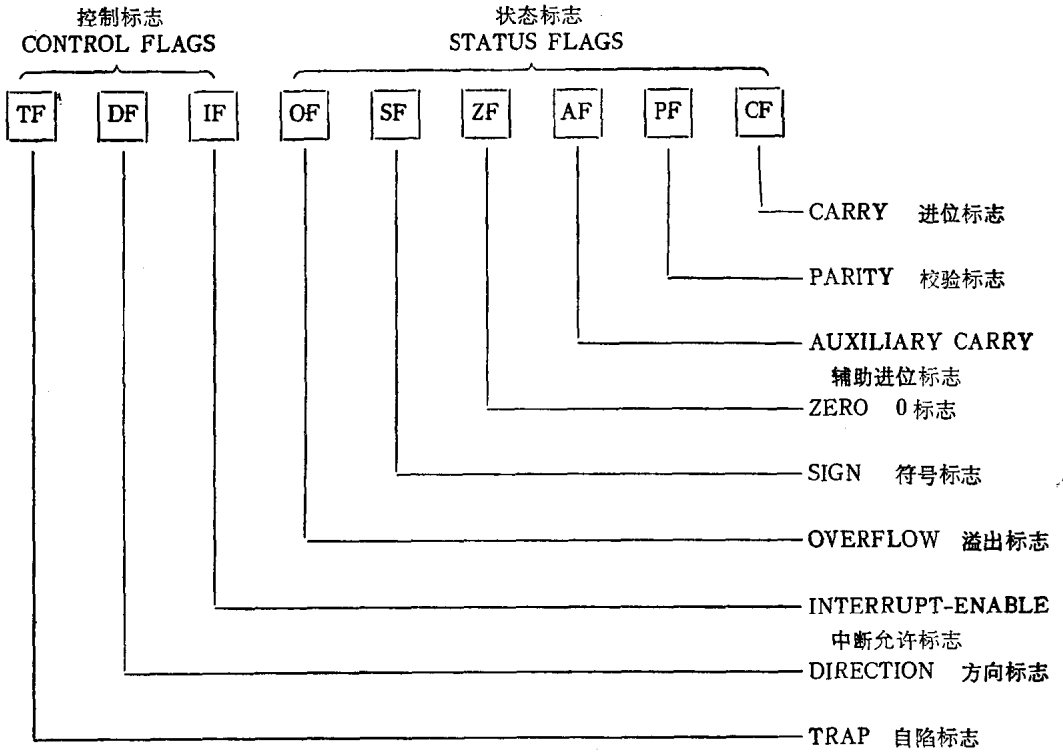


图 2.9 标志位

关于每一个标志位的详细情况,将在整个第 3 章中给出,而该章的§3.11 则概括了这些标志的特性。

§2.6 指令操作数和操作数寻址方式

8086的指令通常在1个或2个操作数上执行操作。例如，ADD指令是把第1个操作数中的值加上第2个操作数中的值，并把结果放回其中的一个中去。INCrement指令是把操作数的值加1，并把这个结果放回到该操作数中去。下面，我们就来详细地讨论8086的寻址方式。

一、单操作数

1条只指定1个操作数的指令称为单操作数指令。例如，INCrement指令。INCrement指令的最经常的用法，是把指示器或变址器寄存器（在计算偏移地址时）或16位通用寄存器（在执行算术运算时）的内容加1。对于这样十分频繁的操作，该指令采用了1种非常简单的1字节形式，如图2.10所示。它含有1个3位的reg字段，用来指定8个16位寄存器中的1个（通用，指示器或变址器寄存器）。在reg字段中所使用的编码在表2.2中表示。剩下的5位为操作码。在INCrement的情况下，操作码是01000。例如，把BP寄存器的内容加1的指令在图2.11中表示。这种操作数寻址方式，有时称为寄存器寻址方式。

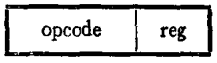


图2.10 单操作数指令其中操作数是在1个16位寄存器中(图中opcode表示是指令中的操作码字段，reg表示是指令中的寄存器字段，以下不再特别说明)。

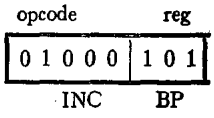


图2.11 把BP寄存器加1的指令

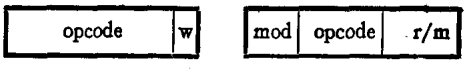


图2.12 操作数是在一个寄存器或存贮器中的单操作数指令

在最一般的形式中，INCrement指令，可以命令任何通用寄存器、指示器、变址器寄存器(8位或16位)或任何存贮器的字节或字加"1"。这种形式是2字节长。图2.12表示了这种指令。现在操作码字段已被分裂，操作码的7位在第1个字节中，3位在第2个字节中。这种形式的操作码是1111111,000。其中w字段指定操作数的宽度。如果w=0，操作数是8位的，否则是16位的。mod字段指定操作数是在寄存器中还是在存贮器中。若mod=11，操作数在寄存器中，否则在存贮器中。若操作数在寄存器中，r/m字段指明是在哪一个寄存器，若操作数在存贮器中，r/m字段指明是在存贮器的什么地方（r/m是register/memory的缩写，代表寄存器或存贮器）。

表2.2 寄存器的编码

reg mod=11 r/m	w=1	w=0
000	AX	AL
001	CX	CL
010	DX	DL
011	BX	BL
100	SP	AH
101	BP	CH
110	SI	DH
111	DI	BH

先考虑操作数在一个寄存器中的情况(mod=11)。r/m字段中使用的寄存器编码在表2.2中表示。这实际是1个寄存器操作数寻址方式。例如，把CL寄存器中的内容加1的指令在图2.13中表示。

现在考虑操作数在存贮器中（mod=00，01或10）的情况，这种操作数寻址方式，有时称为间接存贮器寻址，因为操作数虽然在存贮器中，但偏移

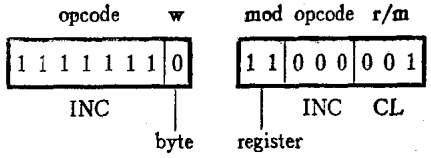


图 2.13 CL 的内容加 1 的指令(其中 byte 表示是字节，register 表示是寄存器)

量却不是直接指定的。操作数的偏移量是把一组看起来有些奇怪的值相加而取得的（这种方式的用途将在下一节中证明）。在这种寻址方式下，操作数的偏移量可以是多至三个数的和：

(1) 一个 16 位值(称为位移量)，它在指令中指定。

(2) 在指令中指定的一个变址器寄存器的内容 (SI, DI 或无)。

(3) 在指令中指定的一个基址寄存器的内容 (BX, BP 或无)。

r/m 字段如在表 2.3 中所示的那样指定基址和变址寄存器。mod 字段如表 2.4 所示的那样指定位移量。这样形成的偏移量才在操作数所在段内确定操作数的位置。除非指示器 BP 的内容在计算偏移地址中使用 (这种情况下，操作数在当前的堆栈段中)，操作数是在当前的数据段中。另外，还涉及到一个段寄存器的内容，它是形成该操作数的 20 位地址所必需的。

表 2.3 mod≠11 时 r/m 字段的编码

r/m 字段	基址寄存器	变址寄存器
000	BX	SI
001	BX	DI
010	BP	SI
011	BP	DI
100	无	SI
101	无	DI
110	BP	无
111	BX	无

如果 mod=00 并且 r/m =110，见表 2.4 下面的注

表 2.4 操作数在存贮器中时 mod 字段的含义

mod	位移量	注释
00	零位移量(16 位宽)。	
01	指令的下 1 字节的 8 位内容符号扩展至 16 位	指令含有 1 个附加的字节
10	指令的下两个字节的 16 位内容(下 1 字节是位移量的低 8 位，后面是位移量的高 8 位。)	指令含有 2 个附加的字节。

注：若 mod=00 并且 r/m=110，则：
1. 表 2.3 和表 2.4 不适用
2. 指令含有 2 个附加字节
3. 偏移地址包含在这 2 个字节中(低 8 位在高 8 位的前面)

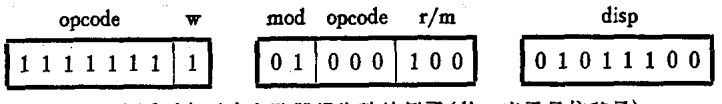


图 2.14 1个存贮器操作数的例子(disp 表示是位移量)

例如，考虑图 2.14 所示的指令。操作码字段是 1111111000，它是 INCrement 指令。w 字段是 1，它表示操作数的值是 16 位的。mod 字段是 01，表示操作数是在存贮器中，而且位移量是在指令的下 1 个字节中(要符号扩展到 16 位)。这样位移量就是 0000 0000 0101 1100。r/m 字段是 100，它指定变址寄存器 SI 的内容要加到位移量上去以形成偏移地址。假定 SI 的内容是 1010 0000 1000 0110，则该偏移地址可以这样得到：

```
  1010 0000 1000 0110      SI 的内容
+ 0000 0000 0101 1100      位移量
-------------------
  1010 0000 1110 0010      偏移地址
```

由于在计算偏移地址时没有用到 BP，所以应当引用当前的数据段。假定 DS 为 1111 0000 1111 0000，则操作数的存贮器地址如下：

	1111 0000 1111 0000	数据段
+	1010 0000 1110 0010	偏移地址
	1111 1010 1111 1110 0010	存贮器地址

这个操作数是 16 位宽的(由w字段指定)，所以该操作数是在地址 1111 1010 1111 1110 0010 和地址1111 1010 1111 1110 0011 中的内容，并且较高的地址字节是高八位。这个实际地址形成的过程，表示在图 2.15 中

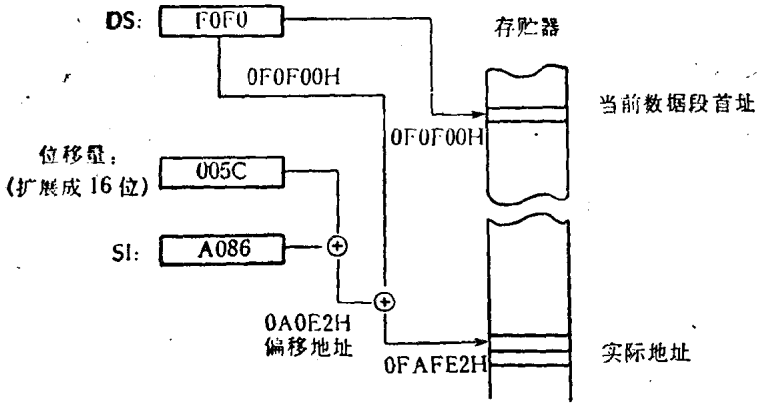


图 2.15 实际地址形成的过程

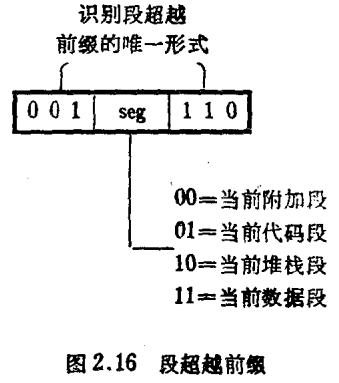


图 2.16 段超越前缀

前面介绍段寄存器时曾经指出，操作数不一定局限在当前的数据段或堆栈段中，通过在指令前放 1 个指定段寄存器的前缀，即可从 4 个当前段中的任何 1 个中取得操作数。我们把这个前缀称为段超越前缀。1 字节段超越前缀在图 2.16 中表示。

图 2.17 是一条与图 2.14 所表示的相同的指令，不过，现在操作数是在附加段里。

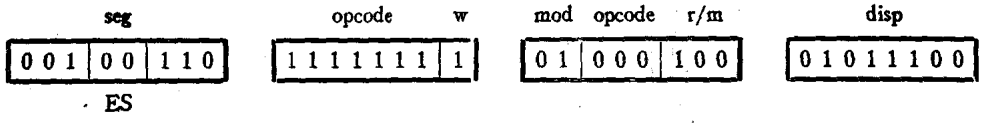


图 2.7 使用段超越前缀的例子

至此，我们已经说明了如何通过一个基址和/或变址寄存器来指定一个在存贮器中的操作数的偏移量。但是，我们常常确切地知道操作数在哪里，并且想要在指令中加以直接指定。这种操作数寻址方式，称为直接存贮器寻址。使用这种方式寻址，偏移量必须放在指令的两个字节中，并且指令的其余部分必须指定操作码和说明是存贮器直接寻址。用 mod 和 r/m 字段的组合来说明这种方式，应该是很方便的，遗憾的是，所有的组合编码都已经被间接存贮器方式和寄存器方式用掉了。然而，这些组合中的一种不经常使用的间接存贮器寻址的编码，可以选择来作为直接存贮器寻址的编码。这种组合是 mod=00，r/m=110。例如，把在当前数据段，其偏移量为 0101 1010 1111 0000 单元的内容加 1 的指令如图 2.18 所示。

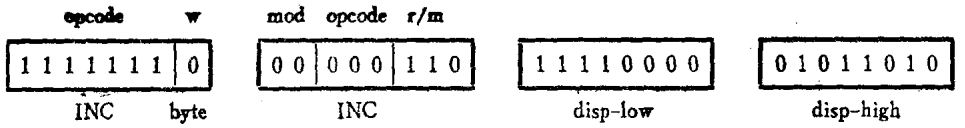


图 2.18 把在当前数据段，其偏移量为 0101 1010 1111 0000 的字节加 1 的指令
(disp-low, disp-high 分别为偏移量的低 8 位和高 8 位.)

二、双操作数

在已经掌握了单操作数指令的基础上，我们考虑一条有两个操作数的指令 ADD。如前面所提到的那样 ADD 指令取得一个操作数的值，把它与另一个操作数的值相加，然后把结果放回到指令所指定的单元中去。如果两个操作数都可以在存贮器中，指令就要给每个操作数以一个 mod 字段和 r/m 字段。为了使指令短些，8086 规定，至少有一个操作数必须是在寄存器中。这样，指令只需要给其中一个操作数以 mod 和 r/m 字段，而对另一个操作数只要一个 reg 字段就可以了。典型的双操作数指令在图 2.19 中表示。

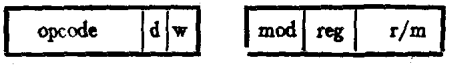


图 2.19 典型的双操作数指令

双操作数指令用 w 字段来指定操作数是 8 位 (w=0)，还是 16 位 (w=1)。还多了 1 个前面没有出现过的字段，即 d 字段 (d 代表目的)。该 d 字段决定结果应当放回到由 mod 字段和 r/m 字段 (d=0) 指定的操作数单元，还是放到由 reg 字段指定的操作数单元 (d=1)。将要存放结果的操作数称为目操作数，另一个操作数则称为源操作数。

例如，考虑如图 2.20 所示的 ADD 指令。ADD 的操作码是 000000。w 字段是 0，说明 2 个操作数都是 8 位，由 reg 字段指定的操作数为 CH。mod 字段是 11，指定 mod，r/m 操作数是在一个寄存器中，并且 r/m 字段识别出寄存器为 BL。d 字段指定结果要放回到由 reg 字段指定的操作数中，即 CH 中。这样，该指令是把源操作数即 BL 寄存器的内容加上目操作数即 CH 寄存器的内容，然后把结果放回到 CH 中。

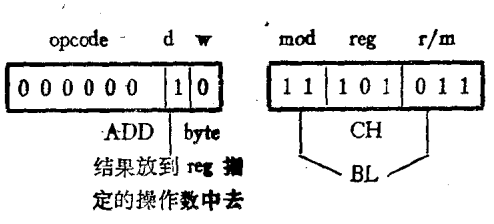


图 2.20 双操作数指令的例子

双操作数指令中的一个操作数可以是放在指令中的常数，这样的常数称为立即操作数。例如，MOVe 可以是一条立即操作数指令。这条指令最通常的用法是把一个常数移入一个寄存器 (通用，指示器或变址器)。在这种情况下，非立即操作数可以由一个 reg 字段来指定，并且指令采取如图 2.21 所示的简单形式。w

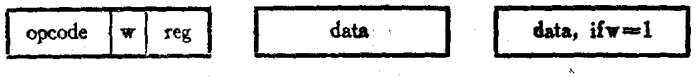


图 2.21 最简单的立即操作数指令(data 表示数据)

字段表示操作数 (立即和非立即是一样的) 是 8 位的 (w=0) 还是 16 位的 (w=1)；若是 8 位的，立即操作数在指令中占一个字节，否则占两个字节，并且以“反字”存放。例如，图 2.22 表示一条把值 1111 0000 0000 1111 传送到 16 位 DI 寄存器中去的指令。

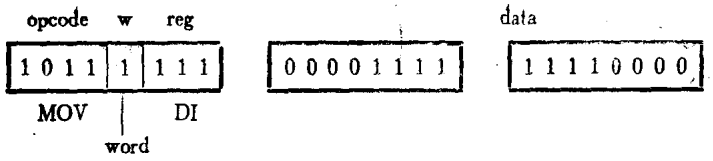


图 2.22 立即操作数指令的例子

稍微复杂一点的立即操作数指令，使用 mod 和 r/m 字段而不是 reg 字段来指定非立即操作数。这是一种更通用的(非立即操作数可以在存贮器中)指令，但需要给指令增加一个字节，如图 2.23 所示。如果指令中的 mod，r/m 字段指出该指令具有立即偏移量或位移量附加字节，则立即操作数附加字节跟在这些字节之后。图 2.24(a) 表示一条把值 1111 0000 0000 1111 传送到偏移量在 DI 中的当前数据段中的字单元中去的指令。图 2.24(b) 则表示字单元的偏移量由 BX、DI 的内容及二字节位移量之和来决定的执行同样功能的指令。在图中假设位移量是 0000 0001 1000 1010。

图 2.23 使用 mod 和 r/m 字段的立即操作数指令

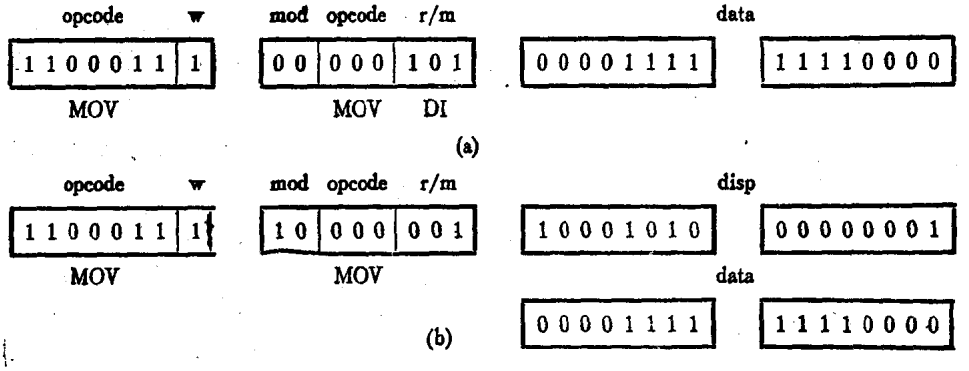


图 2.24 使用 mod 和 r/m 字段的立即操作数指令的例子

由于双操作数指令只有一个 w 字段，两个操作数必须同时是 8 位的或是 16 位的。然而，立即操作数的值往往比较小，因而用 8 位来表示也就足够了，这种情况在使用立即操作数的加法、减法和比较指令中出现得特别多 (对于使用立即操作数的逻辑指令则不一定)。由于在程序中使用立即操作数的指令是比较大量地出现的，我们就可以想象，若不用 16 位来放较小的数，就可以减少立即操作数指令的长度，从而就能有效地压缩整个代码的长度。为了实现这一点，一些立即操作数指令(加法、减法和比较指令)含有一个 s 字段(s 的意思是符号扩展)。这个字段只对 16 位操作数有意义 (即 w=1 时)，并且表示立即操作数的 16 位都在指令中 (s=0)，还是只有低 8 位在指令中并且必须要符号扩展以形成 16 位操作数 (s=1)。这种形式的指令格式在图 2.25 中给出。

图 2.25 含有 s 字段的立即操作数指令

图 2.26 表示一个例子．在这个例子中，值 0000 0000 0000 1111 要和存贮器中一个字的内容相加，并把结果放回到存贮器字中去．该存贮器字是在数据段中并且偏移量在 DI 中．注意，有了 s 字段后，一个字节被省掉了．

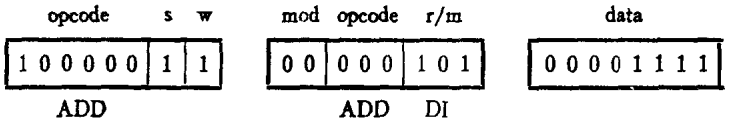


图 2.26 含有 s 字段的立即操作数指令的例子

§2.7 关于操作数寻址方式的说明

一、各种寻址方式都是需要的

在看了前面关于操作数寻址方式的描述以后，读者可能会提出以下问题：

1．在每次使用具有操作数的指令时，是否真需要填入 mod、r/m、reg、w、s、d 等字段？

2．为什么要有那么多的存贮器寻址方式？第一个问题的答案是否定的．除非用机器码来编写程序，否则这些工作是由汇编程序或编译程序做的，它对于程序员来说是透明的．

要回答第二个问题，首先应该想到 8086 是为了能使用高级语言，并能把它翻译成高效的机器代码而精心设计的微处理器．下面，我们将针对高级语言的典型特征进行讨论，以证明 8086 的每一种寻址方式对于支持高级语言的使用都是必不可少的．

大多数程序设计语言都具有简单变量和数组的概念．简单变量是代表单值的变量，数组是代表一个值序列的变量．下面是一条在许多高级语言里找得到的典型的赋值语句：

$$A(I) = X$$

这句语句读作“把 X 的值赋给数组 A 的第 I 个元素”．翻译成代码，它把对应于简单变量 X 的存贮器单元的内容，传送到一个寄存器，设为 BL 中，然后把 BL 的内容传送到对应于数组 A 中的第 I 个元素的存贮器单元中去．假定 X 是在当前数据段中偏移量为 0FF0H 的存贮器单元的内容，另外假定数组 A 的第一个元素 A(0)，是在该段中的偏移量为 0FF1H 的单元．把 X 的内容传送到 BL 的机器指令在图 2.27(a) 中表示．这里利用了选择存贮器直接寻址的

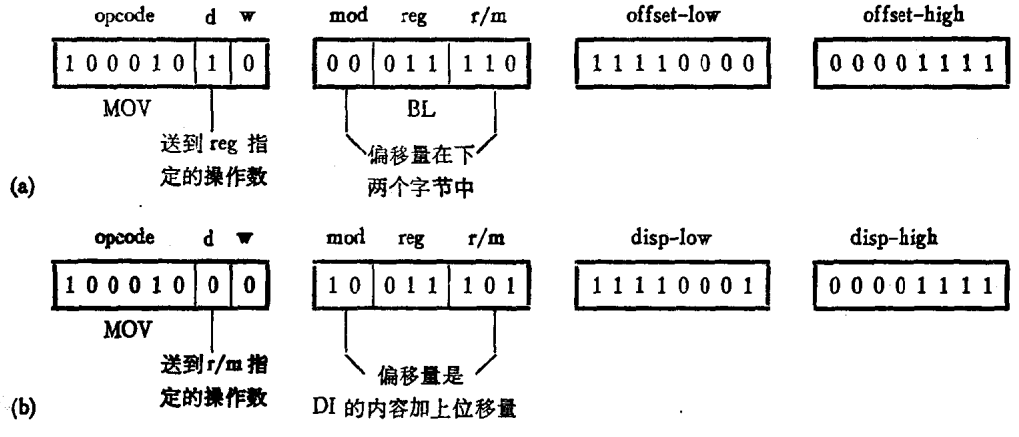


图 2.27 关于 A(I)=X 的机器指令．(a) 把 X 传送到 BL．(b) 把 BL 的内容传送到 A(I)．
(图中 offset 表示偏移量，offset-low offset-high 分别表示偏移量的低 8 位和高 8 位)

mod 和 r/m 的特殊情况（由于访问一个简单变量，如 X，是经常发生的，所以提供一种特殊的寻址方式也就不足为奇了）。把 BL 的内容传送到A(I) 中的机器指令，在图 2.27(b) 中表示。这里假定变址值 I 已经在变址寄存器中，显然，这样的数组访问需要间接存贮器寻址方式“变址寄存器＋位移量”。A(I)＝B(J) 形式的赋值语句，至少需要两个变址寄存器，每一个具有上面提到的寻址方式，即“SI＋位移量”和“DI＋位移量”。访问一个诸如 A(I+2) 这样的数组元素，只要使图 2.27(b) 中的位移量字段含有 0FF3H，即是 A(2) 的偏移量而不是 A(0) 的偏移量就可以了。

某些高级语言具有有基变量的概念。一个有基变量对应于一个存贮器单元，它的地址放在另一个称为指示器的变量中。若指示器的值（即对应于该指示器的存贮器单元所包含的值）改变了，则该有基变量将对应于不同的存贮器单元。访问有基变量的一种方便的办法，是把指示器的值放到 BX 中，然后使用含有 BX 的操作数寻址方式。例如，方式“BX”可用来访问一个简单的有基变量，而方式“BX＋SI”或 BX＋DI”可用来访问一个有基数组中的元素。

一些高级语言使用一种叫做记录的数据类型。记录（在另一些语言中也称为结构）是一个可以是不同类型的有名数据项的集合。这一点和数组不同，数组是一个具有相同类型的数据项的序列（无名的）。例如，薪水帐目程序，对于每一个职工有一个记录。每一个记录可能含有职工的名字、证件号、录用年数和工资数。某一个记录项，如录用年数，在每一个职工的记录中总是在同一个地方。例如，录用年数是在每个职工记录的从头数起的第 4 个字节，并且关于某一职工的录用记录开始于偏移量 03B4H，则该职工的录用年数在偏移量为 03B7H 的存贮单元中。这样，在一个记录里的任何给定项，总是在一个固定的位置，并且可以用直接寻址来访问，从本质上来说，它与一个简单变量没有什么不同。

现在考虑一个有基记录，假定该记录的基址指示器值在 BX 寄存器中。在这样一个记录中，访问一个项目的操作数寻址方式应该是“BX＋位移量”，这里位移量是在记录中对应于该项的位置。例如，项目若是在一个有基职工记录中的录用年数，则位移量是 3。除非记录非常大（多于 256 个字节），否则，位移量可以用一个字节来表示，从而能使用 mod=01 的寻址方式。

虽然访问有基记录中的项目的操作数寻址方式和访问数组元素的寻址方式十分相似（都是“寄存器＋位移量”），实际上却有很大的区别。在数组元素的情况下，位移量对应于数组的开头，寄存器对应于进入数组的距离。在有基记录的情况下，寄存器对应于记录的开头，而位移量对应于进入记录的距离。

数组和记录还可以组合起来。考虑一个数组，该数组的每一个元素是一个雇员记录，另外，再假设这是一个有基数组。假定基指示器在 BX 中，对应于一个数组元素的变址值在 SI 中。这样，访问被变址的那个记录的雇用年数项必须要用的操作数寻址方式是“BX＋SI＋位移量”，在这里，位移量将是 3。对这种数组元素中某一项的寻址，证明了 8086 最复杂的操作数寻址方式“基址寄存器＋变址寄存器＋位移量”也是需要的。

我们还要证明的是含有 BP 作为基址寄存器和相应的用堆栈段代替数据段的操作数寻址方式。这些方式在块结构语言和重入子程序的执行中很有用。重入子程序是一种子程序，它可以在执行时调用自己。这种调用可能发生在这样三种情形下：

1. 重入子程序在执行时直接调用自己。
2. 重入子程序在执行时调用了一个其他的子程序，该子程序在执行过程中又调用原来的

重入子程序.

3. 重入子程序的执行因发生中断而被挂起，在处理中断时，该重入子程序又被调用.

所有被重入子程序使用的数据（局部变量和参数），必须对于每一次并发调用该子程序有一块唯一的存贮器区域，否则，正被该子程序调用所使用的数据，很可能被下一次调用破坏. 这就意味着，在重入子程序每次被调用时，都要为该子程序的数据分配存贮器. 这样的存贮区被称为活动记录. 由于最后一个调用的子程序是第一个要完成的，所以分配这样的存贮区，堆栈就是一个适宜的地方. 每当一个子程序被调用，就通过改变堆栈指示器 SP 的内容，而把在堆栈顶的一块存贮区保留给它做活动记录. 在该子程序执行期间，保持一个指向该活动记录开头的指示器是必要的. 对在活动记录中的项目的访问，可以用含有 BP 的操作数寻址方式来实现. 在活动记录中的简单变量可以用“BP＋位移量”来寻址，而对在活动记录中的数组元素，可以用“BP＋SI＋位移量”方式来寻址. 由于 BP 包含在地址的计算中，访问将是对于当前的堆栈段进行的，而堆栈段正是活动记录的所在地.

在高级语言中存贮器寻址方式的使用总结在表 2.6 中.

表 2.6　在高级语言中直接和间接存贮器寻址方式的使用

类　型	无 基 的	有 基 的	活 动 记 录
简单变量	直 接	BX	BP＋位移量
数　组	SI＋位移量 DI＋位移量	BX＋SI BX＋DI	BP＋SI＋位移量 BP＋DI＋位移量
记　录	直 接	BX＋位移量	BP＋位移量
记录数组	SI＋位移量 DI＋位移量	BX＋SI＋位移量 BX＋DI＋位移量	BP＋SI＋位移量 BP＋DI＋位移量

二、8086 的目标是能产生高效的代码

从前面关于 8086 寻址方式的论述中，我们可以看到 8086 的寻址方式是十分丰富而又灵活的. 它利用指令中的 reg、mod，r/m，w，d，s 字段把同一条指令组织成各种不同的寻址方式，以适应不同的需要. 前面我们已经论证过，8086 的每一种寻址方式都是支持高级语言的使用所必不可少的.

初学者对于 8086 的寻址方式往往感到十分凌乱，觉得不容易掌握，其实，这正是 8086 的特色. 8086 的设计者在对大量统计数字进行分析的基础上，作出了不去追求指令形式上的规整划一，而把能生成效率最高的代码作为首要目标的抉择. 这一特点，在前面的叙述中已经看得很清楚（在关于指令系统的叙述中，读者一定会对这一点有更深的体会）. 例如，单操作数指令 INC 用 mod，r/m 字段的二字节形式本来已可包罗所有的寻址方式，但鉴于把寄存器加 1 的操作是经常出现的，所以又设置了一字节的 INC 指令，专门对 16 位寄存器进行加 1 操作. 这样做，虽然牺牲了一个代码，却收到了对整个代码长度进行压缩的结果. 8086 对于存贮器立即寻址的处置也很能说明问题. 存贮器立即寻址相对于某些间接寻址来说出现的可能性要大得多，但是 mod，r/m 的所有组合方式已被间接寻址用光了. 在这种情况下，8086 的设计者并没有为照顾表面上的规整划一作出牺牲，而是宁可打破那种看起来整齐舒服的局面，把一种不经常使用的组合方式：0 位移量＋BP＋无，即 mod＝00，r/m＝110 让给存贮器立即寻址

方式，而该种组合原来所对应的寻址方式，则纳入 mod=01，r/m=110 的形式。这样做的结果使该种寻址方式的指令，增加了一个内容为零的位移量字节，从局部来看增加了代码的长度，但由于经常出现的立即寻址方式指令的字节数减少了，从全局来讲还是压缩了代码长度。再如，在具有立即操作数的指令中设置 s 字段，虽然增加了处理上的困难，但由于大多数有关立即数加、减、比较的指令可以缩短一个字节，从整个程序的角度看，能压缩的代码字节数就是相当可观的了。压缩了代码的长度，就意味着可以减少取指令的总线操作，并能提高代码的执行效率。

了解了 8086 设计者在设计时的良苦用心，我们就再也不会对 8086 花样繁多的寻址方式感到困惑了，相反，通过研究和比较，会感到这是一种颇具匠心的设计。还有一点要提醒读者的就是，8086 是为使用高级语言而设计的处理器，它的寻址方面的细节对使用者来说，在大多数情况下是透明的。当这些繁杂的生成机器码的工作，由汇编或编译程序完成以后，展现在我们面前的就是一个高性能的处理器。

三、8086 寻址方式的总结

为了帮助读者掌握 8086 的寻址方式，特把它们归纳总结如下：

(一) 寄存器和立即操作数

只规定寄存器操作数的指令是最紧凑的，执行起来也最快。这是因为只有几位编码的寄存器“地址”就在指令中，而且其操作完全在 CPU 内执行(不需要运行总线周期)。寄存器可用作源操作数、目操作数、或同时作一条指令的两个操作数。

立即操作数是包含在指令中的常数数据。该数据可为 8 位或 16 位。因为立即操作数可以直接从指令排队机构（在后面介绍）中获得，故可以很快地选取，象寄存器操作数一样，获得立即操作数不需运行总线周期。对立即操作数的限制是，它只能用作源操作数，而且只能是常数值。

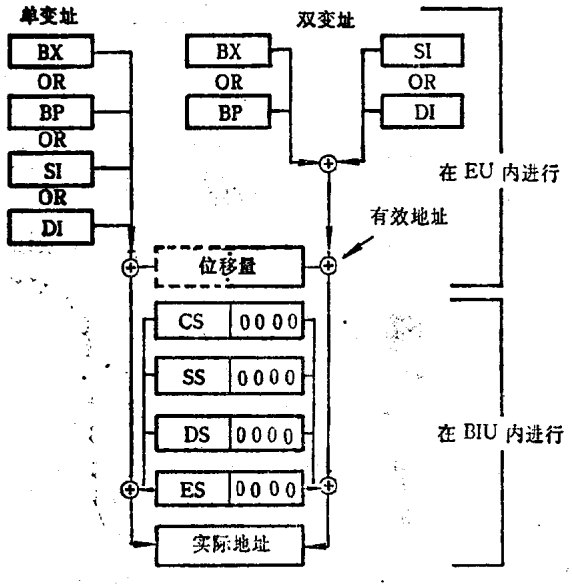


图 2.28 存贮器地址的计算

(二) 存贮器寻址方式

操作数偏移地址是在 8086 的 EU (执行部件，在 §2.8 中介绍) 中计算出来的，它是个不带符号的 16 位数，也就是所谓的有效地址 (EA)，它表示距其所在段起点的字节距离。图 2.28 表示了执行部件 (EU) 把位移量、基址寄存器的内容和变址寄存器的内容相加的方法。我们已经知道这 3 个分量的任一组合，可在给定的指令中，产生 8086 的各种存贮器寻址方式。尽管位移量是一个包含在指令中的 8 位或 16 位常数，但基址寄存器和变址寄存器却可以在执行中改变，从而使一条指令能访问不同的单元。表 2.6 列出了计算位移量、基址寄存器和变址寄存器的各种组合，形成有效地址所需要的时间 (以时钟周期为计量单位)。

1. 直接寻址

直接寻址（见图 2.29）是最简单的存贮器寻址方式，它不涉及寄存器，有效地址直接取自指令的偏移量字节。

2. 寄存器间接寻址

存贮器操作数的有效地址，可直接取自基址寄存器或变址寄存器之一（见图 2.30）。只要对基址寄存器或变址寄存器的值作适当的修改，一条指令就可以对许多不同的存贮单元进行操作，值得注意的是，在用 JMP 或 CALL 指令时，任何 16 位寄存器，都可以用作间接寻址的寄存器。

表 2.6 有效地址(EA)的计算时间

有效地址的构成成分	时钟周期*
只有位移量	6
只有基址或变址 (BX, BP, SI, DI)	5
位移量＋基址或变址 (BX, BP, SI, DI)	9
基址＋变址 BP＋DI, BX＋SI	7
BP＋SI, BX＋DI	8
位移量＋基址＋变址 BP＋DI＋disp	11
BX＋SI＋disp	
BP＋SI＋disp	
BX＋DI＋disp	12

* 段超越要增加 2 个时钟周期

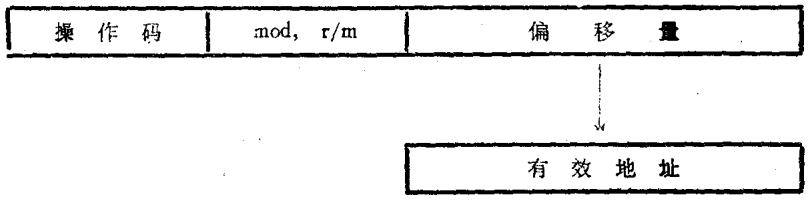


图 2.29 直接寻址

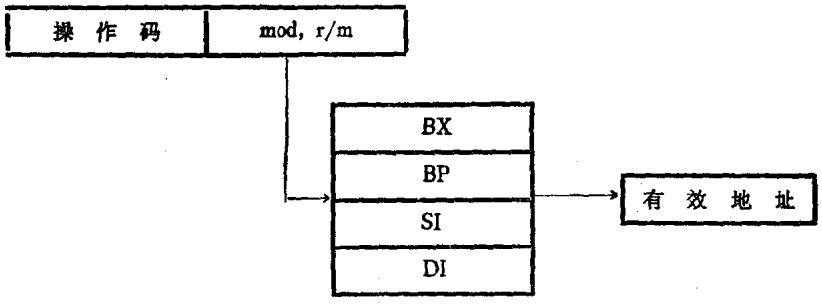


图 2.30 寄存器间接寻址

3. 基址寻址

在基址寻址中（见图 2.31），有效地址是位移量值和寄存器 BX 或寄存器 BP 的内容之

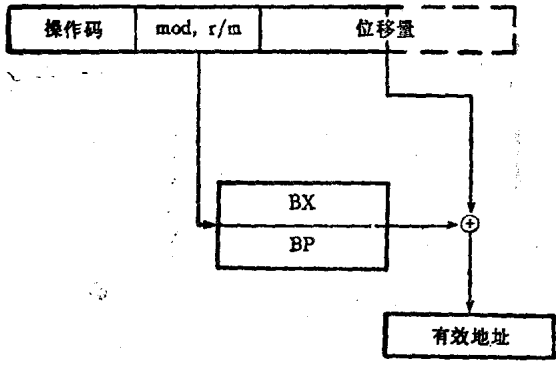


图 2.31 基址寻址

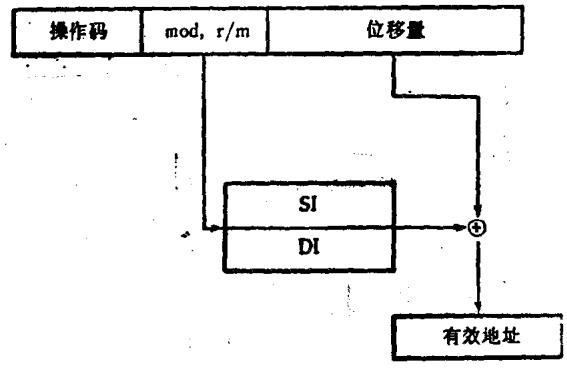


图 2.32 变址寻址

和，必须要记住的是，若 BP 作为基址寄存器，则是从当前堆栈段获得操作数（除非有段超越前缀出现）。

4. 变址寻址

在变址寻址中，有效地址是位移量加变址寄存器 (SI 或 DI) 的内容（见图 2.32）。变址寻址通常用于访问数组中的元素（见图 2.33）位移量定位于数组的起点，变址寄存器的值选择一个元素。因为数组中所有的元素具有相同的长度，所以对变址寄存器进行简单的运算，就可以选择数组中的任何元素。

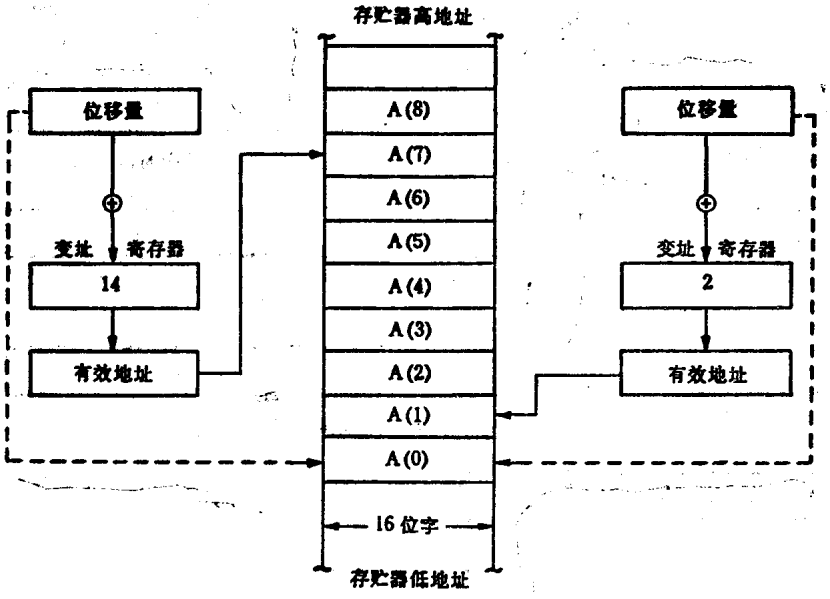


图 2.33 用变址寄存器访问数组

5. 基址变址寻址

基址变址寻址产生的有效地址是基址寄存器、变址寄存器和位移量之和（见图 2.34）。由于在执行过程中可以改变两个地址构成分量，所以这种寻址方式是非常灵活的。

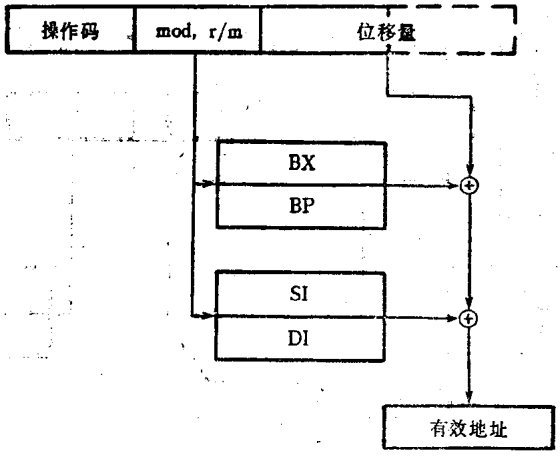


图 2.34 基址变址寻址

基址变址寻址为访问堆栈中的数组的过程提供了方便（见图2.35），寄存器BP可以存放堆栈中某一参考点的偏移量，通常就是该过程在保存了寄存器的内容，并分配了本地存贮器之后的堆栈顶。从参考点到数组起点的距离，可用位移量来表示。变址寄存器可用于访问数组的各元素。

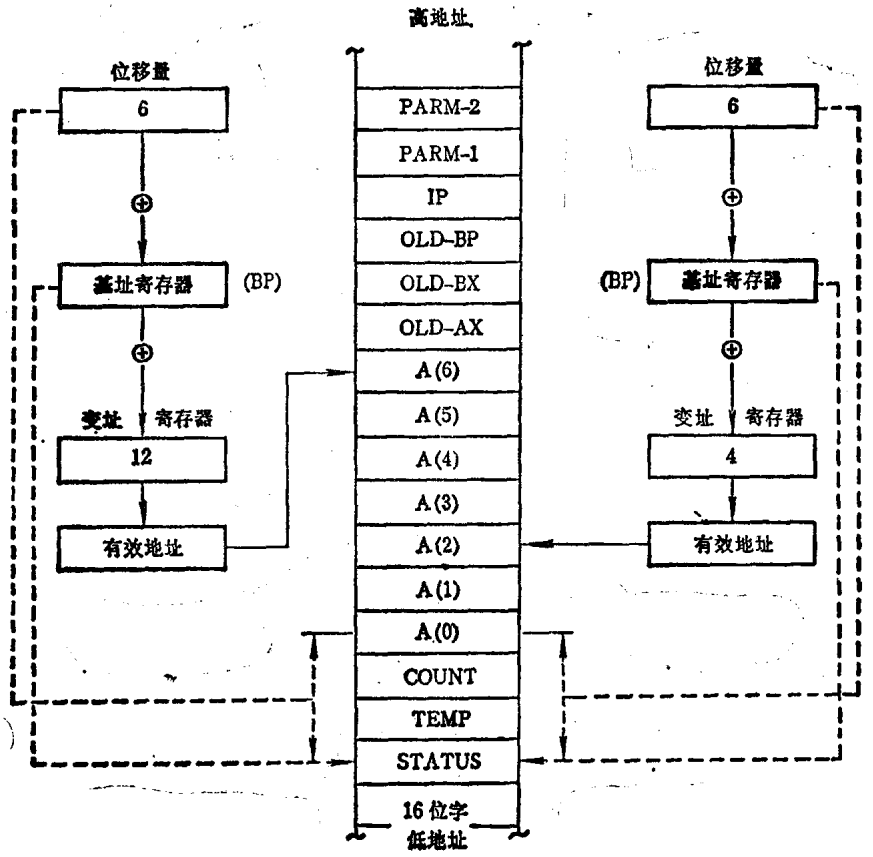


图 2.35　用基址变址寻址访问堆栈数组

6. 串寻址

串指令访问的操作数，不能用通常的存贮器寻址方式，而是隐含地运用变址寄存器（见图2.36）。当执行串指令时，设定SI指向源串的第一个字节或字，DI指向目的串的第一个字节或字。在重复串操作中，CPU自动地调整SI和DI，以获得后续的字节或字。

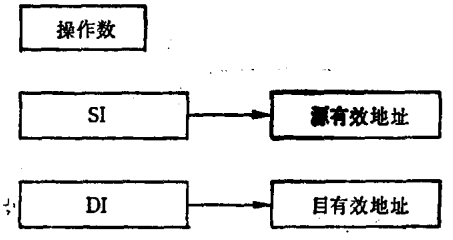


图 2.36　串操作数寻址

7. 输入/输出转接口寻址

如果输入/输出转接口是存贮器映像，则任何存贮器操作数寻址方式，都可用来访问输入/输出转接口。例如，一组终端可作为一个“数组”来访问。串指令还可用于把数据传送给具有相应硬件接口的存贮器映象输入/输出转接口。

在使用输入/输出指令的情况下，有两种不同的寻址方式可用来访问位于输入/输出空间的转接口（见图2.37）。在直接转接口寻址方式中，转接口的号码是8位的立即数，可以固定

访问 0～255 号转接口。间接转接口寻址方式，类似于存贮器操作数的寄存器间接寻址。转接口的编号取自寄存器 DX，其范围为 0—65,535。利用一个调整 DX 值的简单的软件循环，就可以访问一组相邻的转接口。图 2.38 表示了出现在指令中的 reg, seg, mod, r/m, w, s, d 字段的含义。它可以很快地帮助我们决定指令的寻址方式以及操作数的偏移地址（有效地址 EA）由哪几个量组成。下面我们举几个例子来说明如何使用这张图。

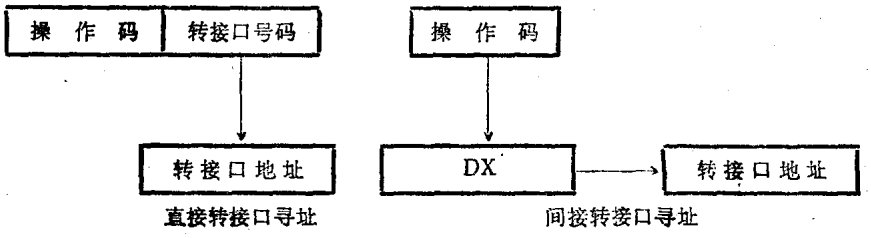
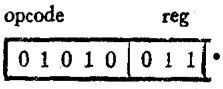
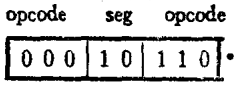


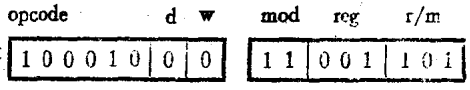
图 2.37 输入/输出通道寻址

例 1：已知有一条单操作数指令 PUSH，它的格式是 

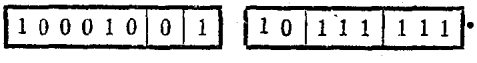
。很明显，单操作数指令的一字节形式是字操作，并且是对寄存器进行操作的，从图中找到单操作数一字节形式，循着 reg 字段引出的箭头，在寄存器编码表中的 w＝1 项下，找到 011 是 BX 寄存器，于是这条指令的功能是把 BX 寄存器的内容压入堆栈。

例 2：已知一条单操作数指令 PUSH，它的格式是 

。这是一条把某一个段寄存器推入堆栈的操作，从段寄存器编码表中可以查到 seg 字段中的 10 是 ss 段寄存器，所以这条指令的功能是把段寄存器 ss 的内容压入堆栈。

例 3：已知一条双操作数指令 MOV 的格式是 

。由于 d 字段是 0，从 d 字段的编码中，可以知道目操作数是由 mod，r/m 字段指定的那个操作数。从 w 字段是 0，可以知道该传送操作是对字节进行的。从图中双操作数两字节形式，可以看到由于 mod 字段等于 11，它的 mod，r/m 字段所代表的操作数寻址应该循着寄存器的编码表进行，在表中 w＝0 项下，可以查到 r/m＝101 是代表 CH 寄存器，同样在表中的 w＝0 项下，可以查到 reg＝001 是 CL 寄存器。于是这条指令的功能就是把源操作数 CL 的内容传送到目操作数 CH 中去。

例 4：已知一条 MOV 指令的格式为 

。从图 2.38 的双操作数指令的形式中，可以看到 d＝0 表示目操作数为 mod，r/m 字段指定的操作数。由于 w＝1，它是一条字操作指令。由于 mod≠11，所以它应该循着 mod≠11 的箭头到 mod r/m 字段的组合编码表中查找，由 mod＝10 可知该条指令具有两个附加字节，它们的内容就是构成该操作数偏移地址的位移量，循着箭头向右查找 r/m＝111，应该是 BX＋无，也就是 BX。这样，该操作数的偏移地址应该是 2 字节的位移量加上 BX 的内容。源操作数循着 w＝1 和 reg＝111 在寄存器编码表的 w＝1 项下查到是 DI。于是这条指令的功能，就是把 DI 的内容，传送到偏移地址由附在指令后的两字节位移量与 BX 的内容之和所决定的字单元中去。

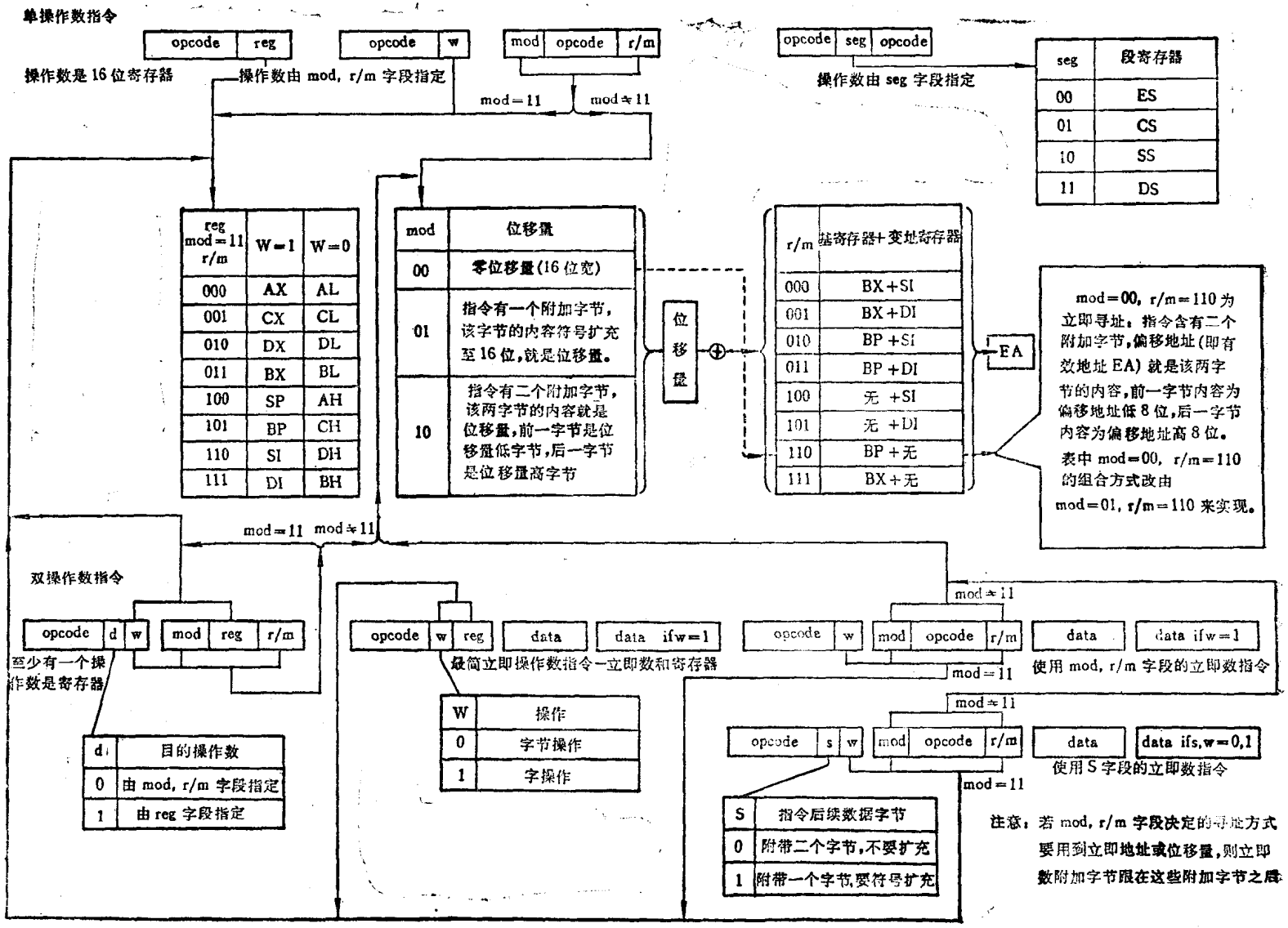
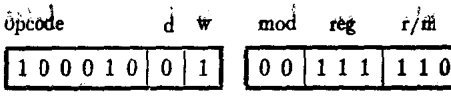
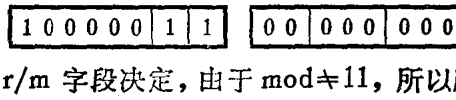


图 2.38 8086 指令操作数寻址一览图

例5：已知一条MOV指令的格式为 

。这条指令和例4的那条指令除了mod、r/m字段不同外，其余都相同。由于d=0，mod、r/m字段所指定的操作数是目操作数，循着mod≠11的箭头到mod，r/m字段的组合编码表中查找，从表中可知mod=00，r/m=110是一种立即寻址方式，即目操作数的偏移地址在指令的下两个字节中。于是这条指令就实现把DI的内容送到偏移地址由该两字节指出的字单元中去的功能。

例6：已知一条ADD指令的格式为 

。这是一条立即操作数指令，它的目的操作数由mod，r/m字段决定，由于mod≠11，所以应在mod，r/m编码表中查找，mod=00，r/m=000表示偏移量是0位移量+BX+SI，也就是BX+SI之和所决定。由于w=1并且s=1，所以该指令后面带有一个字节的立即操作数，在操作时它要被符号扩展为16位，然后和目操作数相加，结果放在偏移量由BX+SI决定的字单元中。在上述例子中，若指令的mod字段等于10时，则偏移量将由在两个附加字节中的位移量+BX+SI来决定。这时指令附加的立即数字节，在两个位移量附加字节的后面。

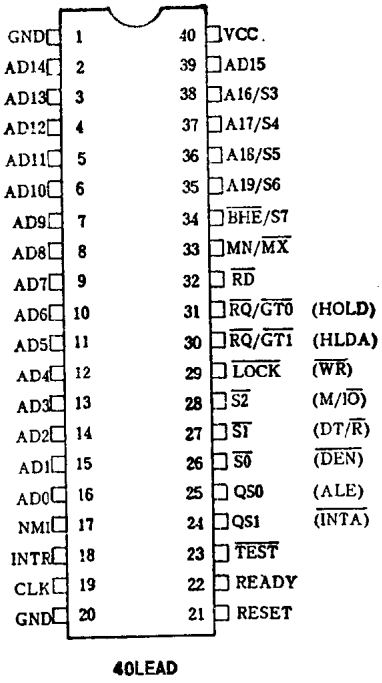
以上这些例子仅说明了各种字段的用途和它们之间的一般关系，读者可以自己找些例子，按图索骥，举一反三，掌握8086的寻址方式就不是一件难事了。

§2.8 8086微处理器体系结构综述

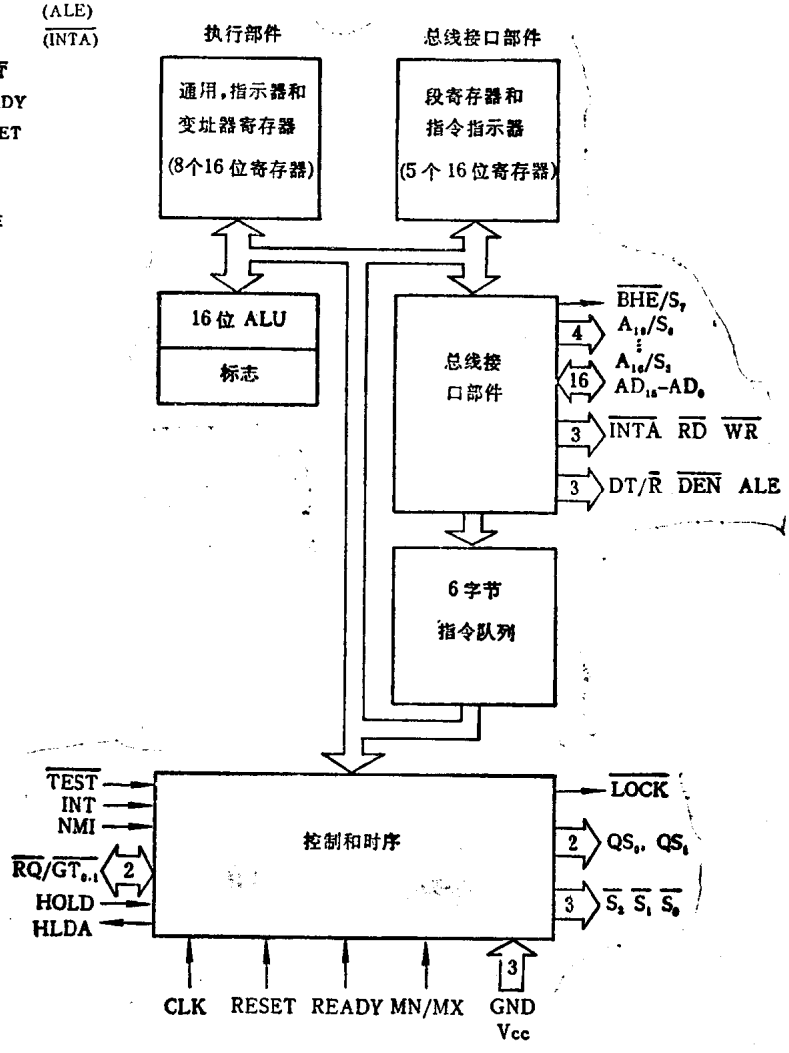
8086是第三代微处理器，是一片强功能的16位CPU。它采用高性能的N-沟道，耗尽型负载的硅栅工艺(HMOS)制造，封装在标准的40引脚的双列直插式管壳内。CPU的时钟频率有三种：8086为5兆赫；8086-2为8兆赫；8086-1则为10兆赫。8086既可用在单处理机系统中，也可用在多微处理机系统中。8086的直接寻址空间为1兆字节。片内有硬件乘除指令和串处理指令，可以对位，字节，字，字节串，字串，压缩的和非压缩的BCD码(10进制)等多种数据类型进行处理。8086具有24种寻址方式，一集包括对汇编语言和高级语言提供有力支持的指令系统。8086与8080/8085具有向上兼容性，可以使用早期为8位系统研制的各种外围支持电路和技术，是当前世界上应用最广泛的16位微处理器。图2.39(a)表示了8086微处理器的引脚安排，图2.39(b)则表示了8086内部功能块之间的相互联系。

一. 8086总线周期

从图2.39中可以看到，8086与存贮器和外部通讯是通过20位分时多路地址和状态数据总线来实现的。为了传送数据或取出指令，CPU要执行一个总线周期(见图2.40)，最小的总线周期由4个叫做T状态的时钟周期(时钟周期是计算机计量时间的一种基本单位，若8086的主频为5MHz，则每个时钟周期为200ns)组成。在第1个T状态(T_1)，CPU在20位多路地址/数据/状态总线上发出地址。在第2个T状态(T_2)，CPU从总线上撤消地址，或使总线低16位输出浮置成3态，以便为读进或写出数据作准备。在T_2期间，多路总线的高4位地址(A_{19}—A_{16})，输出总线周期状态(S_6、S_5、S_4、S_3)这些状态信息主要用于诊断监视。在T_3期间，CPU在总线高4位上继续提供状态信息，在低16位上，或者继续发出写数据，或者采样读入数据。如果选定的存贮器或I/O设备不能以CPU最大传送速率传送数



(a) 8086 的引脚安排



(b) 8086 CPU 功能块

图 2.39 8086 CPU 的引脚和功能块

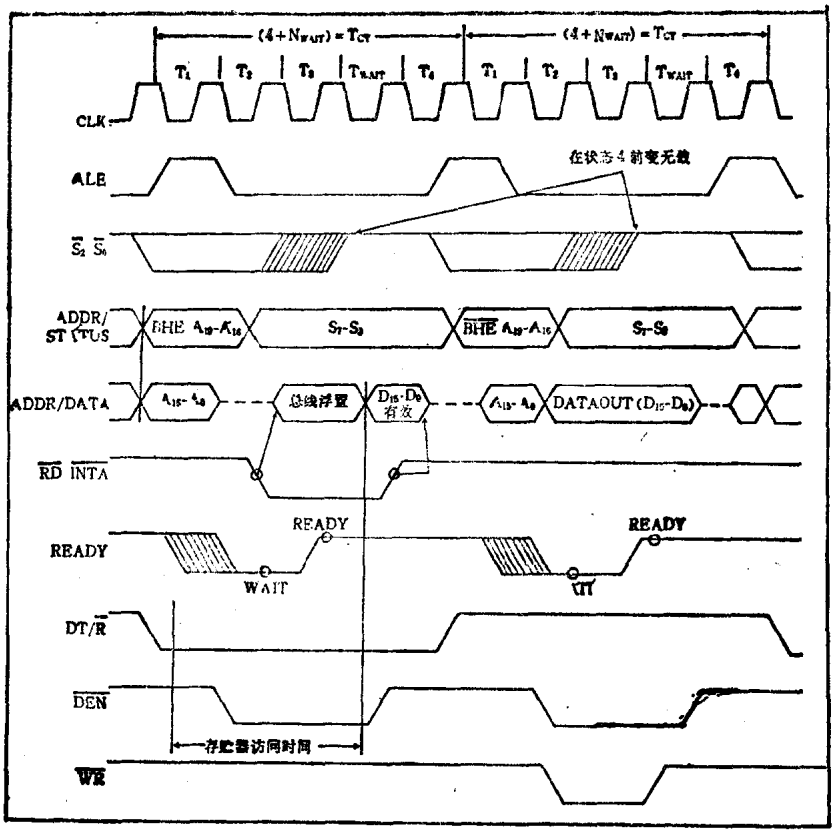


图 2.40 基本 8086 总线周期

据，该设备必须通知 CPU "未准备好"，并迫使 CPU 在 T_3 之后引入附加的时钟周期（等待状态 T_W）。"未准备好" 信号必须在 T_3 启动前送到 CPU。T_W 期间的总线活动与 T_3 期间一样。当选定的设备有足够时间完成传送时，它就发出"准备好"(READY) 信号，并使 CPU 脱离 T_W 状态继续工作。在最后一个等待状态期间，或者当不要求等待状态时，在 T_3 期间，CPU 锁存总线上的数据。总线周期在 T_4 结束（命令线禁止，选定的外部设备从总线脱开）。8086 的总线周期在系统设备中是异步的，它包括选择设备的地址，随后的读选通或是数据的写选通。选定设备在写周期期间接收总线数据，在读周期期间把所需数据送达总线。在命令结束时，该设备锁存写数据或使其总线驱动器禁止。在总线周期期间，设备发出的唯一控制就是使处理器插入等待周期。

当必须对存贮器或 I/O 设备传送（送入或送出）指令操作数时，CPU 才执行总线周期。不执行总线周期时，总线接口执行空闲周期 (Ti)。在空闲周期期间，CPU 在高位地址总线上继续驱动来自前一个总线周期的状态信息。如果前一个总线周期是写，则 CPU 继续在多路复用总线上驱动数据，直到下一个总线周期开始，如果 CPU 执行空闲周期是在读周期之后，则 CPU 在请求下一个总线周期之前不驱动低 16 位总线。

二、8086 引脚功能简介

1. AD_{15}～AD_0　Address Data Bus(地址数据总线)

第 2～16，39 脚　输入，输出双向，三态。

这些引脚是分时多用的，T_1 时输出存贮器或 I/O 的地址；在 T_2、T_3、T_W 和 T_4 时则为数据的收发引脚，对低 8 位数据 D_7～D_0 来说 A_0 与 $\overline{BHE}$ 的作用相同，在 T_1 时 AD_0 为低电平，表示收发存贮器或 I/O 的低 8 位字节。对汇接于低 8 位的那些 8 位型设备来说，可用 A_0 和 $\overline{BHE}$ 作为片选，这些引脚在中断响应和局部总线"保持响应"时期，被浮置成高阻抗状态。

2. A_{19}/S_6～A_{16}/S_3　Address/Status (地址/状态)

第 35～38 脚　输出，三态。

在 T_1 时期这 4 条引脚对存贮器访问来说是最高 4 位地址线，但 I/O 访问时期它们将保持低电平。在存贮器或 I/O 操作期间且 CPU 处于 T_2、T_3、T_W 和 T_4 状态时，则输出 8086 的状态信息。S_6 表示允许中断标志位，它在每个时钟周期的开始时修改。S_4 和 S_3 的编码如表 2.7 所示。这些信息指出当前数据访问所用的段寄存器的情况。当局部总线处于"保持响应"时，这些引脚被浮置，而处于高阻抗状态。

表 2.7　S_4 和 S_3 的编码

S_4	S_3	特　　性
0	0	交替数据(对应附加段 ES)
0	1	堆栈(对应堆栈段 SS)
1	0	代码或不用(对应代码段 CS)
1	1	数据(对应数据段 DS)
S_6=0		指示 8086 在总线上

3. $\overline{BHE}/S_7$　Bus High Enable/Status(高 8 位总线允许/状态)

第 34 脚　输出，三态。

T_1 时，高 8 位总线可用信号 $\overline{BHE}$ 来表示位于 D_{15}～D_8 的高 8 位数据总线上的信息可以使用，对汇接于高 8 位总线的那些 8 位型设备来说，通常用 $\overline{BHE}$ 作为片选信号。对于读、写和中断响应周期，在 T_1 时为了把一个字节传送到高 8 位总线上去，$\overline{BHE}$ 应为低电平。在 T_2、T_3、T_W 和 T_4 时期，这条引脚用来输出 S_7 的信息，S_7 是低电平有效。在"保持响应"时被浮置而处于高阻抗状态。在中断响应的第一个周期的 T_1 时，$\overline{BHE}$ 必为低电平。$\overline{BHE}$ 和 A_0 的编码如表 2.8 所示。

表 2.8　$\overline{BHE}$ 和 A_0 的编码

$\overline{BHE}$	A_0	特　　性
0	0	全字(16 位)
0	1	在数据总线高 8 位进行字节传送(D_{15-8})
1	0	在数据总线低 8 位进行字节传送(D_{7-0})
1	1	保留

4. $\overline{RD}$　　Read (读)

第 32 脚　　输出，三态。

读选通，用来读出 8086 局部总线上设备中的信息，处理器是对存贮器还是对I/O进行读操作，由 M/$\overline{IO}$ 脚的状态决定。在任何读周期的T_2、T_3 和 T_W 时期 $\overline{RD}$ 为低电平时才有效，然后保持高电平直到另一次读或 8086 局部总线被浮置为止。在"保持响应"时期，$\overline{RD}$ 被浮置而处于高阻抗状态。8086 的读时序如图 2.41 所示。

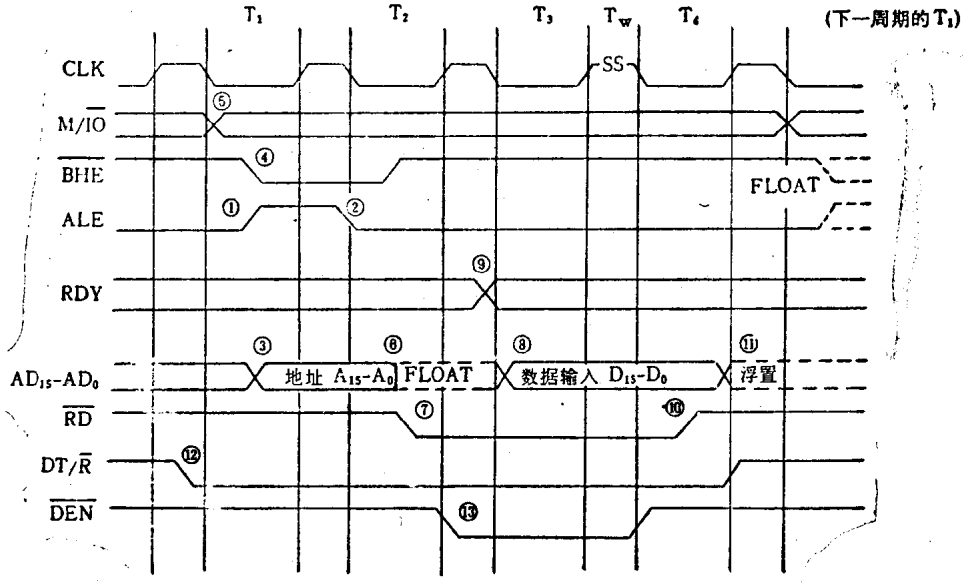


图 2.41　读周期波形

处理器在 T_1 期间开始读周期，保持允许地址锁存信号 ALE①，ALE 的后沿②用来把地址信息（这时③地址信息处于局部总线上）锁存于 8282 或 8283 锁存器内。$\overline{BHE}$ ④和 A_0 信号决定寻址低字节、高字节或整个字。在 T_1～T_4 期间，M/$\overline{IO}$ 信号⑤选择存贮器或输入/输出 (I/O) 设备。在 T_2 时，地址已从局部总线移出，并使处理器的总线驱动器处于高阻抗状态⑥，在 T_2 期间发出的控制信号 ($\overline{RD}$)⑦一直维持着。$\overline{RD}$ 使被寻址设备能把它的总线驱动器加到现在已释放的局部总线上，在稍迟一些时候，总线上有效数据就可以付之使用⑧，然后被寻址器件(设备) 把 READY 线驱动到高电平⑨。当 8086 下一次使 $\overline{RD}$ 返回到高电平时⑩，所寻址器件 (设备) 的总线驱动器将处于三态，再次让出 (放弃) 总线⑪，如果 8286 或 8287 收发器用于缓冲局部总线，则由 8086 的 DT/$\overline{R}$ ⑫和 $\overline{DEN}$ ⑬信号来服务。

5. READY　　Ready (准备好)

第 22 脚　　输入。

这是被访问的存贮器或 I/O 设备发来的回答信号，表示数据传送已准备就绪。来自存贮器或 I/O 设备的准备好信号，被 8284A 时钟发生器同步后，才形成 Ready 信号，它是高电平有效，但到达 8086 的 Ready 输入并不是同步的，如果建立和保持的时间没有满足，就不能保证正确的操作，即需要加 T_W。

6. INTR　Interrupt Request (中断请求)

第 18 脚　输入。

这是一个电平触发输入，在每条指令的最后一个时钟周期时对它进行采样，以决定处理器是否要进入中断响应操作。在系统存贮器中的中断向量表中，可以获得一个中断向量，用来调用中断子程序。可以用软件的办法清除中断允许标志位来进行中断的内部屏蔽。INTR 由内部进行同步。该信号高电平有效。

7. $\overline{\text{TEST}}$　Test (测试)

第 23 脚　输入。

在“Wait”(等待)指令时期，$\overline{\text{TEST}}$ 对它进行监视，如果 $\overline{\text{TEST}}$ 输入为低电平，表示继续执行该等待测试指令；否则，处理器处于空闲的等待状态。这个输入信号在每个时钟周期内，由时钟脉冲的前沿来进行内部同步。

8. NMI　Non-Maskable Interrupt (非屏蔽中断)

第 17 脚　输入

这是一个边沿触发的输入信号，它引起一个类型 2 中断。通过查找系统存贮器内的中断向量表，可以找到对应的中断子程序。NMI 是不能用软件进行屏蔽的，一个从低到高变化的电平边沿，就可以使处理器在现行指令结束后马上进行中断响应，这个输入是片内进行同步的。

9. RESET　Reset (复位)

第 2 脚　输入。

这个信号使处理器马上结束现行操作，它必须保持 4 个时钟周期以上的高电平。随着 RESET 变回低电平，处理器就开始执行再启动过程。RESET 由片内进行同步。

10. CLK　Clock (时钟)

第 19 脚　输入。

为处理器和总线控制提供基本的定时脉冲。它是非对称的，具有 33% 的有效高电平期，从而提供最佳的内部定时。

11. Vcc

第 40 脚

+5 伏电源引脚。

12. GND　Ground (地)

第 1,20 脚

接地引脚。

13. MN/$\overline{\text{MX}}$　Minimum/Maximum Mode Control (最小/最大模式控制)

第 33 脚　输入。

决定处理器是接成最小模式，还是最大模式结构。这两种模式将分别在下面讨论。

8086 在最小模式 (MN/$\overline{\text{MX}}$ = Vcc) 工作时有关引脚功能的描述 (其他引脚功能在前面已

有说明)。

14. M/$\overline{IO}$ Memory/$\overline{IO}$ (存贮器/IO 控制)

第 28 脚 输出,三态。

在最大模式中称为 S_2,这里用它来区分是对存贮器访问,还是对I/O 进行访问。M/$\overline{IO}$ 在前一个总线周期的 T_4 变得有效,从而开始了一个新的总线周期,它一直保持其电平到本周期 T_4 的结束。M/$\overline{IO}$ 为高电平表示访问存贮器,为低电平表示访问 I/O。**在局部总线"保持响应"期间,被浮置成高阻抗状态。**

15. $\overline{WR}$ Write (写)

第 29 脚, 输出,三态。

表示处理器执行存贮器写或 I/O 写的操作,执行哪一种操作则由 M/$\overline{IO}$ 决定。对任何写周期,$\overline{WR}$ 均只在 T_2、T_3 和 T_W 期间有效。$\overline{WR}$ 为低电平有效。在局部总线"保持响应"时被浮置而处于高阻抗状态。

8086 写周期的波形见图 2.42。具体过程写周期和读周期一样,开始就保持 ALE 信号① 并发出一个地址②。再根据 M/$\overline{IO}$ 信号③进行存贮器或 I/O 写操作。在 T_2 时,该地址立即跟着发出去,以便把处理器发出的数据写入到被寻址的单元④,在总线上这个数据保持有效直到T_4 的中间⑤。在 T_2 开始处写信号 $\overline{WR}$⑥ (应比 $\overline{RD}$ 早些出现)处于低电平,并在整个 T_2、T_3 和 T_W 期间有效。

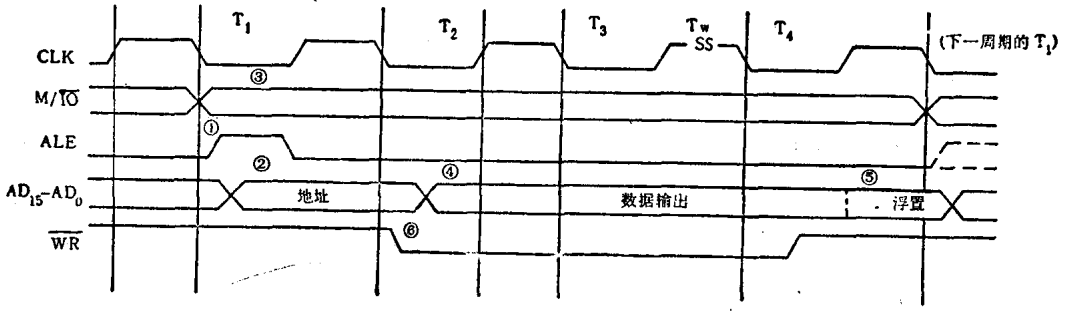


图 2.42 写周期波形

16. $\overline{INTA}$ Interrupt Acknowledge (中断响应)

第 24 脚 输出。

在中断响应周期中,用作读选通信号。在每个中断响应周期的 T_2、T_3 和 T_W 期间,它变成有效的低电平。

17. ALE Address Latch Enable (地址锁存允许)

第 25 脚 输出。

由处理器提供的允许地址锁存于 8282/8283 锁存器的控制信号。它是一个正脉冲,位于任何一个总线周期的 T_1 期间。ALE 不能被浮置。

18. DT/$\overline{R}$　Data Transmit/Receive (数据收发)

第 27 脚　输出,三态。

在最小模式系统中，有时要用 8286/8287 数据总线收发器，这时就用 DT/$\overline{R}$ 来控制数据收发器的传送方向。DT/$\overline{R}$与最大模式的 $\overline{S_1}$是同一条引脚。其定时则与 M/$\overline{IO}$ 相同。当局部总线“保持响应”时，DT/$\overline{R}$ 被浮置成高阻抗状态。

19. $\overline{DEN}$　Data Enable (数据允许)

第 26 脚　输出,三态。

在最小模式系统中,如果用 8286/8287 作为数据总线收发器时，$\overline{DEN}$ 为其提供一个表示输出数据可用的信号,在每个存贮器访问或 I/O 访问周期里，$\overline{DEN}$ 变成有效的低电平，在中断响应周期内,也变成低电平。在读或中断响应周期时，$\overline{DEN}$ 在 T_2 的中间开始有效,并一直保持到 T_4 的中部。对于写周期，从 T_2 之初就开始直到 T_4 的中部这段期间保持有效。$\overline{DEN}$ 在局部总线“保持响应”期间被浮置而处于高阻抗状态。

20. HOLD, HLDA　Hold Request (保持请求), Hold Acknowledge (保持响应)

第 31, 32 脚　输入,输出。

前者是另一个局部总线的主设备,要求 8086 保持并让出总线的请求信号,后者是 8086 同意让出总线向该主设备发出的保持响应信号。HOLD 必须是高电平才有效，处理器收到这个请求信号后,如果同意,应在时钟周期 T_4 中间或 T_1 时期发出 HLDA，与此同时，处理器浮置局部总线和各控制线为三态。当 HOLD 变成低电平时,处理器也把 HLDA 变低,此时处理器即可再度获得总线和控制线的主控权。

要处理器放弃局部总线也应该满足 $\overline{RQ}/\overline{GT}$ 中所述的 4 个规则。

HOLD 不是一个异步输入，如果系统不能决定其定时,则应采取外同步措施。HOLD/HLDA 的定时图在图 2.43 中表示。

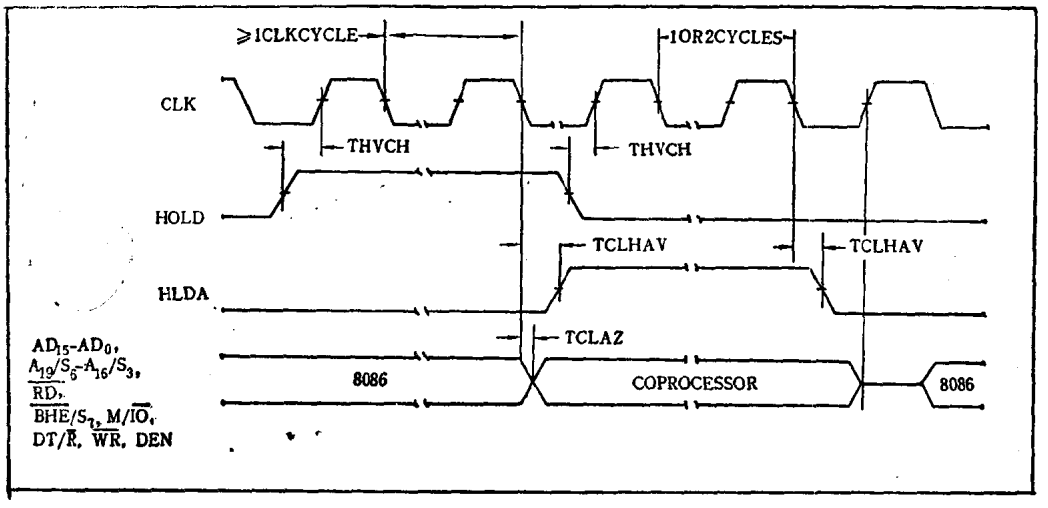
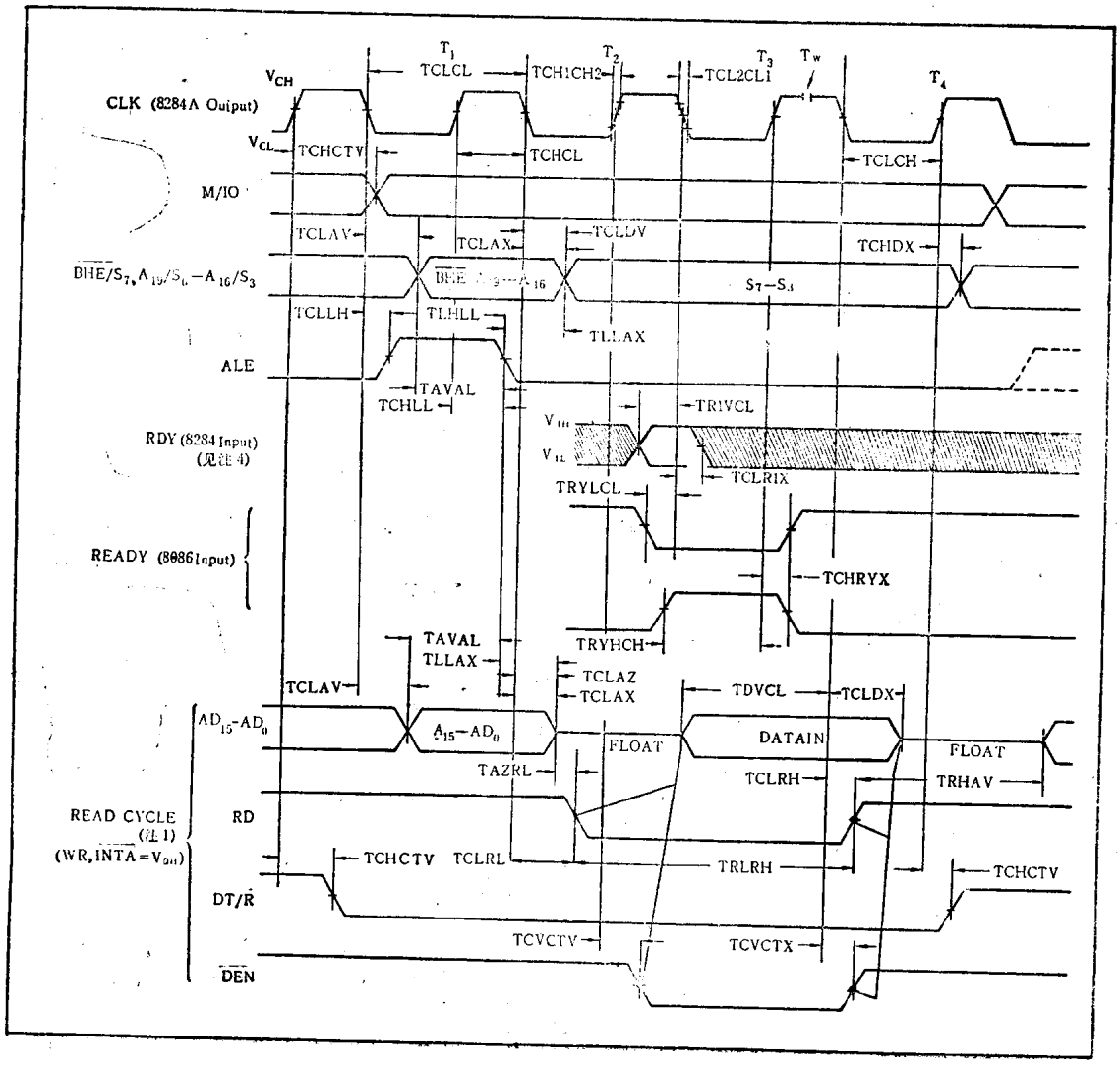
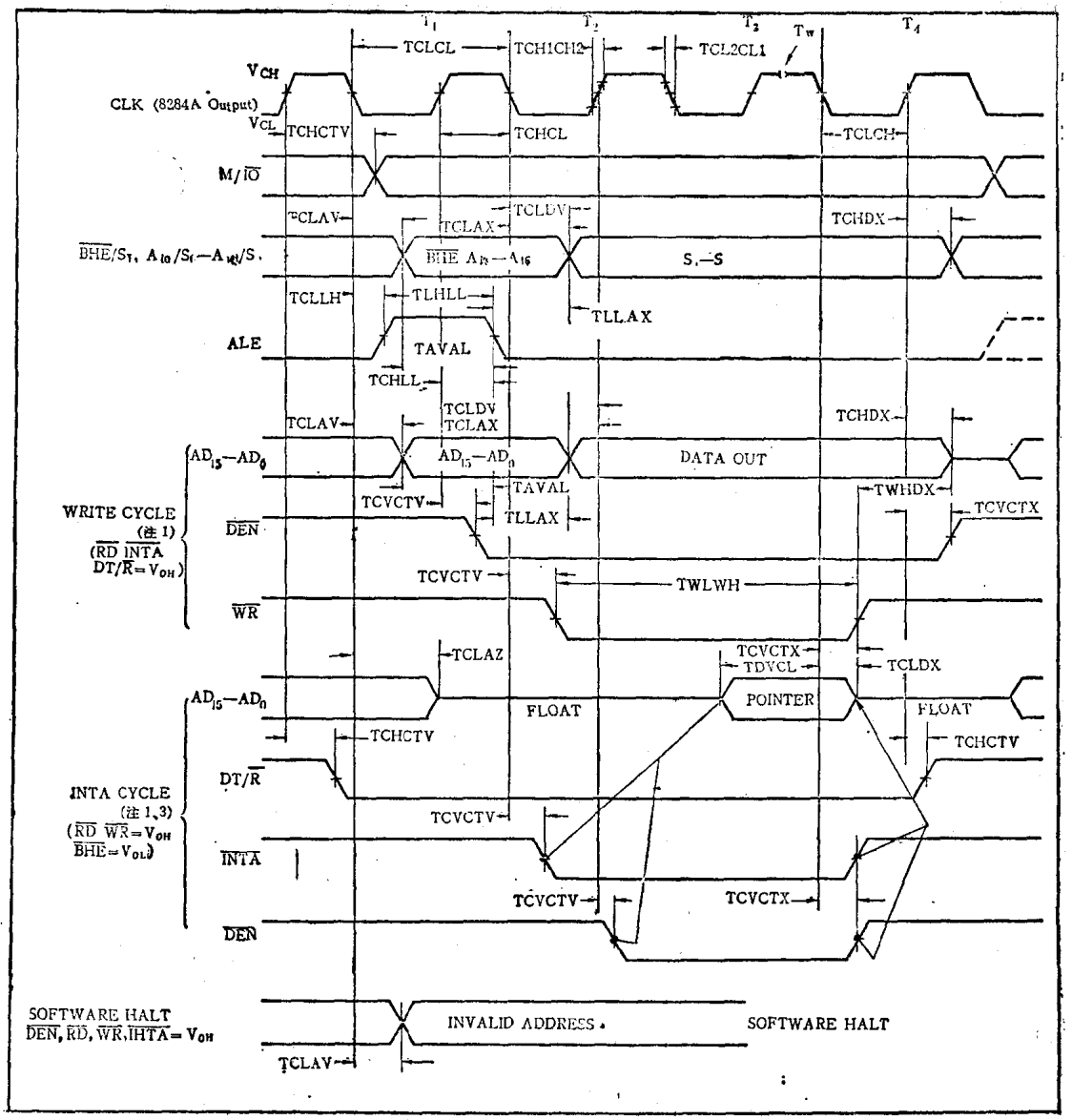


图 2.43　HOLD/HLDA 定时图

图 2.44 是 8086 最小系统的定时图,从中可以看到在最小模式下,各引脚信号在时序上的相互关系,


T1
T2
T3
Tw
T4
TCLCL
TCH1CH2
TCL2CL1
VCH
VCL
CLK (8284A Output)
TCHCTV
TCHCL
TCLCH
M/IO
TCLAV
TCLAX
TCLDV
TCHDX
BHE/S7, A19/S6—A16/S3
S7—S3
TCLLH
TLHLL
TLLAX
ALE
TAVAL
TCHLL
VIH
VIL
TRIVCL
RDY (8284 Input)
(见注4)
TCLRIX
TRYLCL
READY (8086 Input)
TCHRYX
TRYHCH
TAVAL
TLLAX
TCLAV
TCLAZ
TCLAX
TDVCL
TCLDX
AD15-AD0
A15—AD0
FLOAT
DATAIN
FLOAT
TAZRL
TCLRH
TRHAV
READ CYCLE
(注1)
(WR, INTA=VOH)
RD
TCHCTV
TCLRL
TRLRH
TCHCTV
DT/R
TCVCTV
TCVCTX
DEN



注：1. 非经特别指出，所有的信号都在 V_{CH} 和 V_{CL} 之间转换

2. 在 T_1、T_3 和 T_W 的末尾进行 RDY 采样，以决定是否要插入 T_W 状态.

3. 两个 INTA 周期背靠背执行. 8086 局部总线在两个 INTA 周期中浮置. 控制信号只表示了第 2 个 INTA 周期.

4. 8284 的信号仅供参考.

5. 非经特别指出，所有时序的测定是在 1.5V 下进行的.

图 2.44　8086 最小模式系统定时图

表 2.9 是 8086 最小复杂系统的请求和响应定时 (参见图 2.43 和图 2.44)。

表 2.9 8086 最小复杂系统的请求和响应定时

(a) 请求定时

Symbol	Parameter	8086/8086-4		8086-2 (Prellminary)		Units	Test Conditions
		Min.	Max.	Min.	Max.		
TCLCL	CLK Cycle Period — 8086 — 8086-4	200 250	500 500	125	500	ns	
TCLCH	CLK Low Time	(2/3 TCLCL) −15		(2/3 TCLCL) −15		ns	
TCHCL	CLK High Time	(1/3 TCLCL) +2		(1/3 TCLCL) + 2		ns	
TCH1CH2	CLK Rise Time		10		10	ns	From 1.0V to 3.5V
TCL2CL1	CLK Fall Time		10		10	ns	From 3.5V to 1.0V
TDVCL	Data In Setup Time	30		20		ns	
TCLDX	Data In Hold Time	10		10		ns	
TR1VCL	RDY Setup Time into 8284 (见注 1,2)	35		35		ns	
TCLR1X	RDY Hold Time into 8284 (见注 1,2)	0		0		ns	
TRYHCH	READY Setup Time into 8086	(2/3 TCLCL) −15		(2/3 TCLCL) −15		ns	
TCHRYX	READY Hold Time into 8086	30		20		ns	
TRYLCL	READY Inactive to CLK (见注 3)	−8		−8		ns	
THVCH	HOLD Setup Time	35		20		ns	
TINVCH	INTR, NMI, TEST Setup Time (见注 2)	30		15		ns	

(b) 响应定时

Symbol	Parameter	8086/8086-4		8086-2 (Preliminary)		Units	Test Conditions
		Min.	Max.	Min.	Max.		
TCLAV	Address Valid Delay	10	110	10	60	ns	C_L=20-100pF for all 8086 Outputs (in addition to 8086 self-load)
TCLAX	Address Hold Time	10		10		ns	
TCLAZ	Address Float Delay	TCLAX	80	TCLAX	50	ns	
TLHLL	ALE Width	TCLCH-20		TCLCH-10		ns	
TCLLH	ALE Active Delay		80		50	ns	
TCHLL	ALE Inactive Delay		85		55	ns	
TLLAX	Address Hold Time to ALE Inactive	TCHCL-10		TCHCL-10		ns	
TCLDV	Data Valid Delay	10	110	10	60	ns	
TCHDX	Data Hold Time	10		10		ns	
TWHDX	Data Hold Time After WR	TCLCH-30		TCLCH-30		ns	
TCVCTV	Control Active Delay 1	10	110	10	70	ns	
TCHCTV	Control Active Delay 2	10	110	10	60	ns	
TCVCTX	Control Inactive Delay	10	110	10	70	ns	
TAZRL	Address Float to READ Active	0		0		ns	
TCLRL	RD Active Delay	10	165	10	100	ns	
TCLRH	RD Inactive Delay	10	150	10	80	ns	
TRHAV	RD Inactive to Next Address Active	TCLCL-45		TCLCL-40		ns	
TCLHAV	HLDA Valid Delay	10	160	10	100	ns	
TRLRH	RD Width	2TCLCL-75		2TCLCL-50		ns	
TWLWH	WR Width	2TCLCL-60		2TCLCL-40		ns	
TAVAL	Address Valid to ALE Low	TCLCH-60		TCLCH-40		ns	

注：1. 8284 的信号仅供参考.

2. 关于异步信号的建立请求,仅在下一时钟周期确保识别.

3. 仅适用于 T_2 状态.

8086 处于最大模式工作时 (MN/$\overline{MX}$=地)，有关引脚的功能描述 (其他引脚的功能均如前述)：

21. $\overline{S_2}$、$\overline{S_1}$、$\overline{S_0}$　Bus Cycle Status (总线周期状态)

第 26，27，28 脚　输出。

这些处理器状态输出引脚，在 T_4、T_1 和 T_2 期间输出有效，而在 T_3 或 T_W 期间且当 READY 为高电平时，则返回被动状态 (1，1，1)。这些信号被总线控制器 8288 用来产生存贮器和 I/O 控制信号。在 T_4 时，$\overline{S_2}$、$\overline{S_1}$ 和 $\overline{S_0}$ 状态的任何改变，都用来指示相应总线周期的开始，而在 T_3 或 T_W 时则回到其被动状态，以表示该总线周期的结束。在"保持响应"时期，它们被浮置而处于高阻抗状态。三种状态的编码如表 2.10 所示。

表 2.10 $\overline{S_2}$，$\overline{S_1}$，$\overline{S_0}$ 的编码

$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	特　性	$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	特　性
0	0	0	中断响应	1	0	0	访问指令
0	0	1	I/O读	1	0	1	存贮器读
0	1	0	I/O写	1	1	0	存贮器写
0	1	1	暂停	1	1	1	被动状态

22. $\overline{RQ}/\overline{GT_0}$、$\overline{RQ}/\overline{GT_1}$　Request/Grant (请求/同意)

第 30、31 脚　输入，输出双向。

这两条引脚，是供两个外部处理器用来请求和获得局部总线控制权的。每个信道都是双向的，其中 $\overline{RQ}/\overline{GT_0}$ 的优先权比 $\overline{RQ}/\overline{GT_1}$ 的高。$\overline{RQ}/\overline{GT}$ 在片内已经接了一个拉高电阻，因而不用时可以悬空。总线请求/同意的时序如下：

(1) 一个时钟宽的脉冲，从局部总线上的其他主设备发向 8086，要求 8086 "保持" 并让出总线。

(2) 在 CPU 的 T_4 或 T_1 时钟周期8086送出一个时钟宽的脉冲通知该主设备，表示 8086 已经同意让出总线，并从下一时钟周期开始浮置到"保持响应"状态。在"保持响应"时期，8086 的总线接口部件 (BIU) 在逻辑上已和局部总线断开。

(3) 该主设备工作结束就送出一个时钟宽的脉冲给 8086，表示总线请求周期已经结束，8086 可在下一个时钟脉冲时收回局部总线的控制权。

每一次局部总线主控设备的改变，均需要一串三个脉冲的序列。每次总线改变之后，还必须有一个空闲的时钟周期。这些脉冲均是低电平有效。

如果总线请求，正逢8086在进行存贮器访问，当下列条件均满足时，CPU 将在 T_4 时让出局部总线：

(1) 请求是在 T_2 之前出现。

(2) 当前周期不是字的低 8 位周期。

(3) 当前周期不是中断响应时序的第一个周期。

(4) 当前不是在执行封锁指令。

如果局部总线空闲时，总线请求可能发生下列两种情况：

(1) 处理器将在下个时钟周期让出总线。

(2) 启动一个包含3个脉冲的存贮器访问周期。因为一旦条件 (1) 满足，就提供了启动当前存贮器周期所要满足的4个规定。

图2.45表示了请求/同意的时序。

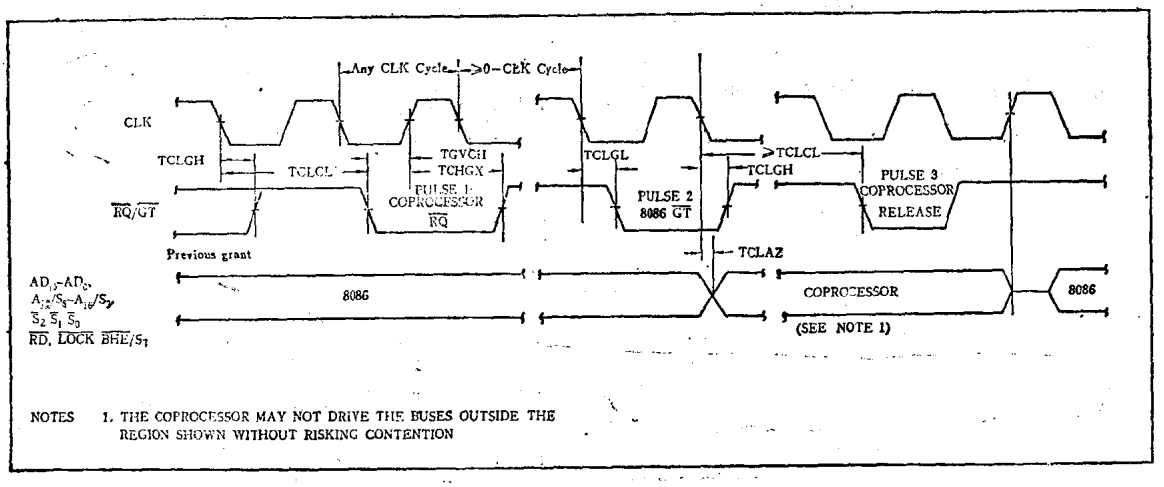


图2.45 $\overline{RQ}/\overline{GT}$ 时序图.

23. $\overline{LOCK}$ Lock (封锁)

第29脚 输出，三态.

当 $\overline{LOCK}$ 为低电平时，表示其他总线主设备不能获得系统总线的控制权. $\overline{LOCK}$ 由指令的封锁前缀来产生，并保持到下条指令结束. $\overline{LOCK}$ 为低电平有效. 在"保持响应"期间被浮置而处于高阻抗状态. 图2.46是 $\overline{LOCK}$ 信号的时序示意图.

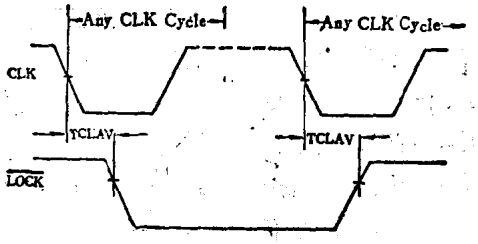


图2.46 总线封锁信号时序

24. QS_1、QS_0 Instruction Queue Status (指令队列状态)

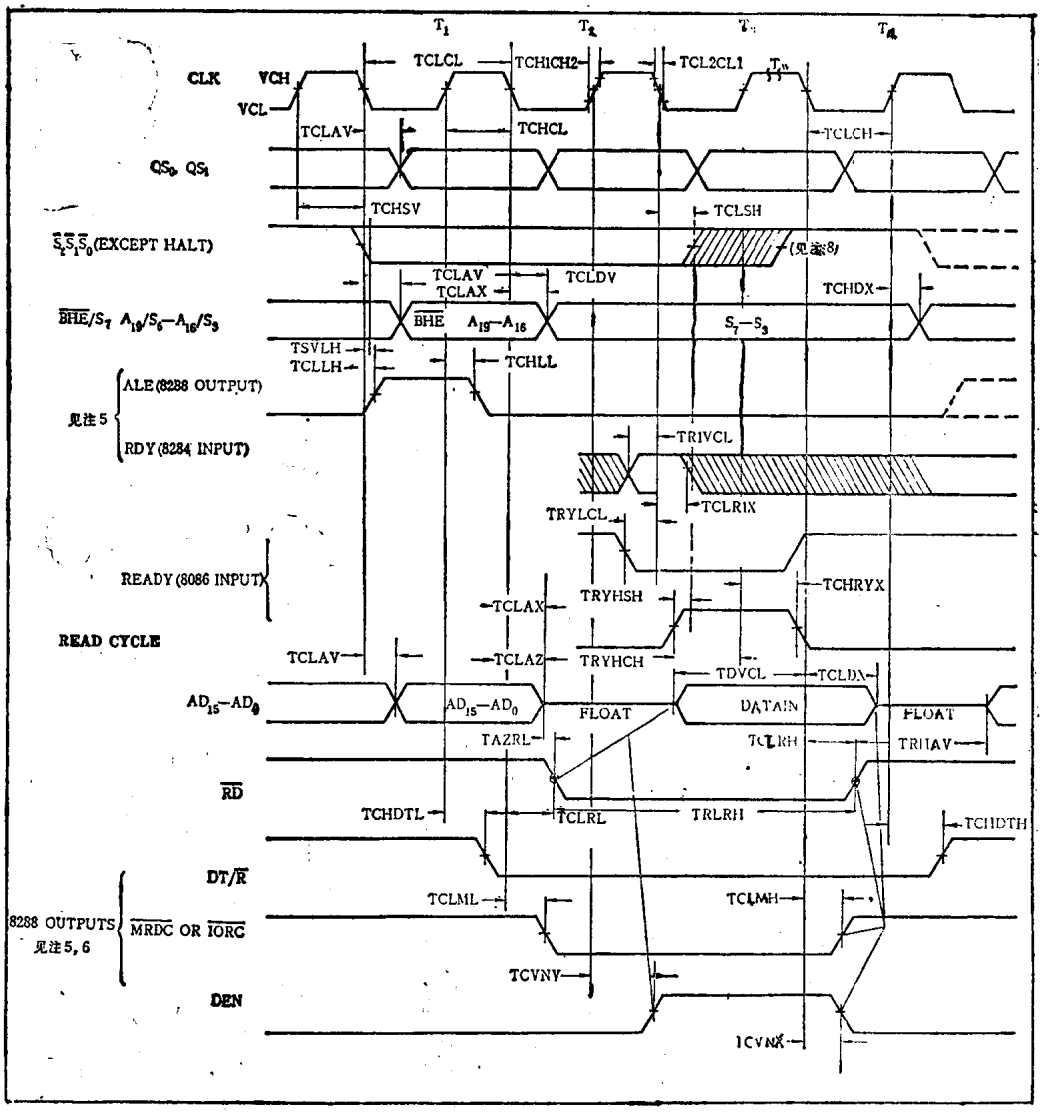
第24，25脚 输出.

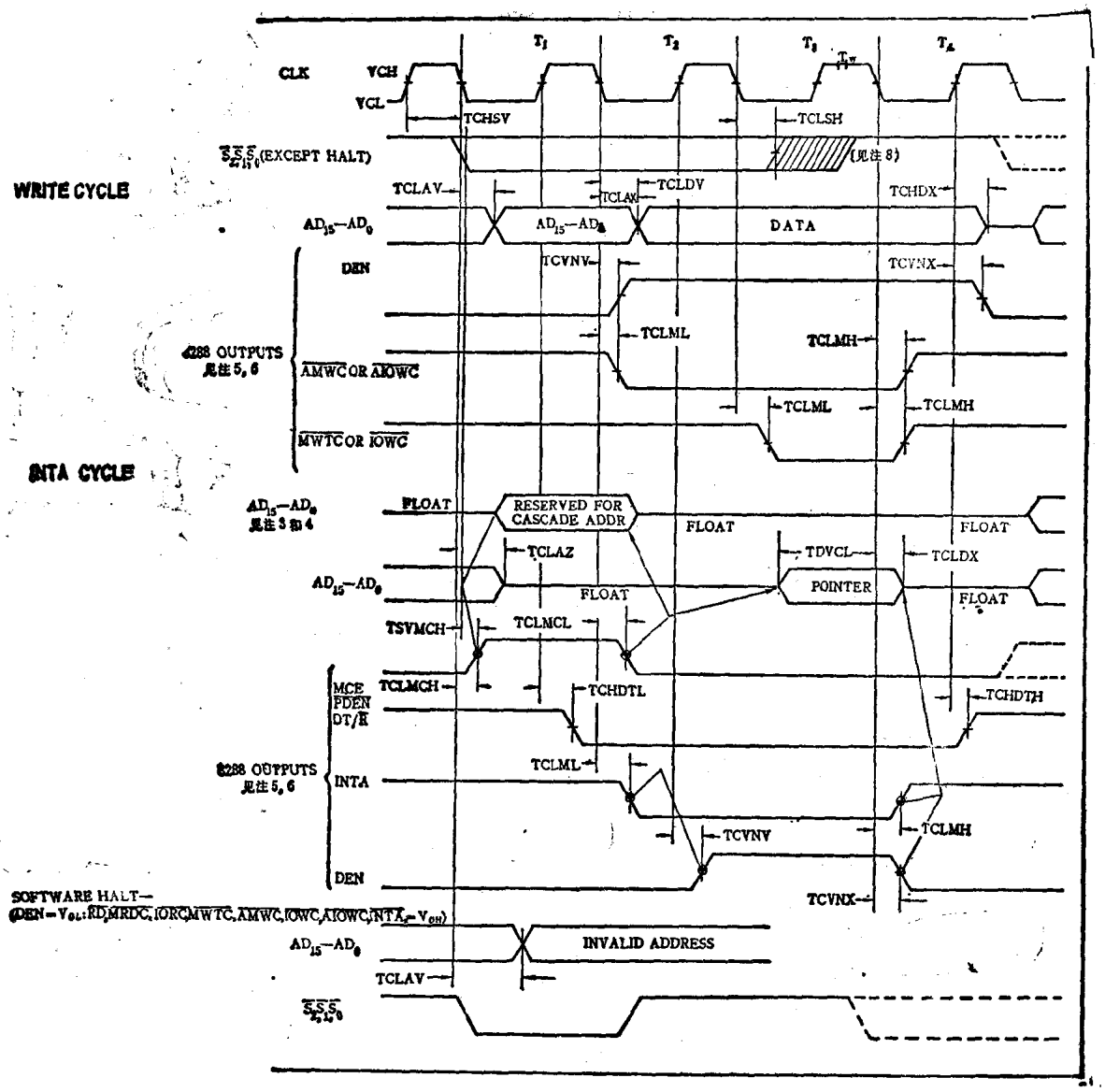
当排队已实施以后，QS_1 和 QS_0 提供队列的状态，以便于在外部对8086内部指令队列进行跟综. 表2.11表示了 QS_1，QS_0 的编码.

表 2.11 QS_1、QS_0 的编码

QS_1	QS_0	含　　义
0	0	无操作
0	1	来自队列中操作代码的第 1 个字节
1	0	空队列
1	1	来自队列的后续字节

图 2.47 是 8086 在最大模式系统中的定时图，从中可以看到，在最大模式下各引脚在时序上的相互关系。





注：1. 非经特别指出，所有的信号都在 V_{CH} 和 V_{CL} 之间转换.

2. RDY 在靠近 T_2、T_3、T_W 的末尾被采样，以决定是否要插入 T_W 机器周期.

3. 串联地址在两个 INTA 周期之间有效.

4. 两个 INTA 周期背靠背执行. 8086 局部地址/数据总线在两个 INTA 周期中浮置. 图中指示器地址是对第 2 个 INTA 周期表示的.

5. 8284 和 8288 的信号仅供参考.

6. 8288 命令和控制信号 ($\overline{MRDC}$, $\overline{MWTC}$ $\overline{AMWC}$ $\overline{IORC}$ $\overline{IOWC}$, $\overline{AIOWC}$, $\overline{INTA}$ 和 DEN) 滞后于高电平有效的 8288 CEN 引脚.

7. 除非另外注明，所有的时序均是在 1.5V 上测定.

8. 状态在进入 T_4 前失效.

图 2.47　8086 最大模式系统 (使用 8288) 的总线时序.

表 2.12 是 8086 最大模式系统 (使用 8288 总线控制器) 的定时 (参见图 2.45，图 2.46，图2.47).

表 2.12 8086 最大模式的时间

(a) 请求时间

Symbol	Parameter	8086/8086-4		8086-2 (Preliminary)		Units	Test Conditions
		Min.	Max.	Min.	Max.		
TCLCL	CLK Cycle Period—8086 —8086-4	200 250	500 500	125	500	ns	
TCLCH	CLK Low Time	(2/3 TCLCL) −15		(2/3 TCLCL) −15		ns	
TCHCL	CLK High Time	(1/3 TCLCL) + 2		(1/3 TCLCL) + 2		ns	
TCH1CH2	CLK Rise Time		10		10	ns	From 1.0V to 3.5V
TCL2CL1	CLK Fall Time		10		10	ns	From 3.5V to 1.0V
TDVCL	Data in Setup Time	30		20		ns	
TCLDX	Data in Hold Time	10		10		ns	
TR1VCL	RDY Setup Time into 8284 (见注 1,2)	35		35		ns	
TCLR1X	RDY Hold Time into 8284 (见注 1,2)	0		0		ns	
TRYHCH	READY Setup Time into 8086	(2/3 TCLCL) −15		(2/3 TCLCL) −15		ns	
TCHRYX	READY Hold Time into 8086	30		20		ns	
TRYLCL	READY Inactive to CLK (见注 4)	−8		−8		ns	
TINVCH	Setup Time for Recognition (INTR, NMI, TEST) (见注 2)	30		15		ns	
TGVCH	RQ/GT Setup Time	30		15		ns	
TCHGX	RQ Hold Time into 8086	40		30		ns	

(b) 响应时间

Symbol	Parameter	8086/8086-4		8086-2 (Preliminary)		Units	Test Conditions
		Min.	Max.	Min.	Max.		
TCLML	Command Active Delay (见注 1)	10	35	10	35	ns	
TCLMH	Command Inactive Delay (见注 1)	10	35	10	35	ns	
TRYHSH	READY Active to Status Passive (见注 3)		110		65	ns	
TCHSV	Status Active Delay	10	110	10	60	ns	
TCLSH	Status Inactive Delay	10	130	10	70	ns	
TCLAV	Address Valid Delay	10	110	10	60	ns	
TCLAX	Address Hold Time	10		10		ns	
TCLAZ	Address Float Delay	TCLAX	80	TCLAX	50	ns	
TSVLH	Status Valid to ALE High (见注 1)		15		15	ns	
TSVMCH	Status Valid to MCE High (见注 1)		15		15	ns	
TCLLH	CLK Low to ALE Valid (见注 1)		15		15	ns	
TCLMCH	CLK Low to MCE High (见注 1)		15		15	ns	
TCHLL	ALE Inactive Delay (见注 1)		15		15	ns	C_L=20—100pF for all 8086 Outputs (In addition to 8086 self-load)
TCLMCL	MCE Inactive Delay (见注 1)		15		15	ns	
TCLDV	Data Valid Delay	10	110	10	60	ns	
TCHDX	Data Hold Time	10		10		ns	
TCVNV	Control Active Delay (见注 1)	5	45	5	45	ns	
TCVNX	Control Inactive Delay (见注 1)	10	45	10	45	ns	
TAZRL	Address Float to Read Active	0		0		ns	
TCLRL	RD Active Delay	10	165	10	100	ns	
TCLRH	RD Inactive Delay	10	150	10	80	ns	
TRHAV	RD Inactive to Next Address Active	TCLCL-45		TCLCL-40		ns	
TCHDTL	Direction Control Active Delay (见注 1)		50		50	ns	
TCHDTH	Direction Control Inactive Delay (见注 1)		30		30	ns	
TCLGL	GT Active Delay	0	85	0	50	ns	
TCLGH	GT Inactive Delay	0	85	0	50	ns	
TRLRH	RD Width	2TCLCL-75		2TCLCL-50		ns	

注：1. 表示的 8284 或 8288 信号仅供参考。

2. 关于异步信号的建立请求仅在下一时钟周期确保识别。

3. 仅适用于 T_3 和等待状态。

4. 仅适用于 T_2 状态 (8ns 进入 T_3)。

三. 8086 处理器的结构

微处理器同一般计算机一样重复下列步骤来执行程序：

1. 从存贮器取下条指令；
2. 读操作数 (如指令要求的话)；
3. 执行指令；
4. 写结果 (如指令要求的话)。

在以前的处理器里，这些步骤大多是串行执行的，或者只用单总线周期交叉取。8086 在执行这些步骤时，把它们分配给 CPU 内的两个独立的处理部件去执行。执行部件 (EU) 执行指令，总线接口部件 (BIU) 取指令，读操作数和写结果。这两个部件能独立地进行操作，在多数情况下，取指令和执行指令能重迭进行。这样，通常取指令所需的时间"消失了"，原因是 EU 执行的指令已由 BIU 预先取出。图 2.48 说明了这种重迭，并同传统的微处理器操作进行了

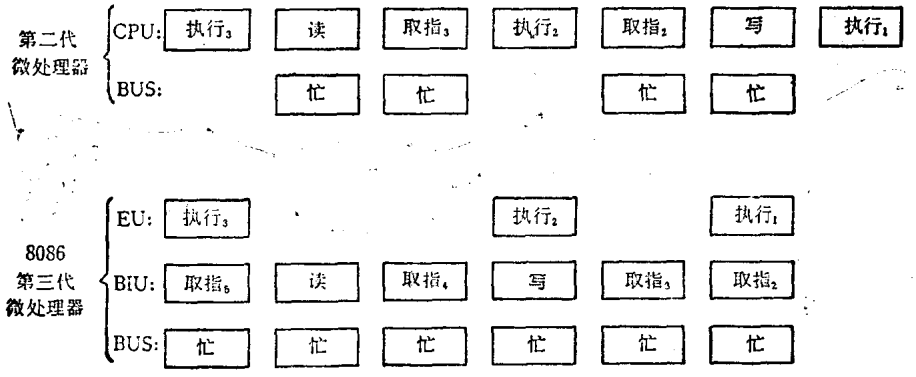


图 2.48 8086 取指令和执行指令的重迭

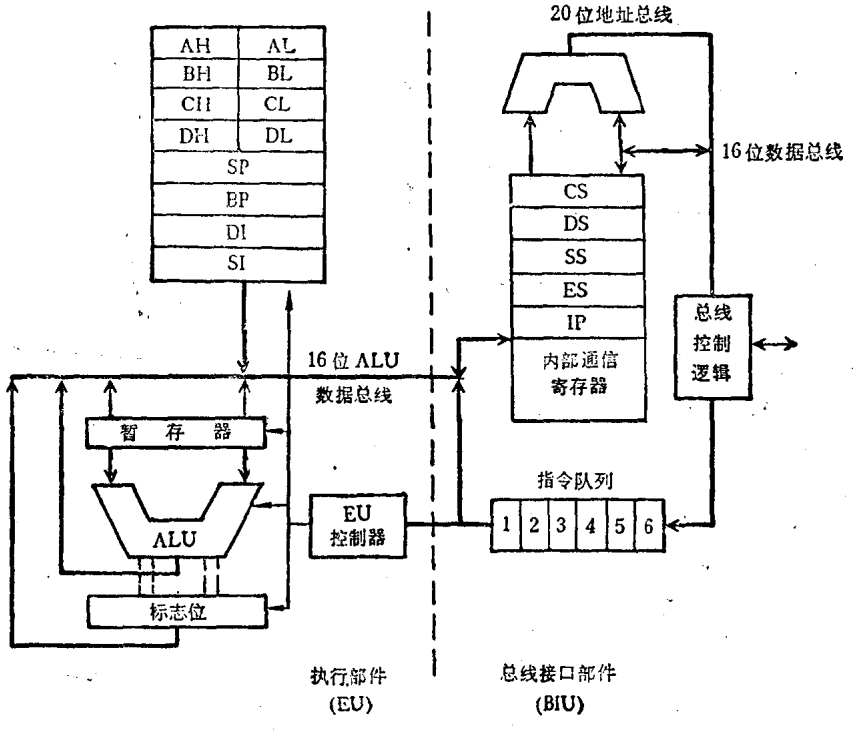


图 2.49 8086 的内部结构

比较．在该例中，重迭减少了3条指令所需的执行时间，还可预取两条另外的指令．

8086的内部结构见图2.49．从中看得很清楚，8086被分成了执行部件（EU）和总线接口部件（BIU）两部分．8086的这种结构，给取指令和执行指令的重迭执行提供了硬件支持．下面我们对这两个部分的主要结构和功能作简单的介绍．

（一）总线执行部件（EU）

总线执行部件中含有8个16位的寄存器（参见图2.49）它们被分成二组，即数据存贮器组（H和L组），及指示器变址器寄存器组（P和I组）．数据寄存器是唯一可按高，低字节分别寻址的，既可用作16位寄存器，也可用两个8位的寄存器．CPU的其他寄存器只能按16位存取．8086的有些指令，可以隐含地运用某些寄存器，使编码紧凑，功能更强．这些寄存器的隐含应用总结在表2.13中．

表2.13　某些寄存器的隐含运用

寄存器	操　　作	寄存器	操　　作
AX	字乘、字除、字I/O	CL	变量移位和环移
AL	字节乘、字节除、字节I/O、转换、10进运算	DX	字乘、字除、间接I/O
AH	字节乘、字节除	SP	堆栈操作
BX	翻译	SI	串操作
CX	串操作、环移	DI	串操作

总线执行部件中的16位算术逻辑部件（ALU），是用来对寄存器和指令操作数进行算术逻辑运算的．显然，状态控制标志也必须在执行部件中．为了快速传送，执行部件中，所有其他寄存器和数据通道都是16位宽的．

执行部件没有连接到系统总线上，对于系统总线来说，它是“外界的”．执行部件（EU）从总线接口部件（BIU）的排队机构中获得指令．同样，当指令要求访问存贮器或外围器件时，EU向BIU发出请求，由BIU通过总线获得存贮数据．

（二）总线部口部件（BIU）

BIU执行EU的所有总线操作，按照EU的要求，在CPU和存贮器或I/O器件间传送数据．由于EU所能提供的访问存贮器的地址是16位的，而8086访问1兆存贮器空间却需要20位地址，为了形成这种20位地址，就必须借助于在BIU中的4个段寄存器．处理器在某一时刻可以直接访问4个段（每个段可以多至64K字节），这些段的基地址就保存在这4个段寄存器中．我们已经知道，CS寄存器指示当前代码段，从这个段中取出指令．SS寄存器指示当前堆栈段，堆栈操作在这个段的单元上实现．DS寄存器指示当前数据段，它通常保存程序变量．ES寄存器指示当前附加段，一般用作数据存贮．

16位的指令指示器（IP）由BIU修改，它包含着下条指令从现行代码段开始的偏移量（字节距离），即IP指示出下条指令．正常执行时，IP包含BIU要取的下条指令的偏移量．然而，每当IP被保存在堆栈中时，首先自动调整到要执行的下条指令．程序不能直接访问指令指示器，但指令却可以使其改变．IP的值可以保存在堆栈中和从堆栈中恢复．

使8086处理器的处理速度得以提高的一个重要原因，是BIU具有预取指令的功能．这就是，在EU处于执行指令的忙周期期间，BIU从存贮器“超前”取出较多的指令，这些指令被

存放在 BIU 内部的 RAM 阵列，构成指令队列。在图 2.49 中可以看到8086 的排队机构，能存放 6 个指令字节，这个排队机构的规模，使系统总线基本上处于忙状态。每当 BIU 排队机构中空出两个字节，而没有来自 EU 的总线访问的有效请求时，就预取另一个指令字。如果程序转移迫使从奇数地址取指令时，BIU 自动从奇数地址读一个字节，然后再从邻接的偶数地址中取两个字节的字。

在多数情况下，排队机构至少保持一字节指令流，EU 才能不需等待取指令。排队机构中的指令，就是存放在同现行执行指令相连接，而又比它高的那些存贮器单元中的指令。就是说，只要是顺序执行，则排队机构中的指令，就是紧接在后面的逻辑上的指令。若 EU 执行转移指令，则 BIU 清除排队机构，从新地址取指令，并立即送给 EU，然后再从新的单元开始，重新填满排队机构。此外，当 EU 请求存贮器或 I/O 读或写的时候，BIU 就不取指令。

(三) 8086 处理器的启动

当 8086 的 RESET (复位) 引脚上被输入一个高电平时，处理器马上结束现行的操作，随着RESET 变回低电平，8086 就开始启动，启动后，处理器内部的各寄存器和标志位，被自动设置为如下值：

CS	0FFFFH
DS	0000H
SS	0000H
ES	0000H
IP	0000H
标志寄存器	0000H (禁中断)
排队机构	空

这时候处理器首先寻址位于存贮器地址 0FFFF0H (IP＋CS＊16) 的指令，所以 8086 的引导程序就放在那里。

现在，我们可以对存贮空间的使用作一小结了。并不是存贮器的所有单元都可以任意使用的。存贮器的最高和最低区域是留给某些特殊的处理器功能专用的，或是由 Intel 公司为 Intel 的软件和硬件产品保留的。如图 2.50 所示，这些单元是 0H 到 7FH (128 个字节，存放

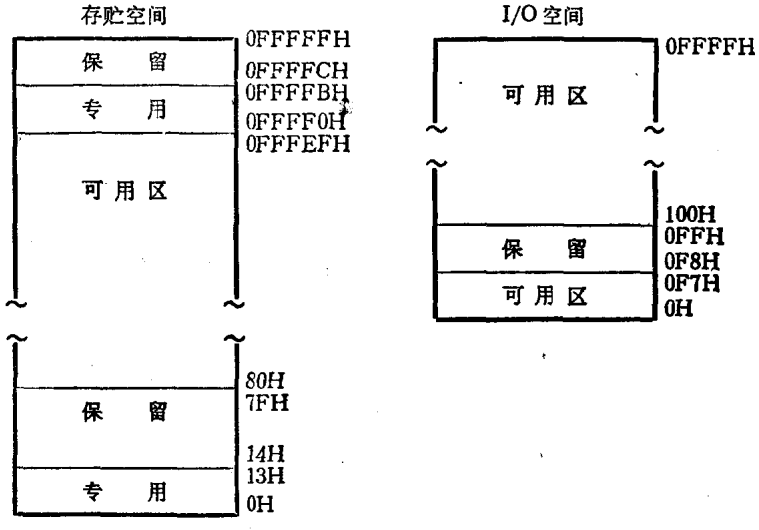
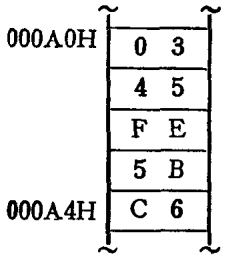


图 2.50 存贮空间和 I/O 空间的保留单元和专用单元

Intel 保留的 32 种中断指示器) 和 0FFFF0H 到 0FFFFFH (16 个字节,存放启动程序). 这些区域用于中断和系统复位处理, 8086 应用系统不能把这些区域改作其他用途, 否则会使系统与未来的 Intel 产品不兼容. 同样,在 I/O 空间, 0F8H 到 0FFH 的 8 个单元也是 Intel 公司保留的 8 个单元 (见图 2.50).

习　　题

1. 有2个 16 位字 807F、5FEF, 它们在一个 8086 系统的存贮器中的地址,分别为 00020H 和 00023H, 请用图表示出它们在存贮器中的位置.
2. 在一个 8086 系统的存贮器中放着如下的信息, 现在处理器要读一个地址为 000A3 的字, 试说出该字的内容读取读字的过程.



3. 8086 的寄存器结构是怎样的? 它们和 8080 的寄存器有什么关系? 这样的关系有什么好处? 8080 中的 C 寄存器和 D 寄存器各自对应 8086 的何种寄存器.
4. AX, BX, CX,DX 作为寄存器名称,并不是 A、B、C、D 字母的顺序排列,它们分别是 Accumulator、Base、Count、Data 的缩写,试说明给它们起这些名称的道理.
5. 为什么 8086 采用使某些寄存器专用于某些指令的结构,并最终可使代码长度缩短?
6. 8086 的 SP、BP、SI、DI 寄存器的名字是从哪里引伸出来的,它们各有何种特殊的用途?
7. 在某个程序的运行过程中, 堆栈的栈顶单元偏移地址是 7F80H, 而堆栈中的一个数据区的基偏移地址为 7CA4H, 当程序要引用该数据区中的数据时, 这两个地址应当放在哪两个寄存器中? 一条串传送指令,要把在当前数据区中开始于偏移地址 4000H 的字符串, 传送到附加段中开始于偏移地址 1000H 的地方去,传送第一个字符时 SI 和 DI 的内容各是什么?
8. INC 指令的寄存器寻址方式的操作码是 01000, 现在要把 SP 的内容加 1, 该指令 reg 字段应是什么?
9. 一条 INC 指令的格式是这样的

opcode	W	mod	opcode	r/m	位移量	位移量
1 1 1 1 1 1 1	1	1 0	0 0 0	0 0 0	0 1 0 1 1 1 0 0	0 0 1 0 1 1 0 1

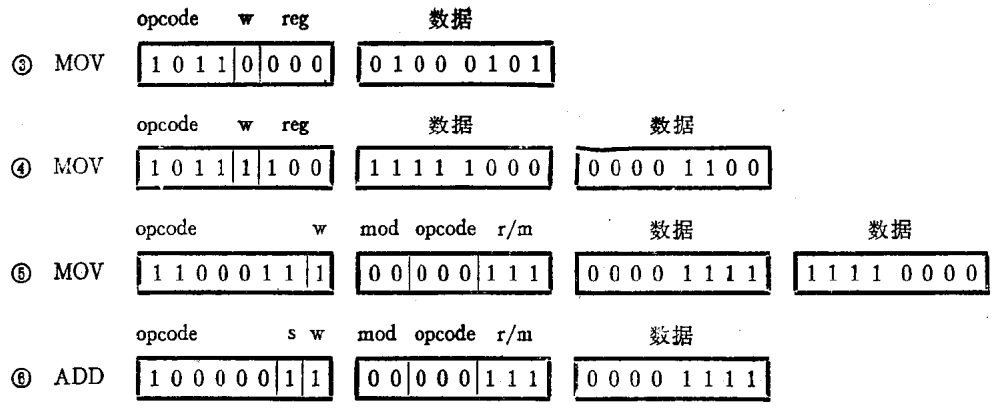
, 设 DS 的内容为 1111 0000 1111 0000, SI 的内容为 1010 0000 1000 0110,BX 的内容为 0100 0001, 试计算出操作数的实际地址,并用图表示形成的过程.
10. 算出图 2.17 中的操作数实际地址、ES 的值设为 0001 0000 1000 0000, 其余条件同第 9 题.
11. mod=00, r/m=110 这种间接寻址方式给直接寻址用掉了, 现在要写一条用到 BP 寄存器来形成偏移地址的指令,怎么写?
12. 在双操作数指令中, W 字段和 d 字段的作用是什么?立即操作数指令中的 S 字段的作用是什么?下面是几条指令,试找出它们的源、目操作数.

① ADD

opcode	d	w	mod	reg	r/m
0 0 0 0 0 0	0	1	1 1	0 0 1	0 1 1

② ADD

opcode	d	w	mod	reg	r/m
0 0 0 0 0 0	1	0	0 0	1 1 1	0 1 0



13. 简述 8086 各种寻址方式，它们对使用高级语言各自提供了何种支持？

14. 8086 内部分成那二大部件，它们各自的组成和功能是什么？

15. 8086 预取指令队列有什么好处？简述预取指令机构的工作情况。

16. 8086 的存贮器和 I/O 空间各有哪些保留区域，它们各派什么用处？

第三章　8086 指令系统

前一章中叙述了指令操作数的寻址方式，这一章着重叙述指令的执行。指令是用一种非正式的方式来叙述的，正式详细的介绍，可以在本章的 §8.12 中找到。

8086 的指令具有通用的(长)形式和约束的(短)形式。短的形式使用较少的字节，但在允许使用的操作数方面，有较多的限制。使用短形式的目的，是为了在最经常的情况下，能用最少的字节数来编制程序。例如，普通形式的 PUSH 指令，把一个在寄存器中或在存贮器中的操作数推入堆栈。它需要两个字节来指定操作数。短形式的 PUSH 只对寄存器操作，因而只有一字节长。前面已经提到过，除非用机器码写程序，我们并不要去关心指令具有多种形式；一个好的汇编程序将会选择最有效的形式来翻译指令。附录 A 总结了每条指令可能有的形式，附录 B 列出了所有的操作码。

为了叙述方便，把指令分成下列 8 类：

1. 数据传送指令
2. 算术指令
3. 逻辑指令
4. 串指令
5. 控制转移指令
6. 中断指令
7. 标志指令
8. 同步指令

§3.1　数据传送指令

8086 具有 4 种数据传送指令：通用传送，累加器专用传送，地址目标传送和标志传送。表 3.1 列出了这 4 类指令。

表 3.1　数据传送指令

通用型	
MOV (move)：	源→目
PUSH (push)：	源→堆栈
POP (pop)：	堆栈→目
XCHG (exchange)：	源↔目
累加器专用型	
IN (input)：	转接口→AL 或 AX
OUT (output)：	AL 或 AX→转接口
XLAT (translate)：	f(AL)→AL
地址目标传送型	
LEA (load effective address into register)：	源偏移量→寄存器
LDS (load pointer into register and DS)：	源，源+1→寄存器 源+2，源+3→DS
LES (load pointer into register and ES)：	源，源+1→寄存器 源+2，源+3→ES
标志传送型	
LAHF (load AH with flags)：	SF，ZF，AF，PF，CF→AH
SAHF (store AH into flags)：	AH→SF，ZF，AF，PF，CF
PUSHF (push flag)：	标志→堆栈
POPF (pop flag)：	堆栈→标志

一、通用传送

通用传送指令是MOV(move)、PUSH、POP和XCHG(exchange)。段寄存器可用作这些指令的一个操作数，使新的值能放入到段寄存器中去。其他指令不允许把段寄存器作为操作数。在通用传送指令中，段寄存器用2位seg字段指定，此时；00指ES，01指CS，10指SS，11指DS；若指令具有一个d字段，则d=1表示段寄存器是目操作数。

MOV指令把源操作数的一个字节或字传送到目操作数。操作数中的一个用mod字段和r/m字段指定，另一个操作数可以由一个reg字段，一个seg字段(段寄存器操作数)或一个数据字段(立即操作数)指定。为了使经常出现的指令优化，8086提供了若干短形式的MOV指令，如图3.1所示。

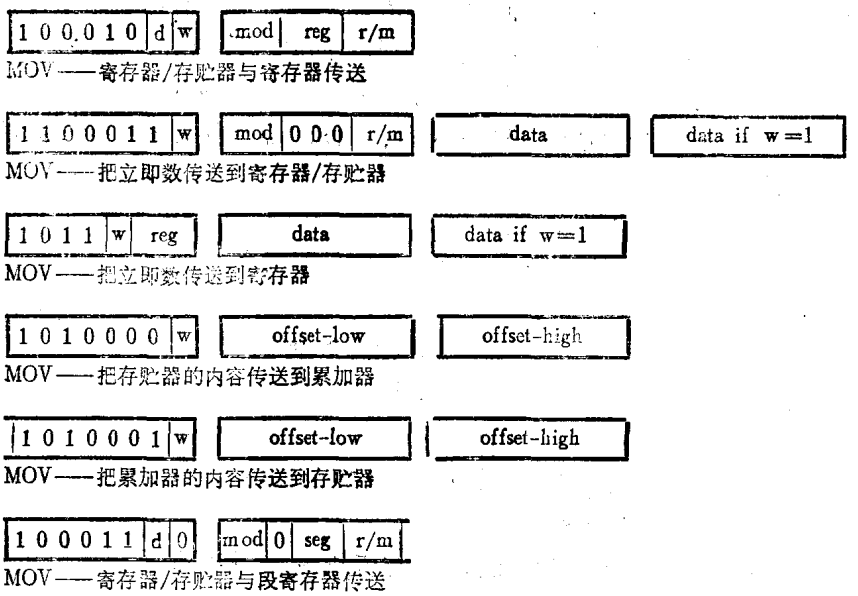


图3.1 MOV指令的格式

8086的堆栈操作，总是按字进行的。PUSH指令把源操作数的一个字"压入"堆栈。POP指令正好相反，它从堆栈中"弹出"一个字到目操作数。堆栈是存贮器的一部分。SP寄存器中，含有最后压入堆栈的字的偏移量。这个字称为堆栈顶。由于后续的字要压在当前的堆栈顶上，因此它们将被连续地放到较低的存贮器地址(堆栈向较低的地址方向生长，而向较高的地址方向收缩)。PUSH指令把SP的内容减2，从而找到堆栈中下一个空白的字。POP指令在把堆栈顶上的字送出以后，把SP的内容减2，从而从堆栈中去掉刚才传送的字。图3.2举例说明了PUSH指令和POP指令的执行效果。

一条PUSH或POP指令的操作数由一个mod、r/m字段，一个reg字段或一个seg字段指定，如图3.3所示。

有一个问题要引起注意，即若执行一条改变CS寄存器内容的指令将会使一个新的段成为当前代码段，此时IP却仍然指示着以前代码段中的下一条顺序指令.这样，CS和IP内容的组合，就是一个没有意义的存贮器地址，而处理器却要试图从该无意义的地址中去取下一条指令来执行。所以，除非改变CS内容的指令在改变CS值的同时，也给IP放一个相对有效的值，否则处理器就将不能正常地工作。由于这个原因，那些允许把段寄存器作为一个操作数的

指令，不能使用 CS 寄存器。这种情况发生在 MOV 指令和 POP 指令的 seg 字段指定 CS 作为目操作数的时候。当遇到这样的指令时，处理器的动作没有定义。这些无定义的指令表示在图 3.4 中。

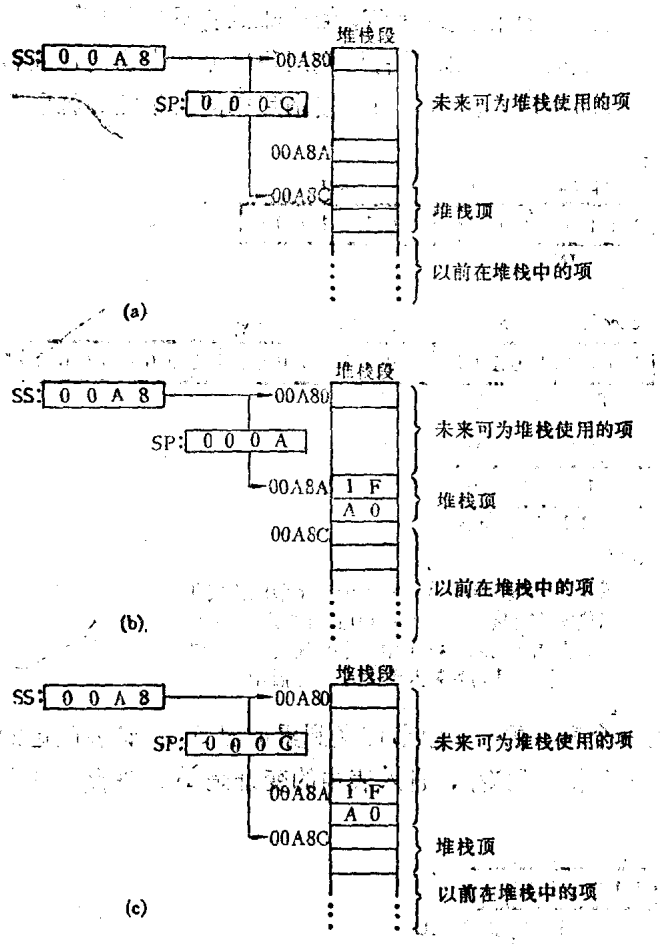


图 3.2 在堆栈中压入和弹出项目的例子：(a)堆栈的初始结构；(b)在执行了把值0A01FH推入堆栈的 PUSH 指令后的堆栈结构；(c)在执行了一条 POP 指令后的堆栈结构。

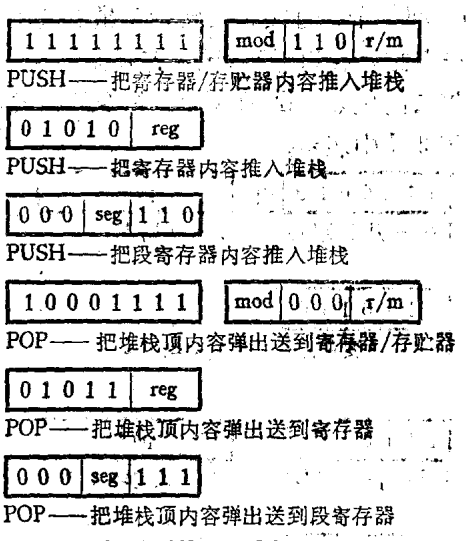


图 3.3 PUSH 和 POP 指令的格式

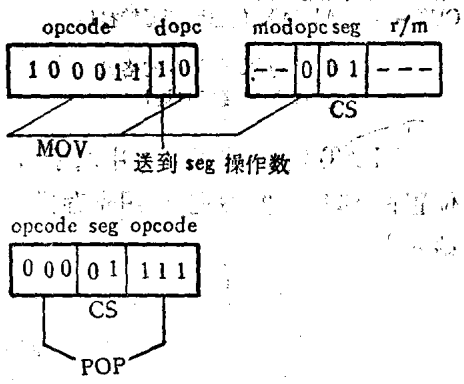


图 3.4 无定义的指令：(a)把一个新值传送到 CS；(b)从堆栈中弹出一个值送到 CS。

最后一条通用数据传送指令是 XCHG 指令。这条指令在两个操作数之间，执行一个字节或字的交换。在这里没有必要把操作数区分为源和目，所以指令没有 d 字段(因而放入另一个操作码)。XCHG 指令具有如图 3.5 所示的通用形式和短的形式。

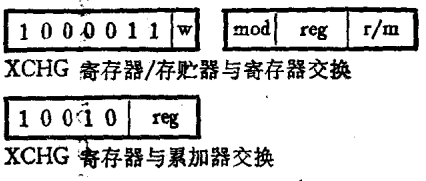


图 3.5 XCHG 指令的格式

二、累加器专用传送型

累加器专用传送指令包括 IN(input)、OUT(output)、和 XLAT (translate)。与以前把除了段寄存器之外的寄存器都一样对待的传送指令不一样，这些传送指令只允许累加器作为操作数。它们做成累加器专用，是为了在指令中可以不要任何 mod、r/m 或 reg 字段。

IN 指令从一个输入转接口中，把数据送到累加器(AL 或 AX)。同样，OUT 指令把数据从

累加器送到一个输出转接口。转接口号码可以由指令中的一个字节直接指定，或由 DX 寄存器的内容间接指定。注意，这是 DX 寄存器的一种特殊用法，其他的寄存器都不能用于这种功能。在指令中只有 256 个转接口可以被直接指定，但间接指定的转接口，却可以多达 2^{16} (约 65000) 个。直接指定转接口虽然要求指令包含一个附加的字节，但是却具有不需要预先执行一条把转接口号码，送入寄存器的指令的优点，间接访问则具有可利用程序循环来访问连续转接口的优点。IN 和 OUT 指令的格式在图 3.6 中表示。转接口直接指定和间接指定的区别在图 3.7 中表示。

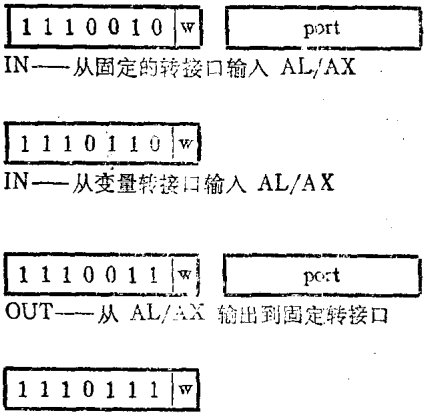


图 3.6 IN/OUT 指令的格式
(port 为转接口号码)

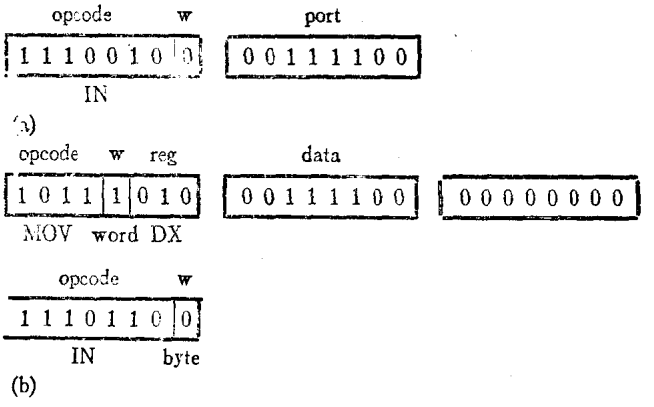


图 3.7 从转接口 3CH 中输入字节的直接和间接指定转接口的对比：(a) 直接指定，转接口号码在指令的附加字节中；(b) 间接指定，转接口号码先装入到 DX 寄存器中。

XLAT 指令(在图 3.8 中表示)，把一个表中的一个字节传送到累加器 AL。该表的起始位置由BX寄存器指定(通用寄存器的另一种特殊用途)，进入表中的变址是 AL 寄存器的原始内容。

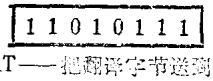


图 3.8 XLAT 的指令格式

XLAT 指令在把一个值从一种编码，翻译成另一种编码时很有用。例如，考虑下列 10 进制数字 0 到 9 的另一种编码：

数字	编码 (5 中取 2 码)
0	11000
1	00011
2	00101
3	00110
4	01001
5	01010
6	01100
7	10001
8	10010
9	10100

这种编码称为 5 中取 2 码，因为每个编码中，均含有两个 "1" 位。这种编码在电话通讯中，

得到实际的应用。假定我们要把2进制数字7翻译成5中取2码，则执行这个翻译的步骤如下：

1. 把编码表的起始位置的偏移量送入BX；
2. 把2进制7 (0000 0111)送入AL。
3. 执行XLAT指令——从表中取出第7项(0001 0001)，并把它送入AL。

这个翻译的过程在图3.9中说明。

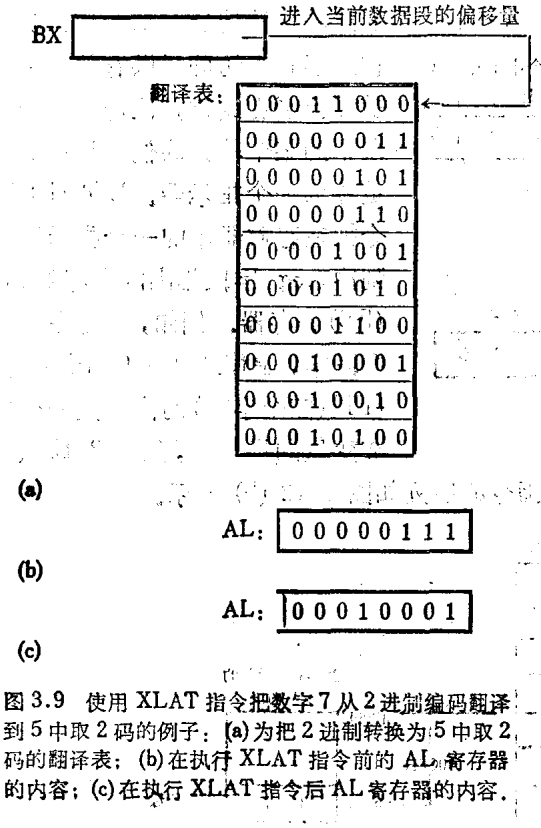


图3.9 使用XLAT指令把数字7从2进制编码翻译到5中取2码的例子：(a)为把2进制转换为5中取2码的翻译表；(b)在执行XLAT指令前的AL寄存器的内容；(c)在执行XLAT指令后AL寄存器的内容。

三、地址目标传送型

地址目标传送指令，有LEA (load effective address)、LDS (load pointer into register and DS) 和LES (load pointer into register and ES)。这些指令给程序员提供了对寻址机构的控制，它们的指令格式表示在图3.10中。注意，虽然这些指令使用了mod和r/m字段来指定一个操作数和用一个reg字段来指定另一个操作数，但却没有一个d字段来指定哪一个是源操作数，那一

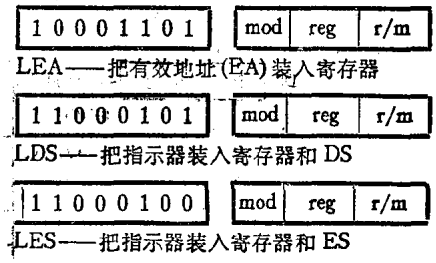


图3.10 地址目标传送指令的格式

个是目操作数。由于这些指令的源操作数总是来自或引用自存贮器，所以必须由mod和r/m字段指定。由此可见，在这些指令中d字段是不必要的。关于这一点，在叙述每条地址目标传送指令时，就会变得很清楚。

LEA指令取得的不是操作数的值，而是源操作数的偏移地址，因此，若源操作数不引自存贮器，这条指令就毫无意义。LEA指令的作用，是把源操作数的16位偏移地址，传送到被指定为目操作数的16位寄存器中去。这种指令可以把某一个变量的偏移地址，从程序的一个部分传送到另一个部分，从而使程序的其他部分能够修改该变量的值。

在程序的两个不同部分间传送的目标称为参数，而这些程序的不同部分称为子程序。例如，一个子程序具有把变量的值加1的功能。程序的其他部分可以调用这个加1子程序，并且用它来把一个特定的变量的值加1.这个要加1的变量的偏移地址可以作为一个参数，并在调用加1子程序前，把它放到一个相互确认的寄存器，例如BX中，然后通过BX再把该变量的偏移地址传送到加1子程序。LEA指令就是专做这项工作的。在上述例子中LEA指令的reg段字将指定BX寄存器(011)而mod和r/m字段则将指定该变量的偏移地址。这条指令

要在调用子程序之前执行，这样，子程序就能使用含有 BX (mod=00, r/m=111) 的适当的寻址方式来访问该变量。

LDS 指令把源操作数的 4 个连续字节 (32 位)，传送到一对 16 位目寄存器中。源操作数必须在存贮器中。一个目寄存器由指令中的 reg 字段指定，另一个目寄存器则规定为 DS。LES 指令除了另一个目寄存器规定为 ES 而不是 DS 以外，与 LDS 指令的功能完全相同。数据传送的情况在图 3.11 中说明。LDS 和 LES 指令，为建立一个段的起始地址和在该段中的一个变量的偏移地址，从而使该变量能被后续指令访问提供有效的手段。前面已经提到过，段起始地址和该段中的某个单元的偏移地址的组合被称为一个指示器。LDS(LES) 指令把在存贮器中的一个指示器传送到由 reg 字段指出的寄存器和 DS (ES) 寄存器。例如，假定在当前数据段中，偏移地址从 0F1CH 到 0F1FH (四个字节) 的内容是一个单字节变量的指示器，如图 3.12 (a) 所示。为把这个变量的值装入 AL 寄存器的两条指令的序列如图 3.12 (b) 所示。

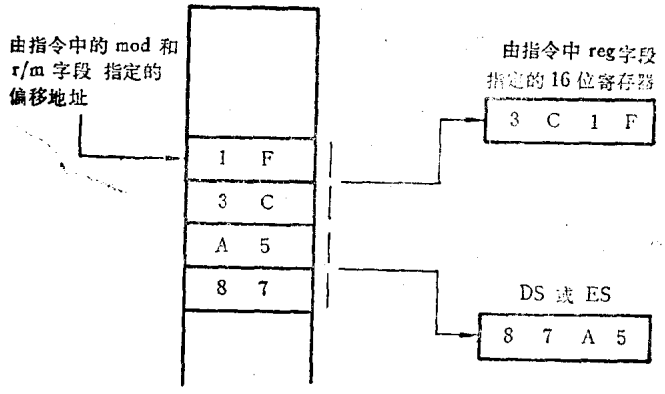


图 3.11 关于 LDS 和 LES 指令的数据传送

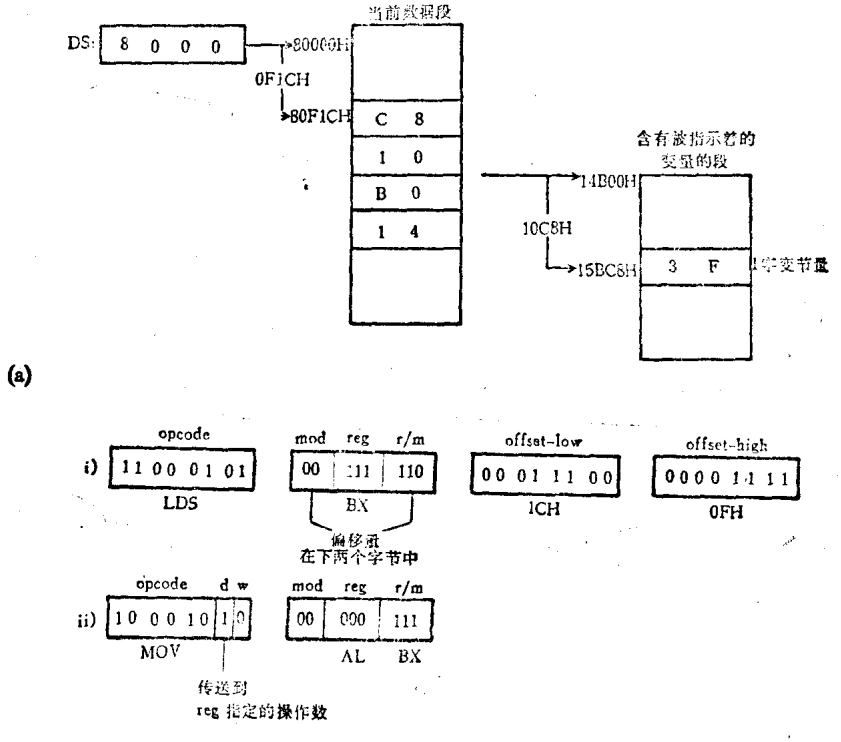


图 3.12 使用 LDS 指令的例子：(a) 存贮器含有一个指向一个变量的指示器；(b) 指令 i) 把指示器装入寄存器 DS 和 BX；ii) 使用含有 DS 和 BX 的操作数寻址方式去访问被指着的变量。

四、标志传送型

标志传送指令(图 3.13),提供了对处理器标志集的访问。这些指令是 LAHF (load AH with flags)、SAHF (store AH into flags)、PUSHF(push flags) 和 POPF (pop flags)。

LAHF 指令把标志寄存器的 SF(符号标志)、ZF(0 标志)、AF(辅助进位标志)、PF (校验标志)和 CF(进位标志)传送到 AH 寄存器的规定位。SAHF 指令把 AH 寄存器的规定的位传送到这些标志位。这 5 个标志单独列出没有其他的理由，就因为它们是出现在 8080/8085 处理器中的 5 个标志。设置 LAHF 和 SAHF 指令，主要就是为了能把为 8080/8085 写的程序，翻译成有效的 8086 程序。5 个标志在 AH 中的对应位在图 3.14 中表示。

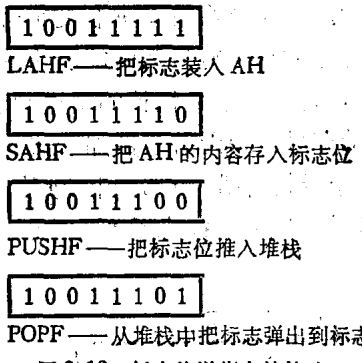


图 3.13 标志传送指令的格式

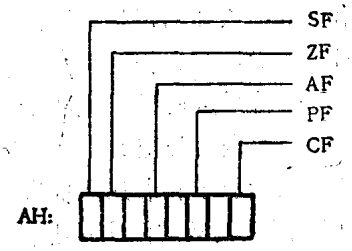


图 3.14 标志位和AH的位之间的对应

PUSHF 指令在堆栈顶上压入一个存放有所有 9 个标志的的字。POPF 指令则从堆栈中弹出一个字，并把该字的对应位传送到 9 个标志位寄存器。堆栈字位和 9 个标志之间的对应关系在图 3.15 中表示。

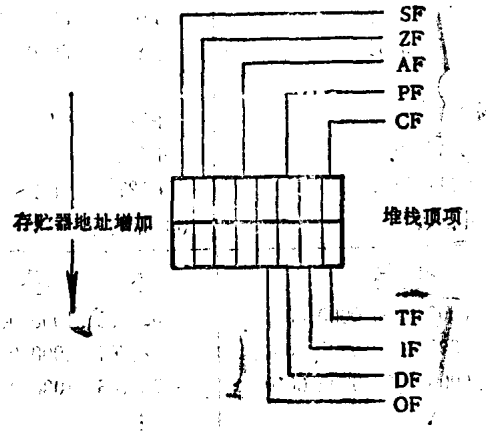


图 3.15 在堆栈中标志和位之间的对应关系

§3.2 算 术 指 令

8086 以多种形式提供 4 种基本的算术运算。8085 的算术指令列在表 3.2 中。8086 提供了 8 位和 16 位的带符号或无符号运算，还提供了允许直接以 10 进制而不是 2 进制数进行运算的校正操作。

表3.2 算术指令

加法		CMP (compare)：	目－源→?
ADD (add)：	目＋源→目	乘法	
ADC (add with carry)：	目＋源＋CF→目	MUL (multiply)：	AL*源$_8$→AX 或 AX*源$_{16}$→DX，AX
INC (increment)：	目＋1→目	IMUL (integer multiply)：	除了是带符号乘法外同上
减法		除法：	
SUB (subtract)：	目－源→目	DIV (divide)：	AX/源$_8$→AL；余数→AH 或 DX，AX/源$_{16}$→AX；余数→DX
SBB (subtract with borrow)：	目－源－CF→目	IDIV (integer divide)：	除了是带符号除法外同上
DEC (decrement)：	目－1→目		
NEG (Negate)：	0－目→目		

带符号数和无符号数之间的区别，在于位格式的翻译。无符号数用2进制的记号翻译，带符号数用第一章中叙述的记号翻译。图3.16表示了带符号数和无符号数所能表达的数值范围。加法和减法操作是在相同类型的数上进行的。这样，为无符号数设计的2进制加法和减法指令应用于带符号数的时候也能给出正确的结果。在带符号数和无符号数加减法之间，唯一不同的是检测结果是否溢出的机构。若结果被翻译成一个无符号数，用加法和减法指令时，CF标志置"1"表示溢出；若结果被翻译成一个带符号数时，则OF标志置"1"表示溢出。图3.17说明了不同种类数的溢出情况。

无 符 号

数	表达法
0	0000 0000
1	0000 0001
2	0000 0010
⋮	⋮
126	0111 1110
127	0111 1111
128	1000 0000
⋮	⋮
253	1111 1101
254	1111 1110
255	1111 1111

(a) 无符号8位数

带 符 号

数	表达法
−128	1000 0000
−127	1000 0001
−126	1000 0010
⋮	⋮
−1	1111 1111
0	0000 0000
+1	0000 0001
⋮	⋮
+125	0111 1101
+126	0111 1110
+127	0111 1111

(b) 带符号8位数

数	表达法
0	0000 0000 0000 0000
1	0000 0000 0000 0001
2	0000 0000 0000 0010
⋮	⋮
32,766	0111 1111 1111 1110
32,767	0111 1111 1111 1111
32,768	1000 0000 0000 0000
⋮	⋮
65,533	1111 1111 1111 1101
65,534	1111 1111 1111 1110
65,535	1111 1111 1111 1111

(c) 无符号16位数

数	表达法
−32,768	1000 0000 0000 0000
−32,767	1000 0000 0000 0001
−32,766	1000 0000 0000 0010
⋮	⋮
−1	1111 1111 1111 1111
0	0000 0000 0000 0000
+1	0000 0000 0000 0001
⋮	⋮
32,765	0111 1111 1111 1101
32,766	0111 1111 1111 1110
32,767	0111 1111 1111 1111

(d) 带符号16位数

图3.16 8位和16位带符号和无符号数的范围

例　子	表示法	翻译为无符号数	翻译为带符号数
(a) 带符号数和无符号数的结果都在范围中	0000 0100 + 0000 1011 0000 1111	4 11 15　CF=0	+4 +11 +15　OF=0
(b) 无符号数结果溢出	0000 0111 + 1111 1011 0000 0010	7 251 2　CF=1 溢出	+7 −5 +2　OF=0
(c) 带符号数结果溢出	0000 1001 + 0111 1100 1000 0101	9 124 133　CF=0	+9 +124 −123　OF=1 溢出
(d) 带符号数和无符号数都溢出	1000 0111 + 1111 0101 0111 1100	135 245 124　CF=1 溢出	−121 −11 +124　OF=1 溢出

图 3.17　带符号数和无符号数加法结果溢出的例子

大多数算术操作，把 6 个状态标志置位或复位以反映操作结果的某些性质。刚才讨论了这些标志中的两个即 CF 和 OF，一般可用这 6 个标志置位与否来识别下列情况：

1. 若操作结果是一个无符号数溢出，则 CF 置位。
2. 若操作结果是一个带符号数溢出(称为符号溢出)，则 OF 置位。
3. 若操作结果是 0 (带符号或无符号)，则 ZF 置位。
4. 若操作结果的最高位是一个“1”从而指示一个负的结果，则 SF 置位。
5. 若操作结果含有偶数个“1”位(称为偶校验)，则 PF 置位。
6. 若对 10 进制数操作需要调整(详细讨论在后面)，则 AF 置位。

这些标志的特性总结在 §3.11 的末尾。

多精度运算是把一个数分成多个 8 位或 16 位字段，并从最低字段开始的连续字段运算。这是一种处理大于 16 位的无符号数的手段。在这些操作中的任何一个中间运算产生溢出时，该结果仍然有效，只不过有一个“1”进(加)到下一个字段的操作上或从下一个字段的操作上借“1”(减)。例如，把 24 位数 0011 1010 0000 0111 1011 0010 加上 24 位数 0100 0000 1100 0010 0101 0011 可以用连续 3 次 8 位加法来实现，具体做法如下：

1. 最低 8 位相加：

$$\begin{array}{r} 1011\ 0010 \\ +\ 0101\ 0011 \\ \hline 0000\ 0101 \end{array}\quad CF=1$$

2. 中间 8 位和由上次加法所产生的进位相加：

$$\begin{array}{r} 1 \quad (\text{上次 CF}) \\ 0000\ 0111 \\ +\ 1100\ 0010 \\ \hline 1100\ 1010 \end{array}\quad CF=0$$

3. 最高 8 位和由上次加法所产生的进位相加：

```
          0   (上次 CF)
     0011 1010
+    0100 0000        CF=0
-----------------
     0111 1010
```

这样结果是0111 1010 1100 1010 0000 0101. 这个例子指出了需要有一条把两个操作数和在CF中的值相加的指令，以及另一条类似的指令，即带借位减指令.

无符号数加法或减法的结果溢出在执行诸如多精度运算时，是可以预想得到的.这是一种正常的事，并不表示一种错误情况.但是，带符号结果的溢出却常常是不能预料的，它表示一个错误已经发生，并且在计算能进行之前该结果必须进行调整(指两个正数相加得到了负的结果或两个负数相加得到了正的结果).

一、加法指令

加法指令有ADD (add)、ADC (add-with-carry)和INC 。(increment). 这些指令可以对任何操作数进行操作.

ADD 指令(图 3.18)，对源和目操作数的内容执行一个字节或字的加法，并把结果存放回目操作数. 其中一个操作数可以在一个寄存器中或在存贮器中(由 mod, r/m 字段指定)；另一个操作数可以在一个寄存器中(由 reg 字段指定)或在指令中(立即字段)，立即数ADD指令具有一般形式和短指令形式.

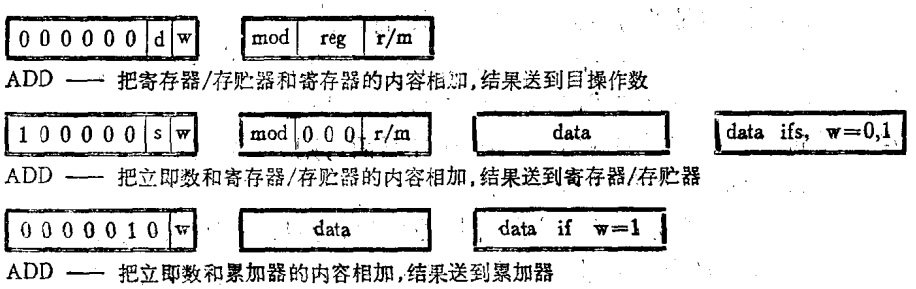


图 3.18 ADD 指令格式

ADC 指令除了在加法中包括 CF 的初始值之外与 ADD 指令类似，这条指令为前面讨论的多精度运算提供了方便. ADC 指令的形式与 ADD 指令的形式相同，它们在图 3.19 中表示.

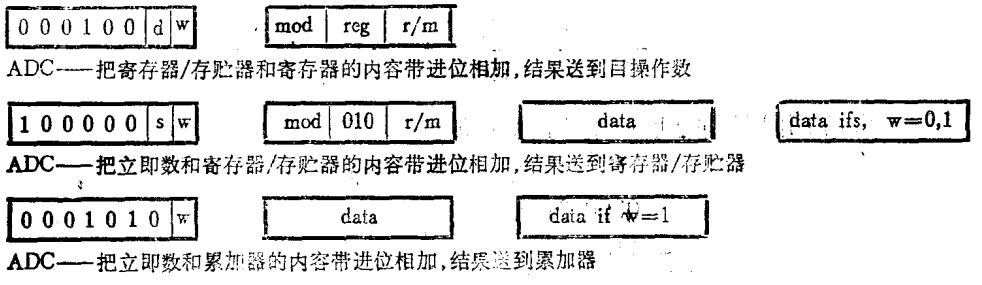


图 3.19 ADC 指令的格式

INC 指令只有一个操作数. 该指令把操作数的内容加 "1"，并把结果存放回该操作数.

INC 指令具有一般的形式和短的形式，如图 3.20 所示。

INC 指令相当于一条带有一个立即操作数 1，但需要更少字节的 ADD 指令。因为加 1 (或减 1) 是非常频繁的操作，故使用的字节越少越好。这就是 INC 指令包含在这个指令集中的原因。

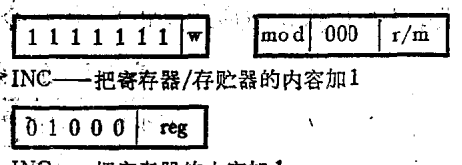


图 3.20 INC 指令的格式

二、减法指令

减法指令有 SUB (subtract)、SBB (subtract with borrow)、DEC (decrement)、NEG (negate) 和 CMP (compare)。前 3 个与 3 条加法指令类似，它们的格式在图 3.21 中表示。

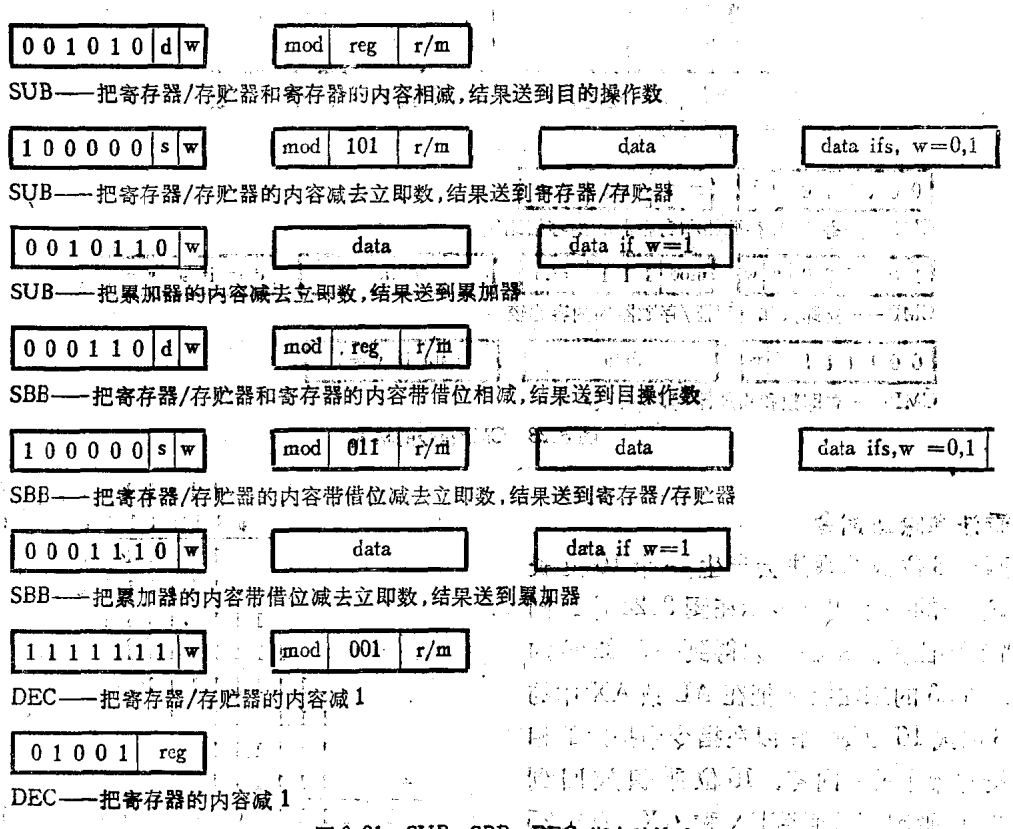


图 3.21 SUB, SBB, DEC 指令的格式

NEG 指令 (图 3.22) 改变它的操作数的符号。例如，若操作数原来是 −1 (1111 1111)，执行 NEG 指令后，操作数的内容将变成 +1 (0000 0001)。

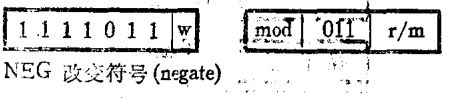


图 3.22 NEG 指令的格式

CMP 指令除了结果不放回目操作数外与减法指令相似。事实上结果并不存放到任何地方，它“消失”在处理器内部。读者可能会感到困惑：“丢失结果的指令有什么用呢?”实际上这条指令得到的是反映结果的某些性质的标志的设置，有时这比结果本身更重要。从这些标志的设置，可以决定参加减法的两个操作数之间的关系。例如，若 ZF 标志置位为“1”，则表示

结果为0，所以两个操作数必然相等。各种可能的关系的标志设置在表3.3中表示。CMP指令后面一般跟着一条条件跳转指令（在后面讨论），它测试设置的标志并判断是否满足跳转的条件。CMP指令的形式是和SUB指令的形式一样的，它的格式如图3.23所示。

表3.3 在一条CMP指令执行后标志的设置

目操作数和源操作数的关系		CF	ZF	SF	OF
相等		0	1	0	1
带符号操作数	小于	—	0	1	0
	小于	—	0	0	1
	大于	—	0	0	0
	大于	—	0	1	1
无符号操作数	低于	1	0	—	—
	高于	0	0	—	—

注表中没有说明的项，可以是"0"或是"1"，依照操作数的实际值而定。

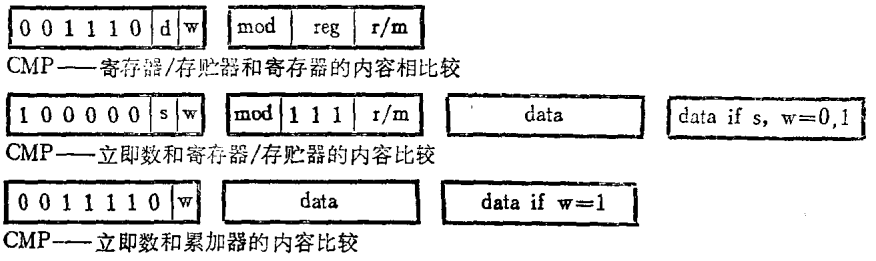


图3.23 CMP指令的格式

三、乘法和除法指令

两个8位数的乘法会产生一个16位长的积，这样的一个例子表示在图3.24中。同理，两个16位数的乘积可以得到一个32位的乘积。8086的乘法指令把在AL或AX中的一个8位或16位数，乘以在指令中指定的相同字长的操作数的内容。16位乘积放回到AX，32位乘积则放回到DX和AX。图3.25表示了这种情况。

```
    1 1 1 1 1 1 1 1    (8位)
  * 1 1 1 1 1 1 1 1    (8位)
    1 1 1 1 1 1 1 1
  1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
+ 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 1    (16位)
```

图3.24 乘积是操作数两倍长的一个例子

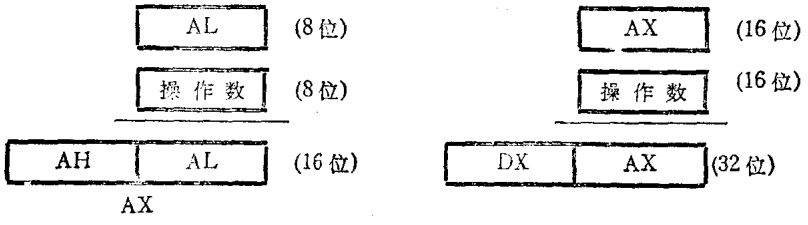


图3.25 乘法的源和目操作数

8086的除法运算，设计成是乘法指令的逆运算。除法指令用指令中指定的（具有被除数

一半字长)操作数,除在 AX 中的 16 位数(或在 AX 和 DX 中的 32 位数)。余数放入 AH (在较大情况下放入 DX),商放入 AL(在较大情况下放入 AX)。这一点在图 3.26 中说明。

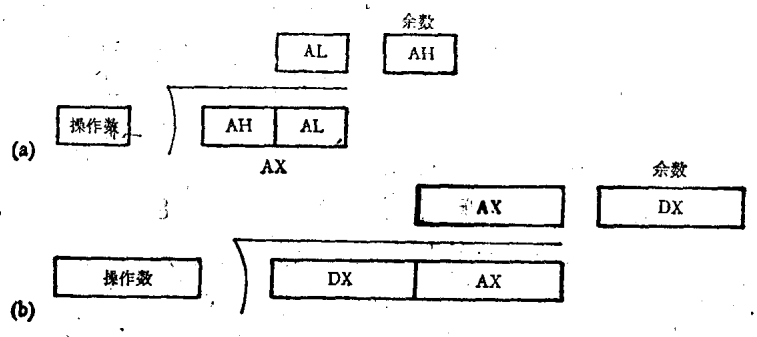


图 3.26 除法的源和目操作数: (a) 8 位除数; (b) 16 位除数.

演 示	翻译成无符号数	翻译成带符号数
1 1 1 1 1 1 1 1	255	−1
* 1 1 1 1 1 1 1 1	* 255	* −1
1 1 1 1 1 1 1 1		
1 1 1 1 1 1 1 1		
1 1 1 1 1 1 1 1		
1 1 1 1 1 1 1 1		
1 1 1 1 1 1 1 1		
1 1 1 1 1 1 1 1		
1 1 1 1 1 1 1 1		
1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 1	65,025	−511
	(正确结果)	(不正确结果)

图 3.27 演示普通 2 进制乘法对带符号数运算不能给出正确结果的例子

不象加法和减法,无符号数的 2 进制乘法和除法指令,对带符号数进行乘除运算不能给出正确结果,图 3.27 举例说明了这种情况。因此,对带符号数必须提供专用的乘法和除法指令。8086 的乘法和除法指令有 MUL (无符号乘法)、IMUL (带符号乘法,有时称为整数乘法)、DIV (无符号除法)和 IDIV (带符号除法,有时称为整数除法)。乘法和除法的指令格式在图 3.28 中表示。

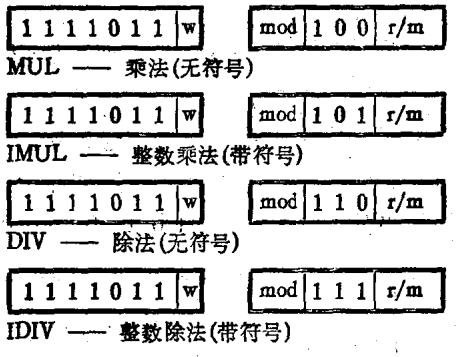


图 3.28 乘法和除法的指令格式

关于带符号除法还有一点要指出,即若把 −26 除以 +7 可以得到一个 −4 的商和一个 +2 的余数,也可以得到一个 −3 的商和一个 −5 的余数,两种结果都是正确的。所不同的只是在一种情况下余数是正的,而在另一种情况下余数是负的。8086 带符号除法指令是这样设计的,它规定余数的符号必须和被除数的符号相同。对于上述的除法,8086 产生一个 −3 的商和一个 −5 的余数。用这样定义的除法,−27 除以 +7,−27 除以 −7,+27 除以 +7 和 +27 除以 −7 将给出绝对值相同的商和余数。

表3.4总结了操作数的长度和各种算术指令的结果之间的关系。8086的指令被设计得使乘法的双倍长度的结果，可以直接在后面的除法中使用（参见图3.25和图3.26，乘法的结果所存放的寄存器正是除法用于存放被除数的寄存器）。但是，若在获得乘法的结果以后，不马上把它用于除法，而是应用于其他方面，则会发生什么情况呢？例如，要做17 (0001 0001)乘以10 (0000 1010)并在积上加上20 (0001 0100)的运算，该怎么做呢？对这个问题，只要忽略乘积的高8位就行了（因为此时高8位都是0，若不全是0，则不能这样做）。但若在不是前面的乘法产生的数上做除法时，就有问题了。例如，要把一个8位形式的35 (0010 0011)除以7 (0000 0111)。除法指令要求有一个16位的被除数在AX中，因而，只是把一个8位的被除数放入AL是不行的，因为除法指令将把任何在AH中的数值都当作被除数的高8位看待。所以8086要求在做8位除以8位的除法之前先把AH清0，在做16位除以16位的除法之前先把DX清0，才能保证除法指令的正确操作。

表3.4 操作数和结果长度的关系

指　令	第一操作数	第二操作数	结　果
加　法	8 (被加数) 16 (被加数)	8 (加数) 16 (加数)	8 (和) 16 (和)
减　法	8 (被减数) 16 (被减数)	8 (减数) 16 (减数)	8 (差) 16 (差)
乘　法	8 (被乘数) 16 (被乘数)	8 (乘数) 16 (乘数)	16 (积) 32 (积)
除　法	16 (被除数) 32 (被除数)	8 (除数) 16 (除数)	8 (商)，8 (余数) 16(商)，16(余数)

把双倍长度的被除数的高半部清0，对于无符号除法是很容易办到的，但对于带符号除法就比较麻烦了。例如，把−2的8位形式 (1111 1110) 转换成16位形式 (1111 1111 1111 1110)要把高8位全部置"1"，而把8位的+3(0000 0011)转换成16位的形式(0000 0000 0000 0011)却要把高8位全部置"0"。好在规则是简单的：只要把8位形式的最左一位(有时称为符号位)扩展到高8位去就行了。这种通过扩展符号位而把数的长度扩大的做法称为符号扩展。8086提供了执行符号扩展任务的指令(见图3.29)。这些指令原来命名为SEX (sign extend)，但后来重新命名为更加恰当的CBW (covert byte to word) 和CWD (convert word to double word)。CBW

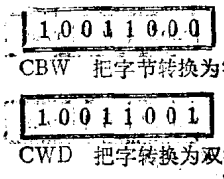


图3.29 符号扩展指令的格式

除　法	带　符　号	无　符　号
8位除以8位	把被除数送入AL，把AL的符号扩展到AH (CBW)，把AX的内容除以除数。	把被除数送入AL，把零放入AH，把AX的内容除以除数。
16位除以16位	把被除数送入AX，把AX的符号扩展到DX(CWD)，把DX、AX的内容除以除数。	把被除数送入AX，把零放入DX，把DX、AX的内容除以除数。

图3.30 等长除法的执行

指令把 AL 的符号位扩展到 AH 的所有位，CWD 指令把 AX 的符号位扩展到 DX 的所有位。图 3.30 总结了 8 位除以 8 位和 16 位除以 16 位的除法执行步骤。

四、10 进制运算

到目前为止，我们讨论的运算操作都是针对 2 进制数进行的。那是因为计算机是用 2 进制来处理的，但人们的习惯却是 10 进制的。为此，我们用计算机进行运算操作的时候，必须把我们习惯的语言转换成机器的语言，经过运算后再把结果转换回来。

计算机之所以用 2 进制，就是因为它只用两个电压级，"0" 和 "1" 来进行工作的，它不需要用 2 进制记号重新表达它们的数字。然而，对 10 进制数字却要用"0"和"1"来分别为它们进行编码。例如，可以用 2 进制等价形式 0010 0101 来表达 10 进制数字 37，也可以用一个 3 (0011) 的 2 进制编码跟着一个 7(0111) 的 2 进制编码来表示，结果，这种表达法是 00110111。注意，这是该 10 进制数的 2 进制编码，它被称为 2—10 进制数 (binary-coded decimal) 或 BCD。表 3.5 列出了每个 10进制数字的 2 进制编码。至于计算机为什么以 2 进制而不是以 BCD 来操作的原因是 2 进制表达法更加紧凑。例如，数 125 可以用 2 进制表达法在 8 位中表示 (0111 1101)，但 BCD 表达法却需要 12 位 (0001 0010 0101)。

表 3.5　10进制数字的BCD编码

数　字	编　码
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

读者可能马上会提出问题：用 BCD 表示的数该怎么运算呢？它可以做加、减、乘、除运算吗？答案是肯定的。实现 BCD 数运算的一种方法；是设置专门的 BCD 加法，BCD 减法，BCD 乘法和 BCD 除法指令，并使它们和 2 进制的加法，减法，乘法和除法指令并存在指令集中。另一种解决办法是对 BCD 数使用 2 进制算术指令，事先完全知道可能会得到错误的答案，然后执行一条能把答案转换成用 BCD 表示的正确答案的专用调整指令。8086 就采用后一种方法。

例如，使用 2 进制加法指令把 BCD 表示的 23 和 14 相加，加法实现的过程如下：

```
  0010 0011=23
+ 0001 0100=14
-------------
  0011 0111=37
```

巧得很，2 进制加法给出了正确的 BCD 结果！因此，在这个例子中不需要调整。让我们再碰碰运气，把 BCD 表示的 29 和 14 相加，这个加法的过程如下：

```
  0010 1001=29
+ 0001 0100=14
-------------
  0011 1101=3?
```

这个答案显然是不正确的，因为编码 1101 不代表任何一个 10 进制数字。出现这种情况的原因是一个 4位的 2 进制编码可以产生 16 种不同的码字，也就是说可以代表 16 个不同的数字。把它们用来代表 10 个不同的 10 进制数字，还多出了 6 个。这样，每当任意两个数相加之和大于 9 时就会进入或穿越这 6 个数字的禁区，从而给出不正确的答案。对这种不正确的答

案进行调整的方法是，每当出现进入或穿越禁区时就进行加 6 修正，以补偿必须传递进位的 6 个被禁止的数字．这样，前一个例子的和可被调整如下：

```
  0011 1101=3?
+      0110=06
-------------
  0100 0011=43
```

43 是正确的答案．在这个例子中，进入禁区的事实是很容易检测的，因为 1101 很明显不是一个 10 进制数字．但问题还没有完全了结，还有一种更微妙的完全穿越禁区的情况会发生．现由 BCD 表示的 29 和 18 相加的过程来说明：

```
  0010 1001=29
+ 0001 1000=18
-------------
  0100 0001=41
```

在这种情况下，结果是不正确的，因为最右数字的和完全穿越了禁区，这样的数字应该通过加 6 来调整．但是，只通过片面地观察相加后的该结果的 0001，是无法发现这种完全穿越禁区的情况的．这种情况只有通过分析才能发现．在结果完全穿越禁区的加法过程中，低位数字位将产生一个向高位数字位的进位，若有办法在每次加法进行后，对是否产生这种进位进行检测的话，就能控制对结果的加 6 调整了．进位标志 CF 在前面已经讨论过．它指出加法何时在最高位产生一个进位（从而是最高位数字产生的进位）．辅助进位标志 AF 的存在，则完全是为了指出加法何时在低 4 位产生了一个向高 4 位的进位（从而是低位数字向高位数字产生的进位）．由此可见，可以用 AF 标志来控制 BCD 调整．在上述例子中，CF 被设置为 0，而 AF 被设置为 1，因而可以得出要进行加 6 调整的结论．

多精度运算也可以在 BCD 数上实现．我们用 BCD 数 2889 和 3714 相加的实例来说明．这个加法要用到如下所示的 2 次连续的加法和调整：

1．数字的最低对相加：

```
  1000 1001=89
+ 0001 0100=14
-------------
  1001 1101=9?     CF=0  AF=0
```

2．执行调整：

```
  1001 1101=9?
+      0110=调整
-------------
  1010 0011=?3
+ 0110     =调整
-------------
  0000 0011=03       CF=1  AF=1
```

3．数字的最高对和上次加法的 CF 值一起相加：

```
       1          (上次 CF)
  0010 1000=28
+ 0011 0111=37
-------------
  0110 0000=60     CF=0  AF=1
```

4. 执行调整：

$$\begin{array}{r} 0110\ 0000 = 60 \\ +\quad\ 0110 = \text{调整} \\ \hline 0110\ 0110 = 66 \end{array}$$

5. 最终结果：

$$0110\ 0110\ 0000\ 0011 = 6603$$

执行 10 进制调整的 8086 指令是 DAA (decimal adjust for addition)，这时操作数已在 AL 中。基于在 AL 中的值以及 CF 和 AF 的设置情况，DAA 指令决定是否对 AL 的值进行调整。一条类似的指令 DAS (decimal adjust for subtraction)，将在减操作以后进行调整。对乘法结果不能进行调整，因为 BCD 结果无法从产生的交叉项中区别出来。同理，除法调整也是不可能的。因此，如果要执行 10 进制数的乘法或除法，必须使用如下所述的另一种 10 进制数表达法。

我们把上面讨论的 BCD 数，称为压缩的 BCD 数，因为在一个字节中装入了两个这种数字。另一种 10 进制数的表达法称为非压缩的 BCD 数，这种表达法在一个字节中只放一个数字。这个数字放在该字节的最低 4 位而高 4 位的值不影响所表示的数字。数字的 ASCII 表达法是非压缩 BCD 数的一个例子。ASCII 是一集字符的 7 位表达法（见附录 C）。数字的 ASCII 表达法在表 3.6 中表示，高 4 位是 0011，它与数字值无关。

表 3.6 数字的 ASCII 表达法

数 字	ASCII
0	0011 0000
1	0011 0001
2	0011 0010
3	0011 0011
4	0011 0100
5	0011 0101
6	0011 0110
7	0011 0111
8	0011 1000
9	0011 1001

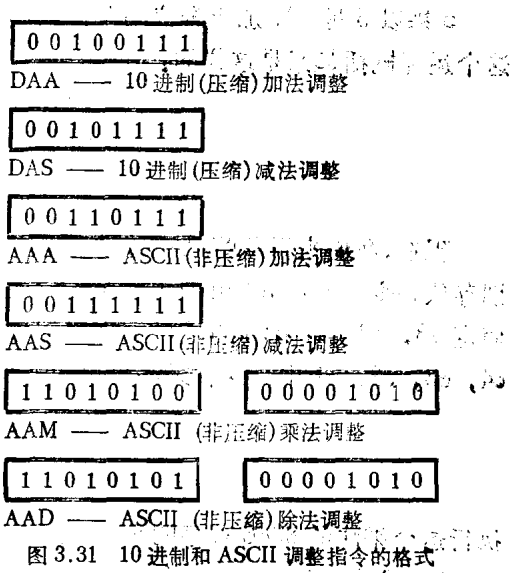


图 3.31 10 进制和 ASCII 调整指令的格式

非压缩 BCD 数的加法和减法除了只影响最低位数字外，可以用类似于压缩 BCD 数调整的方式进行调整。与压缩 BCD 数不一样，对于乘法和除法，非压缩 BCD 数是可以调整的。执行这 4 种调整的指令，称为 ASCII 调整指令（因为 ASCII 是非压缩 BCD 的最通俗的例子），AAA (ASCII adjust for addition) 是加法的 ASCII 调整指令，AAS (ASCII adjust for subtraction) 是减法的 ASCII 调整指令，AAM (ASCII adjust for multiplication) 是乘法的 ASCII 调整指令，AAD (ASCII adjust for division) 是除法的 ASCII 调整指令。10 进制和 ASCII 调整指令（压缩 BCD 和非压缩 BCD）的格式如图 3.31 所示。

考虑一个把非压缩 BCD 数 4 乘以 9 的例子。假定非压缩的 9 (0000 1001) 在 BL 寄存器中并且非压缩的 4 (0000 0100) 在 AL 寄存器中，执行把 BL 作为源（乘数）的无符号 2 进制乘

法指令将把16位的2进制乘积36 (0000 0000 0010 0100) 放入AX寄存器。乘法调整指令AAM必须把AX中的2进制36分解成3 (0000 0011) 和6 (0000 0110)，并分别把它们放入AH和AL寄存器。要实现这一功能只要把AL的内容除以10，把商放在AH中，把余数放在AL中就可以了。实际上，AAM指令是两字节长决不是巧合(看起来似乎一个字节就够了)，它的第二个字节不是别的，就是在执行调整时要作为除数的10 (0000 1010) 的2进制码。由此可见，AAM指令是一种把除数作为立即数放在指令的第二个字节的除法指令(尽管它不把商和余数放到DIV和IDIV所放的地方)。从这里可以推出，若把16 (0001 0000) 放在第二个字节中，执行这条指令就会把在AL中的一个压缩BCD数，转换成一个在AH和AL中的非压缩BCD数。

从上面举的例子中可以看到，操作数0000 1001和0000 0100是最高4位都是0的非压缩BCD数。若不是这样，乘法所产生的交叉项会掩盖所需要的结果0010 0100 (正是这种交叉项才使压缩BCD数的乘法不能进行调整)。显然，在执行非压缩BCD数乘法之前，必须把每个操作数的高4位清0，除非知道它们已经是0。可以把一个字节中选定的位清0的指令是AND指令(在后面讨论)。

至此，我们已经看到了把两个非压缩BCD数相乘的全过程。现在让我们试着把一个多位数乘以一个一位数，例如539乘以6。在刚学算术的时候，我们会这样来执行这个乘法：

9乘以6是54。记下4并进位5。3乘以6是18，加上进位的5是23。记下3并进位2。

5乘以6是30，加上进位的2得到32。最后把它们记下。

这个运算概括起来是这样的：

```
   25  (进位)
  539
*   6
------
 3234
```

现在，我们来看8086是如何着手解决这个问题的。假定数539被作为非压缩BCD数，分别存放在变量a3，a2和a1中，同时假定数6被作为非压缩BCD数存放在变量b中。另外，假定a3，a2，a1和b的最高4位都是0。我们要把a3，a2，a1乘以b并把结果放到变量c4，c3，c2，c1中。这种乘法可用图解表示如下：

```
        a3  a2  a1
*                b
-------------------
  c4  c3  c2  c1
```

执行这个乘法的8086程序的步骤是这样的：

```
1. a1*b→AX      ;9乘以6是…
2. AAM          ;54 (5在AH中，4在AL中)
3. AL→c1        ;记下4
4. AH→c2        ;进位5
5. a2*b→AX      ;3乘以6是…
6. AAM          ;18 (1在AH中，8在AL中)
7. AL+c2→AL     ;加上进位的5变成…
8. AAA          ;23 (2在AH中，3在AL中)
9. AL→c2        ;记下3
```

```
10. AH→c3        ;进位 2
11. a3 * b→AX    ;5 乘以 6 是…
12. AAM          ;30 (3 在 AH 中,0 在 AL 中)
13. AL+c3→AL     ;加上进位 2 变成…
14. AAA          ;32
15. AL→c3        ;记下 2
16. AH→c4        ;记下 3
```

让我们详细地检查上述例子中的一条和加法相对应的 AAA 加法调整指令。在第 8 行中，当 AAA指令把 AL 的内容从不正确的 (0000 1101) 调整到 3 (0000 0011) 时，从 AL 的最低位数字产生了一个进位。该进位并不进到 AL 的高位数字，而是进到了 AH 的低位数字，从而把 AH 从 1 (0000 0001) 调整为 2 (0000 0010)。这样，AAA 指令实际上是执行了对 AH 和 AL 的调整。若 AAA 仅用于加法而不是乘法，则 AAA 的这个副作用不是必需的 (为什么？请读者考虑)。

一个更优化的多数字非压缩 BCD 乘法的算法(包含一个循环)在图 3.32 中叙述。虽然这里只讨论了单个数字的乘数，但扩展到多数字的乘数是简单的。

加　　法

被加数：　a(n) a(n−1) a(n−2)…a(3) a(2) a(1)
加　数：　b(n) b(n−1) b(n−2)…b(3) b(2) b(1)

和　：c(n+1) c(n) c(n−1) c(n−2)…c(3) c(2) c(1)

清除进位标志 CF
对于 i 从 1 到 n 的每一个整数值把下列步骤做一次：
　把 a(i) 送入 AL
　把 b(i) 带进位加到 AL 上
　把 AL 加调整为 AH，AL
　把 AL 送入 c(i)
把 AH 送入 c(n+1)

乘　　法

被乘数：　a(n) a(n−1) a(n−2)…a(3) a(2) a(1)
乘　数：　b

积：　c(n+1) c(n) c(n−1) c(n−2)…c(3) c(2) c(1)

把 b 的高 4 位清 0
把 c(1) 清 0
对于 i 从 1 到 n 的每一个整数值把下列步骤做一次：
　把 a(i) 的高 4 位清0；把结果送入 AL
　把 AL 乘以 b
　把 AL 乘调整为 AH，AL
　把 c(i) 加到 AL 上
　把 AL 加调整为 AH，AL
　把 AL 送入c(i)
　把 AH 送入 c(i+1)

减　　法

被减数：　a(n) a(n−1) a(n−2)…a(3) a(2) a(1)
减　数：　b(n) b(n−1) b(n−2)…b(3) b(2) b(1)

差　：c(n+1) c(n) c(n−1) c(n−2)…c(3) c(2) c(1)

清除进位标志 CF
对于 i 从 1 到 n 的每一个整数值把下列步骤做一次：
　把 a(i) 送入 AL
　从 AL 中带借位减去 b(i)
　把 AL 减调整为 AH，AL
　把 AL 送入 c(i)
把 AH 送入 c(n+1)

除　　法

被除数：　a(n) a(n−1) a(n−2)…a(3) a(2) a(1)
除　数：　b

商：　c(n) c(n−1) c(n−2)…c(3) c(2) c(1)

把 b 的高 4 位清 0
把 AH 清 0
对于 i 从 1 到 n 的每一个整数值把下列步骤做一次：
　把 a(i) 的高 4 位清 0；把结果送入 AL
　把 AH，AL 除调整为AL
　把 AL 除以 b，把余数送入 AH
　把 AL 送入 c(i)

图 3.32　多数字非压缩 BCD 运算

下面考虑一个非压缩 BCD 数的除法，例如 42 除以 6。假定非压缩的 42 是在 AX 中

(0000 0100 在 AH 中，0000 0010 在 AL 中)并且非压缩的 6 是在 BL 中。一个数字的数，如 6 的非压缩表达法不是别的，就是它的 2 进制表达法 (0000 0110)。因此，在这里只要把被除数变成 2 进制的表达法。这可以用把 AH 的内容乘以 10 并把它的内容加到 AL 的内容上去的方法来实现。于是，一个 AL (2 进制 42) 除以 BL (2 进制 6) 的除法，最终将在 AL 中给出 2 进制的 7。由于 2 进制的 7 就是非压缩的 7，所以非压缩数的除法也就完成了。

在上述例子中有三点是要注意的。首先，除法调整 (AAD) 由把 AH 的内容乘以 10 并把结果加到 AL 中的内容上所组成。其次，除法调整在除法操作之前进行，而加法，减法和乘法的调整则在相应的运算操作之后进行。换一句话说，加法，减法，乘法调整是纠正一个错误的结果(即非 BCD 数)，而除法调整却是预防出现一个错误的结果。最后，非压缩 BCD 除法的被除数、除数在高 4 位必须都是 0。这一点对于乘法来说也是必须的，而对于加法和减法却不是必须的(为什么？请读者考虑)。

多数字的被除数，除以单数字的除数的除法，可以用已经举例说明的乘法的同样方法来进行，它的算法在图 3.32 中表示。但是这种方法不能应用于多数字除数的除法。多数字除数的除法，可以用"猜"商的办法，并用非压缩 BCD 数的乘法和减法来检查这种猜测接近的程度，然后逐步求精这种猜测。实际上，这就是我们平时做长除法的笔算方法。Donald E. Knuth 在他写的 The Art of Computer Programming-Volume 2 中对这种长除法进行了详尽的讨论。

§3.3 逻 辑 指 令

8086 逻辑指令由布尔指令和移位/环移指令组成，它们被总结在表 3.7 中。

表 3.7 逻 辑 指 令

AND:	目 and 源 → 目	
TEST:	目 and 源 → ?	
OR:	目 or 源 → 目	
XOR:	目 xor 源 → 目	
NOT:	not 目 → 目	
SHL (shift logical left):	逻辑左移	CF ← 目 ← 0
SHR (shift logical right):	逻辑右移	0 → 目 → CF
SAL (shift arithmetic left):	算术左移	同SHL
SAR (shift arithmetic right):	算术右移	符号→ 目→ CF
ROL (rotate left):	左环移	
ROR (rotate right):	右环移	
RCL (rotate left through carry):	带进位位左环移	
RCR (rotate right through carry):	带进位位右环移	

一、布尔指令

布尔指令有 NOT、AND、OR (非，与，或) XOR (异或) 和 TEST。这些指令的格式表示在图 3.33 中。

AND、OR 和 XOR 指令在源操作数的每一位和目操作数的各对应位之间执行一个逻辑功能，并把结果放回到目操作数的对应位。NOT 指令只有一个操作数，它在该操作数的每一

位上执行它的功能，并把结果放回到相应位。这些指令所执行的逻辑功能在表 3.8 中定义。

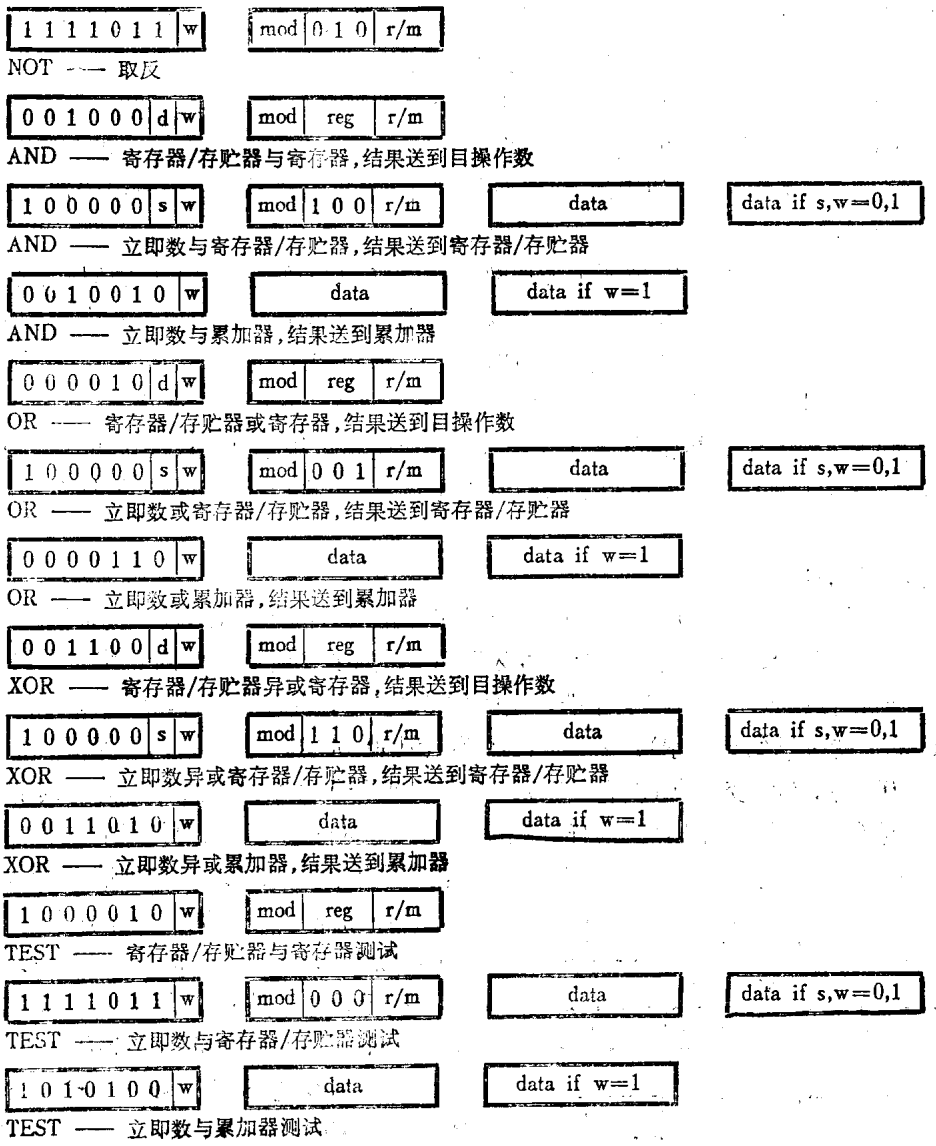


图 3.33 布尔指令的格式

表 3.8 逻辑功能的定义

	源位		非		
单操作数	0		1		
	1		0		
	源位	目位	与	或	异或
双操作数	0	0	0	0	0
	0	1	0	1	1
	1	0	0	1	1
	1	1	1	1	0

“与”功能对于把一个数中的指定位清 0 (有时称为屏蔽)是很有用的。在指令中，一个操作数指定位的位置，而另一个操作数则指定那个某些位要清 0 的数。例如，可以把一个 8 位的数和 0000 1111进行“与”操作而把该数的高 4 位清 0 (在执行 10 进制乘法或除法之前必须把一个非压缩的 10 进制数的高 4 位清 0，此时就要用到这条指令)。

类似地，“或”功能和“异或”功能在需要把一个数的指定位置位和取补时亦很有用。例如，可以把 1000 0000 和一个 8 位的数进行“或”操作而使该数的最高位置位，也可以 0011 1100 和一个 8 位数作“异或”操作而使中间 4 位取反。“非”功能在需要把一个数中的每一位取反时是很有用的，它等价于把该数进行取“反码”操作。

TEST 指令和 AND 指令在把两个操作数的对应位相“与”这一点上是类似的。不同的是，TEST 指令只保持设置的标志位而不保留结果。该指令可用来检查在一个数中的指定位上是“1”还是“0”，也可用来测定某个数中是否有“1”存在。例如，要决定 BL 的低 4 位中是否有一位是 1，可把 0000 1111 送入 BH 执行把 BH 和BL指定为操作数的 TEST 指令，然后执行一条若 ZF 是0 就跳转的条件跳转指令。注意，AND 指令也可以用在 TEST 指令的地方来设置标志位，但由于 AND 指令要把操作结果送回目操作数，所以执行 AND 指令最终会破坏目操作数的初值。

二、移位/环移指令

移位指令对把一个数乘以 2，或除以 2 提供了非常有效的手段(比做一次乘法或除法所需要的字节和时钟周期都少)。把一个无符号数乘以 2，只要把所有的位向左移动一位，并用 0 填补空出的最低位就行了。若把从左端移出的位放入 CF，则通过测试，就可以决定结果是否溢出 (CF 是 1 就表示结果溢出)。例如，把数 65 (0100 0001) 左移一位，即乘 2 以后，可得到 130 (1000 0010)，CF 变成 0 (没有溢出)。然而若把 130 再左移一位时，就得到了 4 (0000 0100) 而此时 CF 变成 1 (溢出)。同样，把一个无符号数除以 2，只要把所有的位右移一位并用 0 填补空出的最高位，再把从右端移出的位放入 CF 就可以了。在这种情况下，CF=1 表示移位前该数是奇数。例如，把数 9 (0000 1001) 除以 2 结果得 4 而 CF 变成 1。

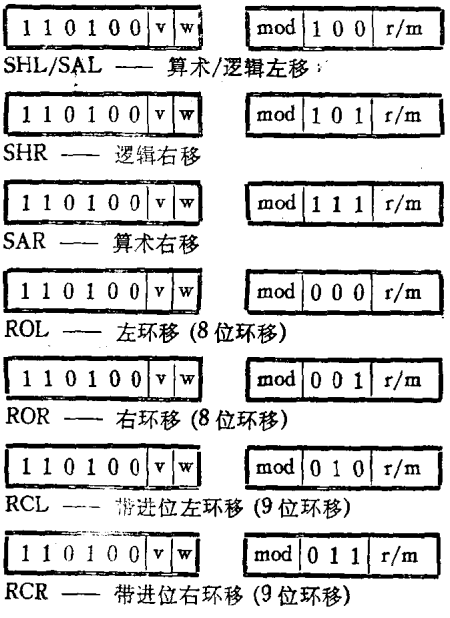


图 3.34 移位/环移指令的格式

执行无符号数乘以 2 和除以 2 的指令是 SHL (shift left) 和 SHR (shift right)。其他两条移位指令是 SAL (shift arithmetic left) 和 SAR (shift arithmetic right)，它们对于带符号数乘以 2 和除以 2 是很有用的。这些指令和环移指令的格式一起表示在图 3.34 中。

在把一个带符号数除以 2 的 SAR 和把一个无符号数除以 2 的 SHR 之间的区别，在于前者的最左位(符号位)必须保持不变。例如把 +6 除以 2，应该得到 +3 (0000 0011)，而把 −120 (1000 1000) 除以 2 应该得到 −60 (1100 0100)。显然，SAR 指令的功能应是把所有的位右移一位同时使符号位保持不变。

把一个带符号数乘以 2 和把一个无符号数

乘以 2 是没有区别的，因此，SHL 和 SAL 实际上只是同一条指令的两个不同的名字。

环移指令，可以用来把一个数中的位进行重新排列。ROL (rotate left) 和 ROR (rotate right) 能把一个数中的所有位进行左环移或右环移，即把从一端移出的位，环移填补到另一端空出的位上去。另外两条环移位指令 RCL (rotate with carry left)，和 RCR (rotate with carry right)，分别是带进位标志 CF 的左环移和右环移。它们把进位标志 CF 和要环移的数串在一起进行操作，即从一端移出的位顺序进入 CF 标志位，而在 CF 标志位中的位，则被环移入到另一端空着的位。

移位或环移指令的操作数，可以在存贮器中也可以在寄存器中(由指令的 mod，r/m 字段指定)；操作数可以是 8 位的，也可以是 16 位的(由 w 字段指定)。还有一个字段 v，它指定所要移位或环移的位数是一位 (v=0) 还是任意位 (v=1)。在后一种情况下，移动位数由 CL (COUNT 寄存器)的内容决定。

很明显，v 字段的设置，提供了有效的多位移位的功能 (但必须了解，要做一个 2 位的移位时，执行两个 v=0 的 1 位移位要比把一个 2 先装入 CL，然后再做一个 v=1 的移位更有效)。然而，设置 v 字段的本意却是为了可以进行可变位的移位和环移。正是由于这个原因，该字段才称为 v (variable)。可变移位指令，是在要把前面计算结果作为移位位数时使用的。图 3.35 表示了一个可变移位的例子。

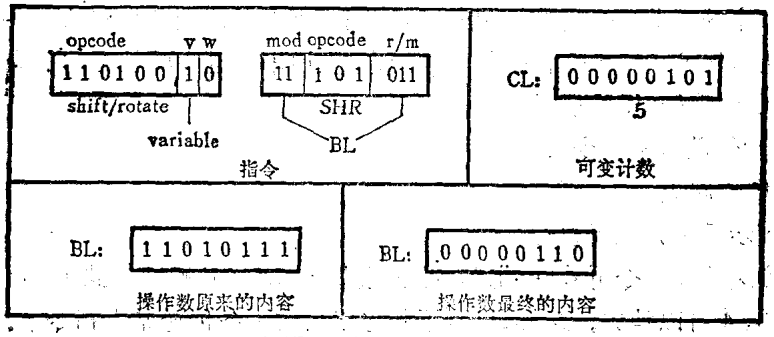


图 3.35 可变移位的例子

§3.4 串 指 令

一个串，就是在存贮器中的一串字节或字的序列。所谓串操作就是对串中的每一项都执行的操作。例如，串传送把一个项目串，从存贮器的一个区域，传送到另一个区域。由于串操作常常是重复进行的，因而它们的执行往往要占用比较长的时间。8086 有一集能减少执行串操作所需要时间的专门指令。能实现"加速"执行的原因主要是：

(1) 有一个强有力的基本指令集，它使处理串中每一项的时间得以减少。

(2) 消除了通常在连续项目处理之间执行的记帐和额外开销。

基本串指令总结在表 3.9 中。

表 3.9 基本串指令

MOVS	传 送	源 → 目	更新 SI, DI
CMPS	比 较	源 − 目 → ?	更新 SI, DI
SCAS	扫 描	AL − 目 → ?	更新 DI
LODS	装 入	源 → AL	更新 SI
STOS	存 放	AL → 目	更新 DI

一、基本串指令

先让我们通过一个传送字节序列的例子，来说明串指令是如何加速串处理的。要实现一个字节序列的传送，必须用指示器指示从存贮器的哪一个单元传送到哪一个单元。寄存器 SI(SOURCE INDEX) 和 DI(DESTINATION INDEX)可以用于这个目的，我们可以在执行传送前，把在当前数据段中要传送序列的第一个字节的偏移量送到 SI 中，而把该字节的传送目的单元的偏移量送入 DI。存放要被传送的字节个数的合适的地方，是 CX 计数寄存器。若CX 的初值是 0，则就不再传送字节。这样，执行一个串传送的条件已经具备，它执行的步骤如下：

1. 若 CX 等于零，则已经执行完。
2. 取偏移量在 SI 中的字节。
3. 把该字节存放到偏移量在 DI 中的单元里去。
4. 把 SI 加 1。
5. 把 DI 加 1。
6. 把 CX 减 1。
7. 回到步骤 1 并重复执行。

步骤 2 和步骤 3，执行每个字节的实际传送。步骤 4 到 6 是记帐，步骤 1 和 7 是额外开销。每个字节的实际传送，可以用一条一字节指令来加速，它把偏移量在 SI 中的字节传送到偏移量在 DI 中的字节单元里去。另外，若该基本指令还能把 SI 和 DI 各加 1，则部分直接记帐工作也就消除了。用了这样一条基本指令，上述串传送步骤就可简化成如下：

1. 若 CX 等于 0，则已经执行完。
2. 执行“传送基本指令”。
3. 把 CX 减 1。
4. 回到步骤 1 并重复执行。

如果该传送基本指令再被“强化”，从而把基于 CX 的“测试—减 1—重复”进行合并，则上述步骤 1，3，4 可以消去。这样，只剩下强化以后的一条传送基本指令了。它的执行步骤现在变成如下所示：

1. “强化”后随的基本指令。

1a. 执行强化后的传送基本指令。

8086 有一条 MOVS (move string element) 指令，它就是上面描述的串传送基本指令。

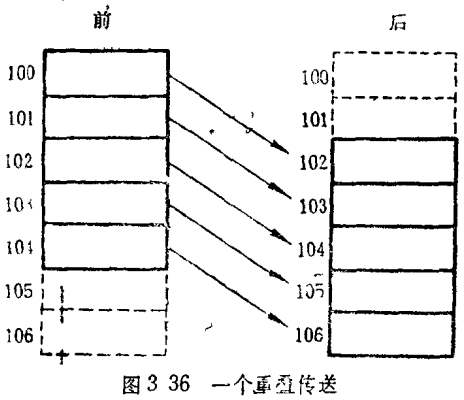


图 3 36 一个重叠传送

另外，任何串传送基本指令，都可以在它前面放一个称为重复前缀的一字节前缀来加以“强化”。这样，重复前缀和 MOVS 基本指令就形成了一条两字节指令。

在进行上述字节传送时，若目地址区域和源地址区域相重叠，就会无法正确进行。例如，要把开始于偏移量 100 的 5 个字节，传送到开始于偏移量为 102 的 5 个字节单元中去，如图 3.36 所示。在把偏移量为 100 和 101 的字节连续拷贝到偏移量为 102 和 103 的字节单元中以后，再要把偏移量为 102 的字节单元的内容，拷贝到偏移量为 104 的单元时，就发生了在 102 的字节已不

是原先在那里的字节，而是来自 100 的字节的问题。从偏移地址 100 来的字节从偏移地址 102 那里再次被拷贝到偏移地址为 104 的单元，并最终还会拷贝到偏移地址为 106 的单元。同理，来自偏移地址 101 的字节将会拷贝到偏移地址为 103 和 105 的字节单元。

如果把上述问题中的 5 个字节，以相反方向传送，重复拷贝的问题就不会发生。来自偏移地址 104 的字节将首先被传送，然后传送来自偏移地址为 103 的字节，等等。但是，如果重叠是在相反的方向，例如 100 到 104 重叠于 98 到 102，则反向传送也会发生重复拷贝的问题，而此时正向传送却是恰当的。

重复拷贝的问题在执行串传送时需要加以注意，否则非但达不到把源串传送到目串地址单元中去的目的，还会破坏源串的初始内容。但事情总是一分为二的，当需要在存贮器的某一部分重复同一字节的传送时，我们可以对出现重复拷贝的现象加以利用，使 MOVS 指令在一片存贮器区域完成赋某一个值的任务。

8086 有一个称为 DF (direction flag) 的标志，它决定着被处理串的处理方向。若 DF=0，串处理就从在 SI 和 DI 中的偏移地址开始向前进行(向较高地址方向发展)。若 DF=1，则串处理就从在 SI 和 DI 中的偏移地址开始向后进行(向较低的地址方向发展)，并把 SI 和 DI 减 1 而不是加 1。这样，若一个重叠传送要把字节串向较高偏移量方向传送(所以就必须要一个反向传送)，DF 就应该初始化为 1。依据设置的 DF 值，SI 和 DI 将含有串的最低偏移地址 (DF=0) 或最高偏移地址 (DF=1)。设置和清除 DF 的指令 (STD, CLD) 将在后面的标志指令中讨论。

为了方便从一个段到另一个段的串传送，SI 和 DI 含有进入不同段的偏移量将是适宜的。我们规定 SI 含有进入当前数据段的偏移量，但我们还没有说明在 DI 中的偏移量引用的是哪一个段，如果串基本指令被设计得使 DI 含有进入当前附加段的偏移量那就最好了。事实上正是这样。现在，要把一个串从一个段传送到另一段，可以由用恰当的段起始地址装入 DS 和 ES 开始，并且使 SI 和 DI 含有在各自段中的恰当的偏移量。显然，在一个段内的串传送，是用相同的值装入 DS 和 ES 来实现的。

某些串操作按字执行，比之按字节执行更有效。例如，若被传送的元素是字，传送就可以进行得更快。为了使串操作能按字进行，每一条串基本指令都含有一个区别字节操作 (w=0) 和字操作 (w=1) 的w字段。对于字的传送基本指令，除了把 SI 和 DI 加 (若 DF=1 是减) 2 而不是 1 以外，和字节传送基本指令是一样的。CX 在每次操作之后总是减 1，由此可知，若使用字基本指令，则必须用串中所含的字数来初始化 CX 寄存器。

现在，让我们考虑另一种串操作。即扫描一个字节序列，以找出一个特殊值的操作。例如，在一个 ASCII 字符串中，寻找第一次出现的某一个字符，就可以使用这条指令。我们再一次把序列的起始偏移地址放在 DI 中，用 CX 存放该序列中的字节数，把要搜索的特定字节放入 AL。执行扫描的步骤如下所示：

1. 若 CX 等于 0，则已经执行完。
2. 取偏移量在 DI 中的字节。
3. 把它和 AL 中的字节比较(比较意味着相减并设置标志，特别是 ZF)。
4. 把 DI 加(若 DF=1，则为减) 1。
5. 把 CX 减 1。
6. 若 ZF=0，则两个字节不相等，回到步骤 1 并重复。

步骤2，3，4是由8086的串扫描基本指令 SCAS (scan string element) 来完成.若该扫描基本指令用重复前缀"强化"，则步骤1，5，6就会由强化后的 SCAS 指令自动执行. 字扫描(w字段=1) 除了用 AX 代替 AL 并且 DI 加(减) 2 而不是加(减) 1 以外，同字节扫描是一样的.

注意，扫描基本指令的重复前缀的特性，与传送基本指令的重复前缀稍有不同. 对于前者，在决定是否重复之前，要测试 ZF 标志. 在一般情况下，若重复前缀所伴随的串基本指令是可能修改 ZF 标志的，重复前缀就要测试 ZF 标志. 例如，MOVS 不影响 ZF 标志，而 SCAS 则依据扫描到的元素与要查找的元素是否匹配而设置或清除 ZF.

另一个串操作，是扫描整个字节序列，以寻找一个不等于某个特定字节的元素. 这种操作的一个例子，是在一个表中寻找第一个非 0 项. 这是用同上述扫描操作一样的办法，在扫描基本指令上加用重复前缀来实现的，所不同的是，现在关于重复的条件是 ZF=1. 由于对 ZF 值的测试，是由重复前缀决定的，所以该前缀必须指明 ZF 的那一个值会引起重复. 这是由在重复前缀中的一个 1 位 z 字段来指定的. 当重复前缀与诸如 MOVS 等不影响 ZF 标志的串基本指令一起使用时，该 z 字段被忽略. 重复前缀和串基本指令的格式在图 3.37 中表示.

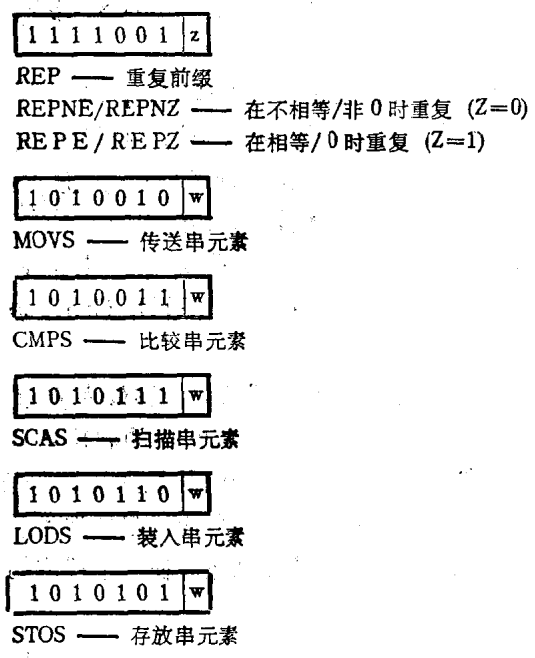


图 3.37 重复前缀和串基本指令的格式

下面要介绍的一个串操作，是对两个字节序列进行比较，看哪一个字节序列应该排在前面. 例如，对于两个ASCII 代码字符串，这个操作把它们按字典序放置 (字典序是字母序概念的扩展，它把非字母字符也同样计入). 我们再次把两个序列的偏移量，放在SI和 DI 中，并把要比较的字节数(较短字节序列的长度)放在 CX 中. 执行串比较的步骤如下：

1. 若 CX 等于 0，则已经执行完.
2. 取出偏移量在 SI 中的字节.
3. 把它与偏移量在 DI 中的字节进行比较.
4. 把 SI 加(若 DF=1，则减) 1.
5. 把 DI 加(若 DF=1，则减) 1.
6. 把 CX 减 1.
7. 若 ZF=1，则这两个字节序列到目前为止相等，回到步骤 1 并重复执行.

步骤2，3，4，5是由 8086 的串比较基本指令 CMPS (compare string elements) 来执行的，若把一个重复前缀 (此时 z 字段中 z=1) 附加于 CMPS 指令，则剩余的步骤也能自动地完成. 这里还要作一些解释. 在第 3 步中，只要被比较的字节是相等的，0 标志 (ZF) 将被置位为 1，并且第 7 步在 CX 不等于 0 时将继续返回到第 1 步. 当两个字节不相等 (步骤 7 将不再返回)或已经到达较短串的末尾时，循环将会终止. 在循环终止后，可以通过测试 ZF 来决

定是否到达了较短串的末尾(在这种情况下 ZF 将仍为 1)。若还没有到达较短串的末尾，可以通过测试进位标志 (CF) 来决定哪一个串较大 (CF=1 表示由 DI 指示着的串较大)。上述关于 CMPS 的执行情况，可以通过两个例子直观地加以说明。先用 CMPS 指令比较源目串分别为 ab 12 和 ab 2 cd 的字符序列，较短的字符串长度 4 将被装入 CX，重复执行到第三次时，发现 ZF≠1，于是循环终止，通过测试 CF，发现 CF=1，因而可以决定目串 ab 2 cd 比较大。再用 CMPS 指令比较源目串分别为 abcd 和 abcd 1234 的字符序列，同样较短的字符串长度 4 将被装入 CX，重复执行 4 次以后，两个字符串仍然相等，但 CX 已等于 0，于是循环终止。通过测试，发现 ZF=1，因而可以决定目串 abcd 1234 比较大。这两个例子中的字符串按字典序应该排列成 ab 12、ab 2 cd 和 abcd、abcd 1234。

最后两条串基本指令是 LODS (load string element) 和 STOS (store string element)。它们分别称为装入基本指令和存放基本指令。装入基本指令把偏移量在 SI 中的字节或字装入 AL 或 AX 并把 SI 加(若 DF=1，是减) 1 或 2。存放基本指令把 AL 或 AX 中的字节或字存放到偏移量在 DI 中的字节或字单元中并把 DI 加 (若 DF=1，是减) 1 或 2。同前面介绍的基本指令不一样，这两条基本指令一般不单独和重复前缀一起使用，它们主要用于和其他的指令一起建立更复杂的串操作。尽管如此，串存放基本指令 STOS，在和重复前缀连接起来使用时，却可以执行一种有用的功能，它可以用相同的值，填入序列的每一个字节或字 (用执行重叠串传送也可以完成这种任务，只是效率稍低些，因为需要两个串而不是一个串)。然而，在串装入基本指令 LODS 前放一个重复前缀却没有任何用处，因为它用在一个序列中的字节或字连续地顺序装入 AL 或 AX，后一次破坏了前一次装入的值，最后得到的就是该串的最后一个字节或字。

二、复合串指令

5 条串基本指令，提供了最常用的串操作。为所有可以想象的串操作，都提供一条基本指令，是一种不切实际的想法，较好的策略是提供一种建立有效的复杂串指令的手段，即把一些串基本指令，作为构筑这种复杂串指令的基本模块。让我们考虑把一个字节序列取补的操作的例子。在这里，每一个字节表示一个 8 位的带符号数，令 SI 含有该序列的第一个字节的偏移量；DI 含有存放已取补字节序列的第一个字节单元的偏移量；CX 含有该序列中的字节数。执行这一操作的步骤可以表示如下：

→ 1. 若 CX 等于 0，则已经执行完，跳过下列所有步骤。
2. 取出偏移量在 SI 中的字节。
3. 把 SI 加 1。
4. 把取得的字节取补。
5. 把结果存放到偏移量在 DI 中的字节单元。
6. 把 DI 加 1。
7. 把 CX 减 1。
— 8. 回到步骤 1 并重复执行。

同前面的例子一样，最好有一条执行步骤 2，3，4，5，6 的基本指令，但是，8086 没有这样的指令。因此，下一步最好的办法是用 8086 的指令来构造这些步骤。若一些模块使用串基本指令，则 SI 和 DI 加 (减) 1 (2) 就不要付出额外的代价了。显然，步骤 2 和 3 可以用装入基

本指令，步骤4可以用取补指令，步骤5和6可以用存放基本指令，这样就可把任务简化为：

1. 若 CX 等于0，则已经执行完，跳过下列所有步骤。
2. 执行“装入基本指令”。
3. 把 AL 中的字节取补。
4. 执行“存放基本指令”。
5. 把 CX 减1。
6. 回到步骤1并重复执行。

在这里，步骤1，5，6以前是用重复前缀来实现的。现在，由于循环的主体，由多条指令（包括串基本指令）所组成，重复前缀就不能使用了，我们希望有一些能模拟重复前缀操作的指令来完成这些操作。步骤1需要一条条件跳转指令，它在 CX 等于0时就跳转，跳转的目的应当在尽可能少的位中指定。8086的 JCXZ 正是这样的一条指令，它在 CX 等于0时就跳转，跳转的目的，由在该指令中的一个字节里所存放的目偏移量和 JCXZ 指令末尾的偏移量之间的差（一个带符号数）来指定的。我们进一步的希望是要一条能把 CX 减1，然后在 CX≠0 时跳转的指令。8086的 LOOP 就是这样一条指令，LOOP 指令中的跳转目的，象在 JCXZ 指令中那样在一个字节中指定。这样，上述例子现在变成如下所示：

1. 执行“装入基本指令”。
2. 把 AL 中的字节取补。
3. 执行“存放基本指令”。
4. 执行 LOOP 指令，当 CX≠0 时回到步骤1并重复执行。

其中每一步代表一条8086指令。

上面介绍的 LOOP 指令是一条不依据标志位的跳转指令，但我们已经看到对某些串操作，基于 CX 的内容和 ZF 标志的设置来构成循环才是合理的。对应这种要求的8086指令是 LOOPZ（若 ZF 置位且 CX≠0 就循环）和 LOOPNZ（若 ZF 复位且 CX≠0 就循环）。当然，同 LOOP 指令一样，LOOPZ 和 LOOPNZ 在循环之前都把 CX 减1。这些指令的另外的名称是 LOOPE（若相等且 CX≠0，就循环）和 LOOPNE（若不相等且 CX≠0 就循环）。显然，这些名字已经很清楚地指明了我们进行循环的基础条件。

我们再次把对一个字节序列取补的操作过程，作为使用 LOOPNZ 指令的例子，但这次却不具体指定序列中的字节数。已知在序列中除了结尾字节以0标志结束外，没有一个字节是0。这个操作的步骤现在可以表示如下：

1. 执行“装入基本指令”。
2. 把在 AL 中的字节取补。
3. 执行“存放基本指令”。
4. 执行LOOPNZ指令，当 ZF=0 且 CX≠0时回到步骤1并重复执行。

注意，这里虽然事先不需要给 CX 指定确切的字节数，但却要保证 CX 是一个足够大的数（为什么？请读者考虑）。

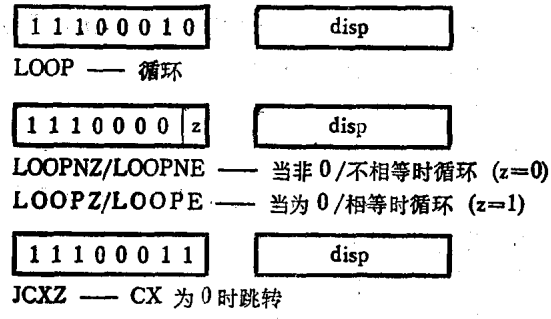


图3.38 模拟 REP 前缀的指令格式（指令中的 disp 表示用一个字节的偏移量之差）

模拟重复前缀的指令格式在图3.38中表示。

最后，让我们用一个把在0和15之间的数翻译成格雷码（Gray code）的例子，来结束关于

串操作的讨论。格雷码的特点是在连续的值中每次只有一位变化。数 0 到 15 的格雷码如下：

2 进制数	格雷码
0000	0000
0001	0001
0010	0011
0011	0010
0100	0110
0101	0100
0110	0101
0111	0111
1000	1111
1001	1110
1010	1100
1011	1101
1100	1001
1101	1011
1110	1010
1111	1000

假定有一个在当前数据段中，开始于偏移量为 100 的字节序列，这个字节序列由 0 到 15 之间的 2 进制数组成。同时假定 CX 含有该序列的字节数；另外，再假定 BX 含有上述 16 个字节格雷码翻译表的第一个字节的偏移地址，对这种操作，XLAT 指令无疑是最理想的。若我们要把已翻译的字节序列，放入从偏移量为 50 的单元开始的附加段，则完成这项工作的步骤可以表示如下：

1. 把 100 传送到 SI。
2. 把 50 传送到 DI。
3. 执行“装入基本指令”。
4. 执行 XLAT 指令。
5. 执行“存放基本指令”。
6. 执行 LOOP 指令，当 CX≠0 时回到步骤 3 并重复执行。

在这里，XLAT 指令完全适合做串循环操作，就好象它是专为此目的而设计的一样（为什么？请读者自己思考）。

§3.5 无条件转移指令

8086 的无条件转移指令的主要类型有跳转、调用和返回。跳转指令把一个值装入指令指示器，所以就打断了指令原来执行的序列。调用指令在改变指令指示器的内容方面，同跳转指令是一样的，但是，它们在改变指令指示器的值之前，先把当前指令指示器的内容保护到堆栈中，以便在将来的某个时刻能返回并在它原来离开的地方继续执行下去。返回时，把被保护的值，从堆栈中弹出并把该值放回到指令指示器，从而重新开始按以前被中断的顺序执行，调用和返回是用来支持调用子程序的机构。所有这一切都是我们已经掌握的老内容。

我们所要接触的新的内容是 8086 的调用、跳转和返回出现了两种不同的情况——段内和

段间。段内是指在当前代码段内转移控制；段间是指把控制转移到另一个任意代码段(通过改变 CS 的内容)，经过段间转移，该任意代码段就变成了当前的代码段。

很明显，段间转移可以完成段内转移所能完成的一切任务，并且还能完成一些段内转移所不能完成的任务。8086 之所以要使两种转移指令并存，是因为段间转移指令，往往需要更多的代码字节和执行时间。

让我们先看一个段间跳转的例子。假定当前的代码段开始于 0B0000H，指令指示器的内容为00A0H，这说明下一条要执行的指令，是在地址为 0B00A0H 的单元中。假定在该单元中有一条要把控制转移到地址为 0A0100H 的单元的跳转指令，由于当前代码段的范围是从 0B0000H 到 0BFFFFH，因此，从地址在 0B00A0H 的单元到地址为 0A0100H 的单元的跳转，将必须是个段间跳转。这样的一个段间跳转，必须在为 IP 指定一个新值(0100H)的同时，也为 CS 指定一个新值(例如为 0A000H)。这个例子可由图 3.39 形象地说明。

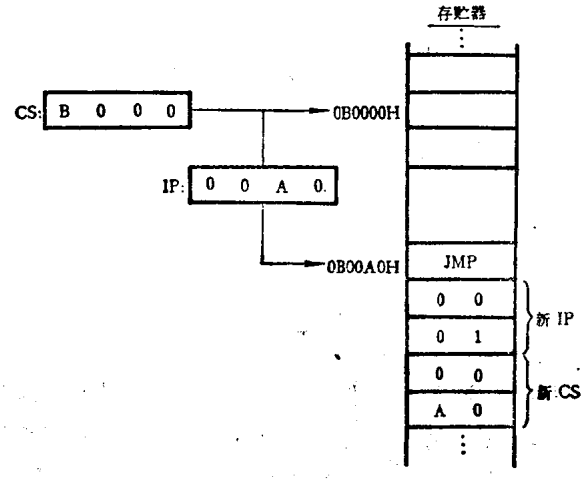


图 3.39 一条段间跳转指令的例子

段间调用，必须象把指令指示器的当前值保护到堆栈中一样，把代码段寄存器的当前值也保护到堆栈中。段间返回则相应地要从堆栈中弹出 2 个 16 位值，并把它们分别放入指令指示器 IP 和代码段寄存器 CS 中。段间调用和返回同段内调用和返回之间的区别是显而易见的，后者只保护和恢复指令指示器的值。

前面的例子，除了说明了段间跳转外，还说明了一个概念，即直接跳转。直接跳转(或调用)立即告诉我们要跳转到那里去，而间接跳转则只告诉我们到哪里去找要跳转的目标。间接跳转在我们编制程序时，不知道确切的跳转目标时是很有用的。例如，一条间接段间跳转或调用指令，可以使用 mod，r/m 字段来指定存贮器中 4 个连续字节单元中的第一个单元的偏移量(因为没有 4 字节的寄存器)。这 4 个字节中的前 2 个字节是 IP 的新值，后 2 个字节则是 CS 的新值。这些值可以是以前某些指令的运算结果。

返回指令用不到含有返回到哪里去的信息，它只是把控制返回到原来转来的地方。因此，间接返回这种概念是没有的。无条件跳转、调用和返回指令的格式表示在图 3.40 中。

段内跳转指令为指令指示器指定一个新值，但并不为代码段寄存器指定一个新值。例如，考虑在当前代码段中偏移量为 01A8H单元的一条跳转指令，这条跳转指令要使程序跳回 8 个字节而到 01A0H 单元，值 01A0H 可以放在跳转指令的 2 个字节中，事实上，在许多其他处理器中也确实是这样做的。但这样做有两个缺点：第一，许多跳转的目的是附近的单元，而指令却必须为此贡献出两个字节来指定跳转目的；第二，若由于某些原因，整个从 01A0H 到 01B0H 的代码区要迁移到偏移量为 0500H 到 0510H 的区域去，则指定偏移量 01A0H 的跳转指令就再也不会执行向后跳转 8 个字节的正确操作了。我们把上述那种代码位置一经移动就不能执行正确操作的代码，称为非浮动代码，而把在移动之后仍能正确执行的代码称为浮动代码(有时也称为位置独立代码)。若跳转指令不指定 01A0H，而只指定 =8H，显然，它就

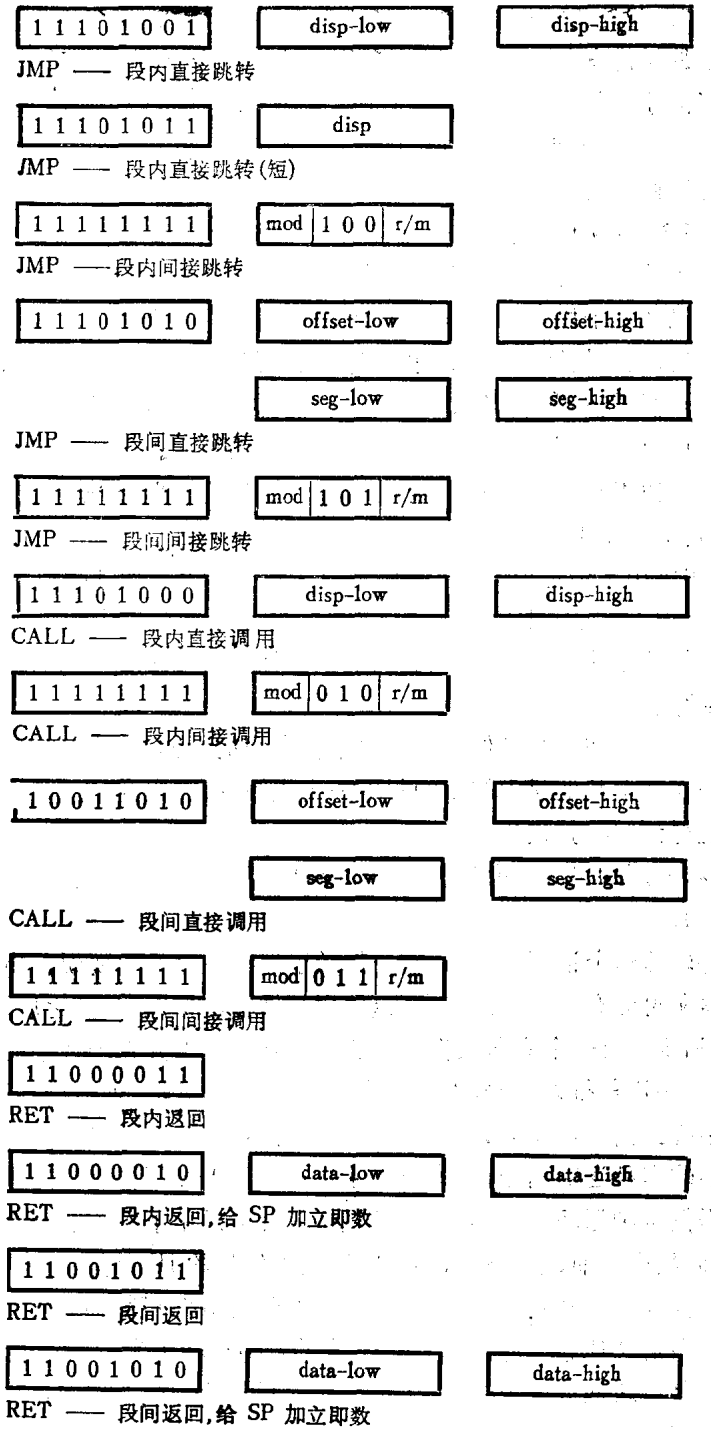


图 3.40 无条件跳转，调用和返回指令格式

变成了浮动代码(即是位置独立的)，并且跳转目的只需要一个字节就可以存放了。这样，段内直接跳转和调用指定的就不是目的偏移量，而是目的偏移量和跳转或调用指令末尾的偏移量之间的差(作为带符号数，图 3.40 中用 disp 表示)。另外，若一条跳转指令的偏移量之差可以放入 8 位(这种情况是大量出现的)，就能使用一种段内直接跳转指令的短形式，这种短形式的

段内直接跳转指令，比普通的段内直接跳转指令要短一个字节。调用指令没有短形式，因为对附近单元的调用不是经常发生的。

上面我们已经论述了使用相对偏移量（偏移量之差），而不使用真实偏移量的好处。一个真实的偏移量是一个16位的无符号数（从0到65535），它可以用来指定在当前代码段中的任何单元。一个相对偏移量是一个带符号数（从 −32768 到 +32767），它能覆盖相同的范围吗？例如，当一条跳转指令，要从偏移量0到达偏移量65535时如何用相对偏移量来表示？最大的正相对偏移量是 +32767，显然，用这个数不能达到目的。但是，当我们转而考虑负的相对偏移量时，由于跳转指令已经处于段中的最低偏移量，因而只要用一个值为 −1 的相对偏移量，就可以到达在段中的最高偏移量，即65535。事实上，在偏移量为0的跳转指令，可以用一个负的相对偏移量来到达偏移量在32768到65535之间的任何单元。由此可见，在一个段中任何单元的跳转指令，可以用相对偏移量跳转到在该段中的任何单元。

前面讨论了关于使用相对偏移量，而不是真实偏移量的问题，对于它们的讨论并不适用于间接跳转和调用，也不适用于段间跳转和调用。这是由于：

1. 间接跳转和调用不指定目的，它们只是指定到哪里去寻找跳转目的，很可能会有多个间接跳转或调用，指定使用放有某一目的地址的存贮器单元中的内容作为跳转目的，这样，使用相对偏移量就没有意义了，因为我们不知道有多少指令与该单元有联系（因而也就无法“浮动”）。
2. 段间跳转和调用，在某个其他的代码段中指定目的。如果含有段间跳转或调用的代码区被移动，但在某个其他段中的目的却不一定也被移动，因此，在这种情况下使用相对偏移量，并不能得到“浮动”代码。
3. 段间跳转和调用的目的，并不一定在附近的单元，因此希望使用相对偏移量来节约字节是不现实的。

在离开无条件转移这个话题之前，还有一件关于返回指令的事要说。有一种返回指令，它在恢复指令指示器和可能有的代码段寄存器（使用从堆栈中弹出的值）之后，给堆栈指示器加一个常数（包含在指令中的一个立即操作数）。它具有弹出和丢弃在堆栈中的附加项的功能。这些项，往往是在发出调用前推入堆栈，以便使被推入的值的序列，通过堆栈这个中介而传递到被调用的子程序中去。当子程序完成它的工作并执行返回时，这些值就不需要了。这种值称为参数。上面描述的那种返回指令，就为子程序执行完后丢弃不再需要的参数，提供了一种有效的手段。若不提供这种返回—丢弃指令，这些参数就必须用下列方法来丢弃：

1. 在执行返回指令之前，子程序从堆栈中移出被保护的IP（可能还有CS）的值，并把它保护到存贮器的某个地方。这样就暴露了在被保护的IP（可能还有CS）值之下的参数。
2. 然后，子程序给SP加一个常数。这一步具有弹出和丢弃参数的效果。
3. 然后，子程序重新把被保护的IP（可能还有CS）值，推入到堆栈顶。
4. 最后，子程序执行一条返回指令。

显而易见，返回—丢弃指令的执行过程，要简单而又有效得多。

丢弃参数的另一个方法，是在子程序执行返回指令之后减堆栈指示器。初看起来，这个办法几乎和返回—丢弃指令一样有效，但是，要知道子程序一经执行返回指令，减堆栈指示器的工作就再也不能由子程序来完成了。这步工作必须在每个子程序返回的地方加做。一想到一个子程序可以被从许多不同的地方调用时，这种解决办法就不那么吸引人了。

返回—丢弃指令使用两个字节 (16 位) 来存放参数的个数 (即要加到 SP 上去的值)。在大多数情况下，存放这个数用一个字节 (8 位) 已经足够，并且还可以使指令缩短一个字节。然而，可能会出现 8 位不够存放的少数情况，此时再改变为如上所述的丢弃参数的方法就太讨厌了。所以额外的字节还是放在指令中。

§3.6 条件转移指令

8086 提供了伴随着比较指令 (CMP) 的条件跳转指令。条件转移实际上是分 2 步来完成的。8086 首先执行实现 2 个数相减的比较指令，根据结果设置标志并丢弃相减的结果；然后执行一条条件跳转指令，该指令对标志进行测试，若标志指出 2 个数满足一种特定的关系就跳转，否则继续顺序执行。例如，假定若 BH 中的数值等于 BL 中的数值，就执行某些指令，否则就不执行这些指令。这种任务可以这样来完成：

1. 比较 BH 和 BL (设置标志位)。
2. 若 0 标志 ZF=0，则跳转到第 4 步。
3. ZF=1，执行特定的指令。
4. ……

在这个例子中，比较指令从 BH 中减去 BL，并根据结果设置标志。若 BH=BL，则结果是 0，ZF 被置位为 1。这样，关于相等的测试就是对于 ZF 的测试，这一步是由在步骤 2 中的条件跳转指令来完成的。若 BH≠BL，则 ZF 是 0，第 3 步 (执行某些特定的指令) 就被跳过。

条件跳转指令的格式表示在图 3.41 中。

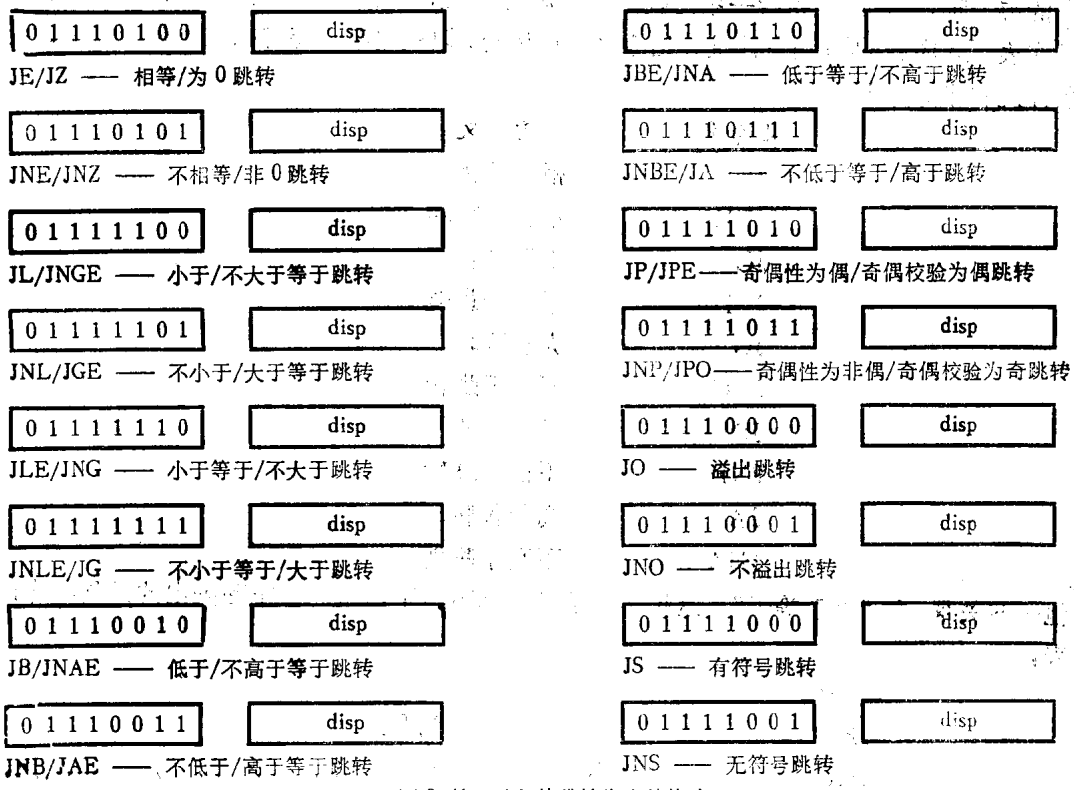


图 3.41 无条件跳转指令的格式

注意，这些指令都由一个 8 位操作码字节和一个跟着的 8 位跳转目的字节所组成。在这个跳转目的字节中，存放着目的偏移量和条件转移指令末尾的偏移量之差。正如前面已经介绍过的那样，这样安排可以提供浮动代码（跳转是相对的）和使代码结构紧凑（只用 8 位来指定跳转目的）。但也有不足之处，就是这样的安排限制了条件跳转指令跳转的范围，使它只能在指令前后的 127 个字节之内实行跳转。如果再提供一种长距离的条件跳转指令，就势必会占用太多的操作码。对大量统计资料的分析表明，“近的”（在 127 个字节之内）跳转出现的可能性要比“远的”跳转出现的可能性大得多，所以 8086 只提供了一种形式（“近的”）的条件跳转指令。对于另一种形式（“远的”）的跳转则作出一定的牺牲，即用两条指令来实现，一条条件跳转指令，先跳转到一个附近的单元，然后再通过放在该单元中的一条无条件跳转指令，跳转到真正的目的单元。由于这种情况出现得较少，从全局来看，得到的节约操作码的好处，比花 2 条指令执行一条长距离跳转指令的功能所付出的代价要大得多。

除了对相等进行测试以外，了解两个数中哪一个比较大常常是很有用的。但这就提出了一个有趣的问题。8 位数 1111 1111 大于 0000 0000 吗？答案是既对，又不对。若这些数被考虑为无符号的 2 进制数，第一个数是 255，当然大于 0。但如果这些数被考虑为带符号的 2 进制数，则第一个数是 −1，就比 0 小。显然，依据这些数是带符号的还是无符号的，将有两种方法来看待“较大”和“较小”。为此，我们就要引进一些新的术语来区分这两种情况。如果把数作为带符号数来比较，就使用术语“小于”和“大于”；如果把数作为无符号数来比较，就使用术语“低于”和“高于”。因此，1111 1111 高于 0000 0000 而同时又小于 0000 0000。但是，0000 0000 既低于也小于 0000 0001。

概括起来，在两个数之间，可能存在的各种关系是相等，高于，低于，小于和大于。这些关系中的每一种，都可以在执行一条比较指令之后，由标志的设置来决定。这些标志的设置已经表示在表 3.3 中。8086 用条件跳转指令来测试标志，以决定某一特定的关系是否满足。特定的条件跳转指令如下：

指令名	含　义
JE	相等跳转
JNE	不相等跳转
JL	小于跳转
JNL	不小于跳转
JG	大于跳转
JNG	不大于跳转
JB	低于跳转
JNB	不低于跳转
JA	高于跳转
JNA	不高于跳转

其他一些关系，如“小于等于”，这和不大于是一回事。下面是一张上面列出的跳转指令的别名表：

指令名	别名	
JE	JZ	为 0 跳转
JNE	JNZ	非 0 跳转

JL	JNGE	不大于等于跳转
JNL	JGE	大于等于跳转
JG	JNLE	不小于等于跳转
JNG	JLE	小于等于跳转
JB	JNAE	不高于等于跳转
JNB	JAE	高于等于跳转
JA	JNBE	不低于等于跳转
JNA	JBE	低于等于跳转

对于各种条件跳转的真正的标志设置表示如下：

指令名	标志设置
JE/JZ	ZF=1
JNE/JNZ	ZF=0
JL/JNGE	(SFxorOF)=1
JNL/JGE	(SFxorOF)=0
JG/JNLE	((SFxorOF) or ZF)=0
JNG/JLE	((SFxorOF) or ZF)=1
JB/JNAE	CF=1
JNB/JAE	CF=0
JA	(CF or ZF)=0
JNA	(CF or ZF)=1

有的条件跳转指令，并不关心两个数之间的关系，而只关心一个特殊的标志位是否被置位。例如，上面提到的 JZ 和 JNZ 指令，实际上只测试 0 标志。又如，上面提到的 JB 和 JNB 指令，它们只测试进位标志。其他只测试一个特殊标志位是否置位的条件跳转指令如下：

指令名	含 义	标志设置
JS	有符号跳转	SF=1
JNS	无符号跳转	SF=0
JO	溢出跳转	OF=1
JNO	不溢出跳转	OF=0
JP	奇偶性为偶跳转	PF=1
JNP	奇偶性为奇跳转	PF=0

其中最后两条指令的别名为：

指令名	别名	别名的含义
JP	JPE	奇偶校验为偶跳转
JNP	JPO	奇偶校验为奇跳转

§3.7 中断指令

大多数现代处理器，都提供了能被外部设备中断的机构。有了中断机构，处理器就可以从观察这些设备是否需要服务的周期性检测中解放出来。处理器和键盘之间的联络，很能说明

这个问题。当键盘上的一个键被按下时，通过某种途径通知处理器，此时已有一个字符在输入转接口等待处理比之由处理器频繁地询问是否有一个字符已由键盘输入到输入转接口（大多数情况下是得到一个否定的回答）的方法要有效得多。上述第一种方法称为中断，而后一种方法则称为轮询。由于中断的方法比之轮询的方法要有效得多，所以这种方法已为现代处理器所广泛采用。

一、中断机构

我们已经知道 8086 的外部设备能通过 2 个中断引脚，来“争取”得到 CPU 为之服务。它们是 NMI (none maskable interrupt)，即不可屏蔽中断引脚和 INTR (interrupt)，即可屏蔽中断引脚。我们先讨论 NMI 引脚。当某个外部设备把一个信号送给该引脚时，处理器不管正在做什么，都将停下来并为这个中断服务。由于外部设备提出中断请求时，处理器很可能正在执行一项非常重要的任务，因此，外部设备除了在十分紧急的情况之外应该避免引起这种中断。一种实际的紧急情况是，如果一台外部设备发现交流电源的电压刚刚通过 100 伏并且继续在下降，用技术术语来说就是发生了“掉电”。在这种情况下该外部设备通过 NMI 引脚提出中断以通知处理器，它没有很多时间可以工作了，当然是正当的。在剩下的几个毫秒中，处理器要在它的晶体振荡器停止振荡以前把内部事务处理好（如把重要的结果转移到安全的地方去等）。除去类似这样的紧急情况，一般的外部设备提出中断，都应该使用 INTR 引脚。当处理器处于不允许这种中断的模式时，处理器将不响应在 INTR 引脚上提出的中断请求。处理器的这种允许或不允许中断的“模式”，是由中断允许标志 IF 来决定的。当 IF＝1 时，处理器处于允许中断模式，此时它将响应由 INTR 引脚提出的中断请求；当 IF＝0 时，处理器处于不允许中断模式，此时它将不响应由 INTR 引脚提出的中断请求。关于设置和清除IF的指令 (STI, CLI)，将在后面的标志指令中讨论。

外部设备提出中断请求时，除了要在 INTR 引脚上加一个信号外，还必须把中断的原因通知处理器。在 INTR 引脚上提出中断的原因可以有多个(可有 256 个)，而在 NMI 引脚上提出中断的原因却只有一个。外部设备在请求处理器中断的时候，必须提供一个在 0 到 255 之间的一个数字，以代表 INTR 中断的原因。这个数字常常被称为中断类型。对于每一种不同的中断类型，处理器有一个必须执行的中断处理程序。这些程序的地址放在一个共有 256 项的表中，该表的每一项是一个 4 字节的指示器，它们是该类型中断处理程序的 CS 和 IP 值。这个表从存贮器地址 0 开始，具体如图 3.42 所示。当中断发生时所执行的程序，常常称为中断处理(或服务) 程序。

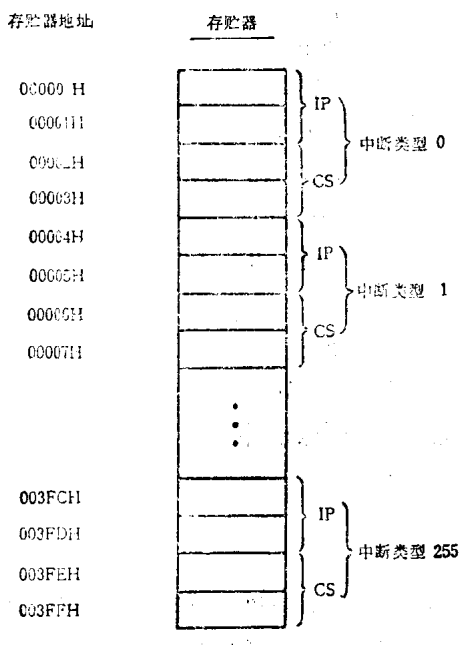


图 3.42 中断类型代码地址表

现在我们来看当处理器在 INTR 引脚上接收到一个中断请求，并且中断是允许的(IF＝1)时，处理器做些什么。在完成当前指令的执行以后，处理器就准备执行相应于该种类型中断的一段代码。首先，处理器将保护中断前主程序的有关信息，以便在它执行完中断处理

程序以后，能重新回到主程序中去继续执行。保护这些信息的最适宜的地方是堆栈，而被保护的信息将包括所有标志的当前值，CS 的当前值和 IP 的当前值。其次，处理器从外部设备取得中断类型代码，通过查表，把对应于该类型代码的指示器装入 IP 和 CS。例如，假定外部设备提供的类型代码为 0001H。在这种情况下，开始于地址 00004 的 16 位值就被装入 IP，而开始于地址 00006 的 16 位值则被装入 CS。这样，处理器要执行的下一条指令，就是对应于中断类型 1 的中断处理程序的第一条指令。

当处理器接收到一个在 NMI 引脚上的中断时（不管中断允许标志 IF 是否置位），它将执行 INTR 中断处理时所做的一切操作，但不需要从外部设备取得中断类型。理由很简单，因为 NMI 中断只有一种原因。在中断表中的类型 2 项，是为放 NMI 中断处理程序的指示器保留的。因此处理器在准备 NMI 中断处理时的最后一步工作，就是把表中对应于类型 2 的指示器装入 IP 和 CS。中断表中的其他保留项（包括那些可能被 8086 的未来型所用的）都表示在图 3.43 中：

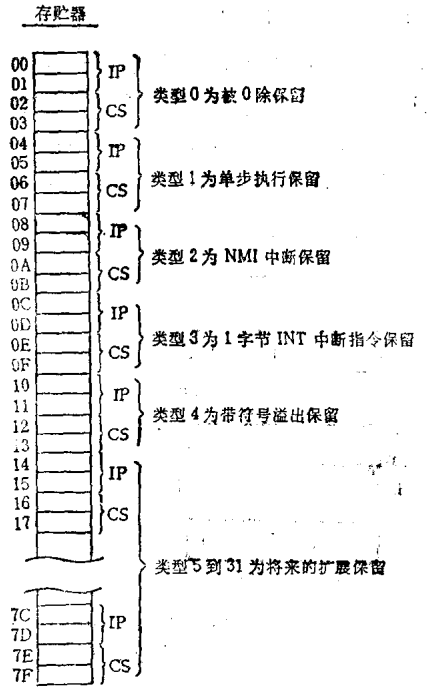


图 3.43　保留的中断类型

还有 2 个保留的中断类型，是被 0 除（类型 0）和带符号溢出（类型 4）。象 NMI 中断一样，这 2 个中断的处理不依赖于中断允许标志（IF）的值（事实上，所有保留的中断类型的处理都不依赖 IF 的值）。一旦处理器要用 0 作为除数来除时，处理器本身将自动地产生一个类型 0 中断。所以，在表中的类型 0 的指示器，应当含有从这种除法中恢复的程序的 IP 和 CS 值。带符号的溢出，虽然也是一个严重的事件，但处理器在带符号的溢出发生时，却不自动地产生一个中断，这是因为对于带符号和无符号运算都使用着相同的算术指令（例如，ADD）。在这种情况下，处理器当然无法知道当前进行的运算是带符号的还是无符号的。为此，8086 提供了一条有效的一字节指令，INTO（interrupt on overflow），这条溢出中断指令在溢出标志（OF）置位时就产生一个类型 4 中断。因此，对那些有溢出可能的运算，就应该在对带符号数进行运算的算术指令后面跟上一条 INTO 指令。

现在让我们考虑中断处理程序本身。中断处理程序改变标志的值并不奇怪，因为原来的标志值已经被保护起来了。但是，若中断处理程序还要改变处理器中其他的寄存器（或某些存贮器单元）的值，例如 AX 寄存器的值，则中断处理程序必须先保护这些寄存器（或某些存贮器单元）的值，待到中断处理程序要终止前，还必须依次把这些被保护的值加以恢复（一般是从堆栈中弹出）。最后中断处理程序通过执行一条 IRET（interrupt return），即中断返回指令而终止。这条指令将恢复保护在堆栈中的 IP、CS 和标志位的值。注意，就恢复标志而言，中断返回和前面讨论的段间返回是不同的，但它们也有共同点，就是两者都恢复 IP 和 CS。

另一条经常和中断联系在一起的指令是 HLT（halt）停机指令。HLT 指令使处理器停止执行，并使 CS 和 IP 处于指向跟在 HLT 后面的指令的状态。当一个中断来临时，CS 和 IP 的值就被保护到堆栈中，随后处理器就开始执行中断处理程序的指令。当遇到处理程序末

尾的 IRET 指令时，被保护的 IP 和 CS 值就得到恢复。在这个时刻，处理器已经"忘记"了在中断前它是处于暂停状态而着手准备执行现在由 CS 和 IP 所指示着的指令，即跟在 HLT 后面的指令了。因此，从效果上来说，HLT 为处理器在等待中断期间提供了一种暂停的手段。

现在考虑所有的中断处理程序，它们共有256个，放在存贮器的各处等待着中断的发生，一旦它们得到调用就进入执行。事实上，在不发生中断的情况下，调用它们中的一些程序也是很有用的。由于执行这些程序所需要的 IP 和 CS 值存放在存贮器的 4 个连续字节中，因而似乎通过执行一条指定这 4 个字节的段间间接调用指令，就能调用一个中断处理程序了。但是，当心！中断程序并不以一条普通的返回指令作为结束，它是用一条 IRET 指令结束的。IRET 指令除了把 IP，CS 从堆栈中弹出以外，还要把被保护的标志从堆栈中弹出。因此，要使这种返回正常工作，标志就一定要在堆栈中。虽然这可以在段间间接调用指令前，放一条把标志推入堆栈的指令 (PUSHF) 来实现，但这样做很麻烦。若有一条使处理器除了中断类型是由指令提供，而不是由外部设备提供，进行中断处理的指令就好了。8086 有这样一条指令，它就是 INT (interrupt)，即中断指令。INT 指令的格式和其他与中断有关的指令 IRET、INTO 和 HLT 指令的格式一起表示在图 3.44 中。中断允许标志 (IF) 的值，不影响 INT 指令的执行。

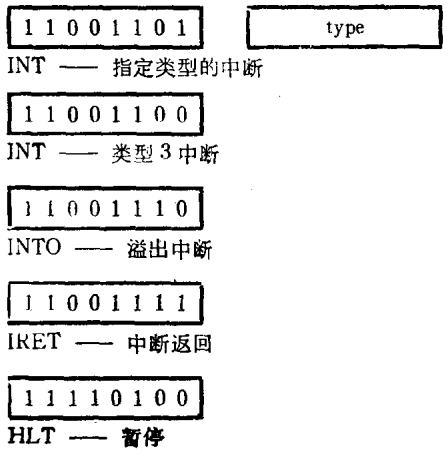


图 3.44 中断指令的格式(图中 type 表示在指令字节中指定的中断类型)

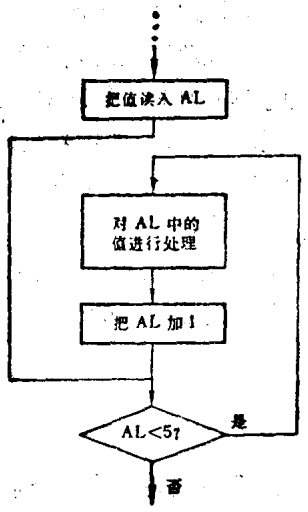
二、调试程序的需要

注意，关于 INT 指令有两种形式。第一种形式，指令是两字节长，并且第二个字节指定中断的类型。第 2 种形式，指令只有一字节长，并且类型被隐含地指定为 3 (见图 3.43 中的保留中断类型)。这条指令的中断类型是 3 这件事倒无关大局(它可以是任何类型)，但它是一字节长这一事实却至关重要。一字节 INT 指令主要应用于软件调试程序操作。要懂得为什么，就必须描述一下调试工作的情形。

软件调试程序，是用来找出程序不能正确工作的原因的人—机对话程序。假定要把程序运行到地址 100，用调试程序的行话来说，叫做在地址 100 "设置一个断点"。调试程序用在地址 100 放一条把控制转移回调试程序的指令的方法来设置一个断点。随后调试程序让程序运行，当程序运行到地址 100 时，将由预先放置的指令把控制返回到调试程序。当然，调试程序在设置断点之前，必须保护地址 100 原来的内容，并在控制返回到调试程序之后恢复原来的内容。

现在的问题是把哪一条 8086 的转移指令，放置在地址 100。如果在任何给定的时刻只设置一个断点，一条跳转指令就能胜任这项工作了。但是，如果要设置多个断点，调试程序就需要知道实际到达了哪个断点。这时 INT 指令就是最理想的，因为它保护了能确定断点的信息 (CS 和 IP)。使用 2 字节的 INT 指令在地址 100 设置一个断点将意味着地址 100 和 101 的内容都必须重写。调试程序要保护并最终把两个字节的内容都恢复。这在大多数情况下不会出问题。但是，不能排除会碰到一个如图 3.45 中所示的程序，它迂回跳转并且要在执行地

址为 100 的指令之前，执行地址为 101 的指令，在这种情况下，地址为 101 的指令，已被放置在地址为 100 的 INT 指令的第 2 个字节暂时重写了。显然，这就是调试程序必须要使用一字节 INT 指令的原因。

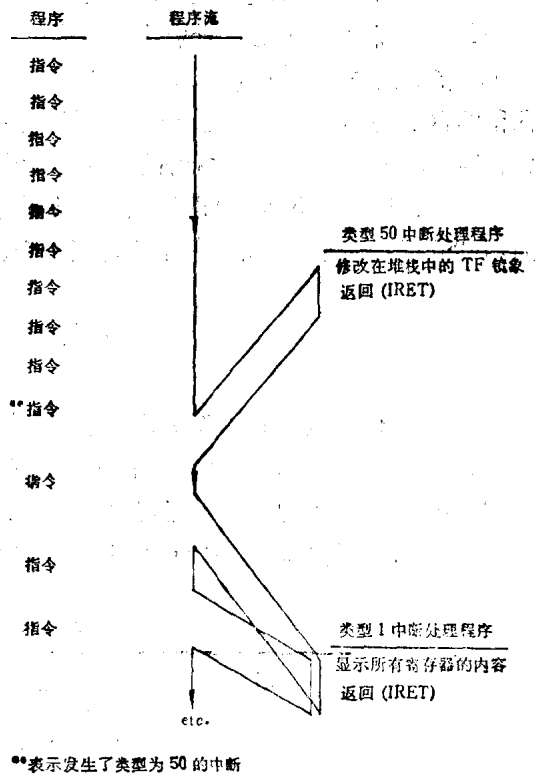


地址	指 令
⋮	
90, 91	从转接口 1 中把数值读入 AL (IN—2字节)
92, 93	转至 101 (JMP—2字节)
94, … 99	对在 AL 中的值进行处理
100	把 AL 加 1 (INC—1 字节) (用 INT设置断点)
101, 102	把 AL 和 5 比较 (CMP—2 字节)
103, 104	如果小于转至 94 (JL—2 字节)

图 3.45 先执行在地址 101 的指令后执行在地址 100 的指令的程序

当程序正在调试时，调试程序将使用1字节的 INT 指令去产生类型3中断。所以，如果打算使用调试程序来调试某个程序，就不应该在该程序中使用 1 字节的 INT 指令或类型 3 中断。

另一个可以用来调试程序的机构是自陷标志(TF)。一旦这个标志置位，处理器将每执行一条指令，就产生一个类型 1 中断 (见图 3.46)。这样就使调试程序能逐条执行所要调试的程序，并在每条指令执行之后，检查这条指令执行了些什么操作，这种执行方式称为单步 (单步执行重复的串指令时，将在每次重复之后产生一个中断，而不是在整个串指令结束之后才产生中断)。



**表示发生了类型为 50 的中断

图 3.46 用单步方式执行一个程序

调试程序用修改由以前的中断保护到堆栈中的标志集(标志的镜象)的办法，使被保护的 TF 值变成 1，然后执行一条中断返回(IRET)指令，而使程序进入单步执行方式。由于是调试程序，而不是由被调试的程序来决定程序何时进入单步执行方式，所以不需要用一条指令来设置或清除 TF。例如，假定某程序正在以全速执行。我们想要把它暂停并使它按逐条指令重新开始执行，在每条指令执行之后，还要检查所有寄存器的内容，以便据此判断程序是否在按预定的方式运行。要达到这个目的，我们可以在 INTR 引脚上加一个信号，并把一个中断

类型，设为50提供给处理器。处理器接到中断请求以后将停止执行原来的程序(假定IF=1)，并且把标志、CS 和 IP 保护到堆栈中，然后就开始执行类型50的中断处理程序。该中断处理程序的任务，就是取出被推入堆栈的信息，并把被保护的 TF 标志的值置位为1。执行完这项工作以后，该中断处理程序，就执行一条 IRET 指令，以恢复被保护的 IP、CS 和所有标志的值。此时，除了 TF 标志现在是1以外，其他值均不改变。这样，处理器就进入单步执行方式，每执行程序的一条指令之后，就产生一个类型1中断。为了响应中断，处理器又把标志(其中 TF=1)、CS 和 IP 的值推入堆栈进行保护，并开始执行关于类型1的中断处理程序。为了防止处理器单步地执行整个中断处理程序，在标志位被保护到堆栈中以后，TF 就被自动地清除。关于类型1的中断处理程序，应该有显示所有寄存器内容的操作指令。这个中断处理程序的最后一条指令又是 IRET，它恢复被保护的 IP、CS 和标志位的值，这样，TF 又一次为1，因而上述过程又将重复进行。这个例子解释了图3.46。刚才叙述的类型1和类型50的中断处理程序就是我们称为调试程序的一个组成部分。

三、一个 8086 的错误

让我们考虑把一个新值传送到堆栈段寄存器 (SS) 的指令。若我们想变换一个堆栈（当处理器交替地执行多个程序，而每个程序又有它自己的堆栈时，这是个有用的操作），这时就应该执行两条 MOV 指令。其中，第一条 MOV 指令把一个新值送给堆栈段寄存器(SS)；第二条 MOV 指令则把一个新值，送给堆栈指示器寄存器 (SP)。在两条 MOV 指令都执行之后，SS 和 SP 寄存器才一起指示新堆栈的顶部单元。然而，在第一条 MOV 指令执行之后，第二条 MOV 指令尚未执行之前，SS 和 SP 的组合却没有任何意义，当然它也就不指向任何为堆栈保留的区域的顶部。如果在此时刻恰好发生一个中断，而中断处理的第一步工作就是把标志位、CS 和 IP 推入堆栈进行保护，这样就会出错(可能会产生哪些严重后果，请读者考虑)。

这个错误是在8086已设计完成，并制造出来之后才发现的。但是，自从发现这个错误以后，8086已经修正，从而使它在执行把一个新值传送入 SS 之后不能立即接受任何中断。

§3.8 标 志 指 令

8086提供了设置和清除进位标志 (STC, CLC)、方向标志 (STD, CLD) 和中断标志 (STI, CLI) 的指令，另外它还有一条把进位标志取补的指令 (CMC)。这些指令总结在表3.10中。这些标志的用途已经讨论过，进位标志 (CF) 用于多精度运算；方向标志 (DF) 用于串处理；中断允许标志 (IF) 用于允许或不允许中断。标志指令的形式表示在图3.47中。

表3.10 标 志 操 作

CLC	(清除进位标志)	0 → CF
CMC	(取补进位标志)	1−CF→CF
STC	(设置进位标志)	1 → CF
CLD	(清除方向标志)	0 → DF
STD	(设置方向标志)	1 → DF
CLI	(清除中断允许标志)	0 → IF
STI	(设置中断允许标志)	1 → IF

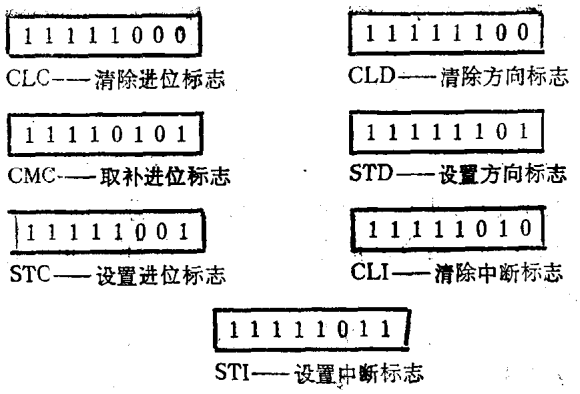


图 3.47 标志指令的格式

§3.9 同步指令

中断为 8086 提供了与外部设备同步的手段，下面再介绍另外两种 8086 结构提供的同步形式。第一种同步形式，能使 8086 和为它完成某些难以完成的工作的协处理器，如 8087 或 8089 相互协调地进行工作；第二种同步形式，使在多处理器系统中的 8086 能与系统中的其他处理器共享某些资源(例如存贮器)。

一、协处理器

8086 虽然有一个强有力的指令集，但毕竟还有不足之处。例如，它没有执行浮点数操作的指令。当然，可以用现有的 8086 指令写一个执行 2 个浮点数加法的程序，但这种程序执行起来要比执行一条浮点数加法指令的效果低得多。一个较好的解决办法，是用一个具有浮点数操作指令并能随时为 8086 提供服务的协处理器来做这种事。如果有这样的浮点处理器，在程序里就可以直接写上浮点指令，在程序执行过程中，一旦遇到这种指令，8086 就会调用该浮点处理器。

协处理器是以持续不断地观察并了解 8086 正在执行什么操作的方式来帮助 8086 工作的。设计 8086 时就决定当协处理器在等待中发现有 ESC... 指令时，就表示 8086 需要协处理器帮助。ESC 指令有一个 3 位的 x 字段和一个 3 位的 y 字段，它们用来指明需要哪一个协处理器帮助和该协处理器应该执行什么指令，当然，这两个字段在 8086 处理器中是无用的。事实上，这 6 位可以任意地进行组合以用来区别协处理器和/或其指令的 64 种组合。另外 ESC 指令有一个 mod 字段和一个 r/m 字段，它们被用来为协处理器指定一个在存贮器中的操作数。这两个字段实际上是由 8086 使用的，8086 计算该操作数的存贮器地址，并且从存贮器中读出该操作数的值。尽管 8086 得到该值并无用处，但协处理器却因此而知道了该操作数的地址和它的值。这样协处理器就知道了要执行的操作及被操作的数和地址，因而就能代替 8086 进行工作了。

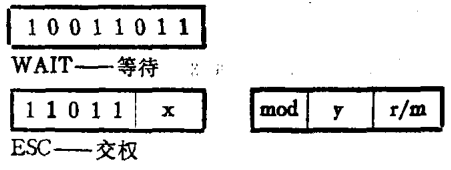


图 3.48 协处理器同步指令的格式

ESC 指令的形式和另一条等待指令 WAIT 的格式，一起表示在图 3.48 中。WAIT 指令是一条同步指令，它由 8086 执行，以测试协处理器何时完成它委托的操作。WAIT 指令将使 8086 暂

停等待，当协处理器完成工作时，它就在 8086 的 TEST 引脚上送入一个信号。8086 检察到在 TEST 引脚上的这个信号后，就开始继续它的操作。WAIT 指令象串指令一样，可以在该指令完成之前被中断。

使用 WAIT 和 ESC 指令的一种方法，是在每条 ESC 指令之前，放上一条 WAIT 指令。这样，我们就能确保在协处理器完成 8086 委托的任务之前，一条指令也不会发送给它。通过把 WAIT 指令放在 ESC 指令之前而不是之后，8086 就可以在协处理器正在执行一条指令的期间做其他的事情。

为了比较形象地说明协处理器帮助 8086 工作的情况，我们举一个例子。假如有两个要相加的浮点数，每个数是 4 字节长。第一个数在当前数据段中，起始于 SI 中的偏移量；而第二个数也在当前数据段中，但起始于 DI 中的偏移量；再假定有一个能响应 x 字段值为 101(2 进制)的 ESC 指令的浮点处理器，并且该浮点处理器能执行如下的指令：

001：把操作数装入浮点累加器。
010：把操作数加到浮点累加器。
011：从浮点累加器减去操作数。
100：把浮点累加器乘以操作数。
101：把浮点累加器除以操作数。
110：把浮点累加器存放入操作数单元。

这里的浮点累加器是在浮点处理器上的一个寄存器。

实现所要求的加法的 8086 指令序列表示在图 3.49 中。在这里，WAIT 指令使 8086 和浮点处理器保持同步，而 ESC 指令则用来在 8086 和浮点处理器之间传递信息。

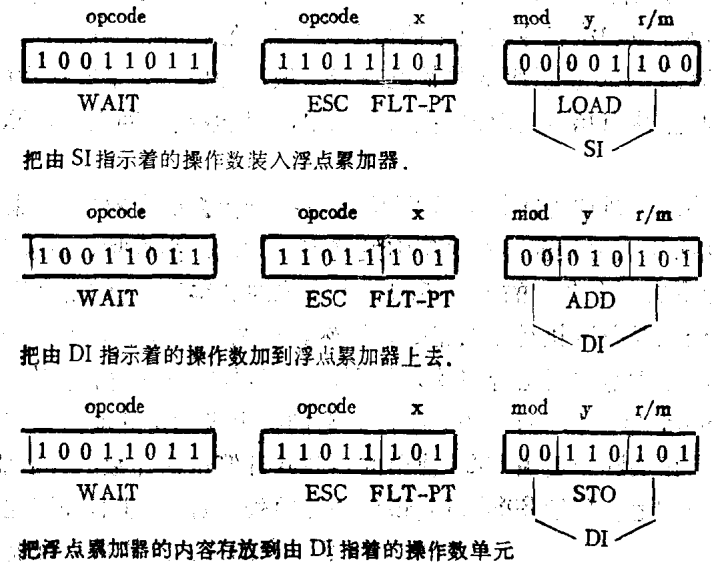


图 3.49 调用协处理器的指令序列(图中 FLT-PT 表示被指定的是浮点处理器)

二、资源共享

第二种形式的同步，是在多个处理器之中共享资源。例如考虑一个飞机订票系统，在系统中，全国各地的订票处理机共享着存贮器的公用数据库。假定某个处理机要为某甲预订从北

京到上海的飞机票。首先，该处理机将读那个数据库，并且假定发现那次航班还有15个位子空着。随后，它就把一个14写到数据库中指明空位子数的项目中去，从而为某甲订了一个座。但是，假如在为某甲订票的处理机读出15之后，但尚未写回14时，另一个订票站的处理机要在同一航班为某乙预订一张机票，并且该处理机也读出了值为15的空位数，并通过写回一个14而为某乙订了一个座。在这种情况下，哪一个处理机先写这个14是无所谓的，当两个处理机都完成了它们的处理之后，座位的计数仅从15减到了14，而甲乙两人却都认为他们已经订了票。

如果为某甲订票的处理机，用一条指令读出15，并写回14来为某甲订票，或许能避免这个问题。例如使用DEC（减1）指令去完成这种操作。但是，在没有空位（计数是0）的情况下，执行DEC指令将使计数变为负的（SF变为1）。此时，非但不能达到预订机票的目的，相反，还必须执行一条INC（加1）指令来恢复计数（恢复为0）。实际上，使用DEC指令什么问题也没有解决，因为除了会发生上述问题以外，读计数和写计数不用一条指令执行时发生的问题，在用一条指令执行时同样也会发生。例如，当某乙的处理机开始执行DEC指令时，正好某甲的处理机也在执行DEC指令，并虽然已经读出操作数但还没有写回结果。可以肯定，有朝一日这种事一定会发生。在这种情况下，某甲的DEC指令将取得一个值为15的数，然后做减1操作（与此同时某乙的DEC指令也取得同一个15），再放回一个14（与此同时，某乙的DEC指令做减1操作）。最后，某乙的DEC指令将也放回一个14。

显而易见，想在一条指令里更新计数也不是个办法，充其量，它只减少了发生这种冲突的可能性。对某甲的处理机来说仍然需要某种方法，使其他的处理机在它执行DEC指令期间，不能访问这个数据库。8086是用一个可以放在任何指令之前的一字节封锁前缀来实现这一功能的。执行这条指令将会使8086在该指令的持续期间，在它的LOCK引脚上输出一个信号。这样，飞机订票系统的硬件，就能够指派发出封锁信号的处理机单独访问存贮器。显然，若没有其他的处理机能使用这个存贮器，数据库就不可能被其他的处理机所访问。一条前面放着封锁前缀的减1(DEC)指令的例子表示在图3.50中。

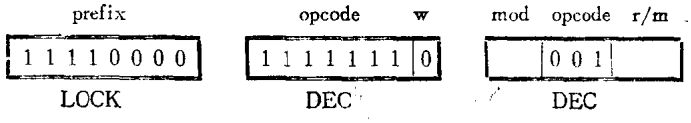


图3.50 一条带有封锁前缀的指令的例子
（图中的prefix表示是前缀字节）

§3.10 关于前缀

我们已经提到了8086中的3个指令前缀，即段超越、重复和封锁前缀。读者很可能会提出以下两个问题：

1. 每个前缀，都能用在任何指令之前吗？
2. 在同一条指令之前，能用多个前缀吗？

除了一个例外，每个前缀都可以放在任何指令之前。这个例外就是重复前缀，它规定只能用于串基本指令。若把它用于其他的指令，很可能由于该机构的执行方法而给出预想不到的结果。封锁前缀可以应用于任何指令，并将使处理器在该指令的持续期间，在它的LOCK引脚上输出一个信号。这个信号往往在处理器执行一条既读又写的指令（例如INC，DEC，

XCHG)期间，提供独占的存贮器访问(在一个多处理器系统中)。然而处理器并不介意是否把封锁前缀和任何其他指令一起使用，即使是一条不访问存贮器的指令，它也照样在LOCK引脚上输出封锁信号。最后，段超越前缀也可以和任何指令一起使用。如果这条指令访问在存贮器中的操作数，段超越前缀就指定一个段，否则就无效。

下面，让我们考虑前缀的组合。封锁前缀和段超越前缀可以在一起使用，并且将各自执行其指定的功能。指令的特性并不受前缀次序的影响。但是，重复前缀和其他前缀一起使用时却有些问题。因为，首先它只能应用于一条没有加过前缀的串基本指令，所以它必须总是放在最后的前缀。其次，封锁和重复前缀的组合能在一个相对长的时间里，如整个重复串指令的持续期间，防止系统中的其他处理器访问存贮器。

任何前缀和重复前缀的组合，会使串操作在被中断以后，丢失重复前缀以外的前缀。为了了解为什么会这样，让我们考虑在一条重复串指令的执行期间，当发生一个中断时会出现什么情况。如果中断要被迫等到所有重复都完成，它可能要等相当长的时间。因此，正如前面已经提到的那样，处理器在一条串指令的一次重复之后允许接受中断，在重复操作时，指令指示器包含着重复前缀的偏移量。如果这条指令被中断，它就是被保护的偏移量，并且也就是在中断处理完成以后重新开始执行的起始偏移量。显然如果这条指令含有任何先于该重复前缀的前缀，则经过中断再重新执行时，该前缀将再也不是指令的一部分了，也就是说它被"丢失"了(注意，重新执行的串指令，并不重做在中断前已经做过的事，在CX中的计数以及在SI和DI中的数，在中断之前的每次重复期间被更新，并且下一次执行就使用这些更新过的值)。如果在重复期间，指令计数器含有这条指令的第一个前缀字节的偏移量，这个问题就不存在了。这是8086设计中的一个缺陷！

另一条使用前缀有潜在问题的指令是WAIT。WAIT指令象重复串指令一样，能在完成它的任务之前被中断，并且由于上面给出的相同原因，如果在一个中断之后重新执行，WAIT指令将失去它的前缀。重复前缀不能和WAIT指令一起使用，并且封锁前缀和段超越前缀对WAIT指令都没有作用。因此，WAIT指令无论有或没有前缀，它在被中断之后，总是会正确地重新开始执行。

§3.11 标志设置

对于某些指令执行之后标志位设置情况的查询，贯穿着整个这一章。在这一节中，我们把它们连结起来以完整地描述标志的特性。

8086的标志，可以分成两种类型：状态标志和控制标志。前者反映某些指令的执行结果的性质，而后者则控制处理器的操作。表3.11列出了执行结果影响状态标志的指令和设置控制标志的指令。下面就这些标志中的特性作一些解释。

加法和减法指令用下面的方式影响所有的状态标志：溢出标志(OF)和进位标志(CF)，指出该指令的执行是否造成了一个溢出的带符号或无符号结果；辅助进位标志(AF)，指出在进行BCD数运算时，结果是否需要进行调整；符号标志(SF)，0标志(ZF)及校验标志(PF)则分别指出结果是否为负、0或含有偶数个1。

与加法和减法指令在同一组的，是比较指令(CMP，CMPS，SCAS)和取补指令(NEG)。比较指令执行一个减法并设置标志位以反映这个减法结果的性质。NEG指令(在把所有位取

表3.11 标志设置

A——状态标志	OF	CF	AF	SF	ZF	PF
加法和减法						
ADD ADC SUB SBC	+	+	+	+	+	+
CMP NEG CMPS SCAS	+	+	+	+	+	+
加1和减1						
INC DEC	+	−	+	+	+	+
乘法和除法						
MUL IMUL	+	+	?	?	?	?
DIV IDIV	?	?	?	?	?	?
10进制运算						
DAA DAS	?	+	+	+	+	+
AAA AAS	?	+	+	?	?	?
AAM AAD	?	?	?	+	+	+
布尔						
AND OR XOR TEST	0	0	?	+	+	+

A——状态标志	OF	CF	AF	SF	ZF	PF
移位和循环移位						
SHL SHR (单位)	+	+	?	+	+	+
SHL SHR (变量)	?	+	?	+	+	+
SAR	0	+	?	+	+	+
ROL ROR RCL RCR (单位)	+	+	−	−	−	−
ROL ROR RCL RCR (变量)	?	+	−	−	−	−
恢复标志						
POPF IRET	+	+	+	+	+	+
SAHF	−	+	+	+	+	+
进位标志设置						
STC	−	1	−	−	−	−
CLC	−	0	−	−	−	−
CMC	−	*	−	−	−	−

B——控制标志	DF	IF	TF
恢复标志			
POPF IRET	+	+	+
中断			
INT INTO	−	0	0
方向标志设置			

B——控制标志	DF	IF	TF
STD	1	−	−
CLD	0	−	−
中断标志设置			
STI	−	1	−
CLI	−	0	−

其中：+=影响 *=取补 1=设置为1
?=无定义 0=设置为0 −=不影响

反之后) 加1，并设置标志位以反映这个加法结果的性质。NEG 指令把进位标志设置为 1 的唯一时机是在被求补的值为 0 时；它把溢出标志设置为 1 的唯一时机是在被求补的值为 −128 (8 位) 或 −32768 (16 位) 时。

加 1 和减 1 指令，除了它们不影响进位标志外，其响应状态标志的方式，同加法和减法的方式一样。这样，我们就能写一个执行多精度运算的循环如下：

1. SI 取得第一个操作数的最低字节的偏移量。
2. DI 取得第二个操作数的最低字节的偏移量。
3. 清除进位位 (CLC)。
4. 把由 SI 指示着的字节带进位加 (ADC) 到由 DI 指示着的字节。
5. 把 SI 加 1 (INC) 以使它指着第一个操作数的下一个较高字节。
6. 把 DI 加 1 (INC) 以使它指着第二个操作数的下一个较高字节。
7. 若操作数还有字节没有被相加过，则跳转回步骤 4。

显然，如果在步骤 5 和 6 的 INC 指令影响进位标志的话，则下一次循环中在步骤 4 的 ADC 指令的执行就很可能不会给出正确的结果。

乘法指令产生 2 倍长的结果，并且可能因此而以 32 位作为状态标志的依据。由于没有其他指令有这种情况，势必要处理器单为这条指令安排一个特殊的标志设置机构。为了使处理

器不致过份复杂，所以在执行乘法指令后，对大多数标志的值不加定义。不加定义表示处理器不以任何特殊的方式去设置标志。处理器的未来型可能会以不同的方式执行这种指令，并给出对标志的不同的设置。

在执行一条乘法指令之后，了解积是否能作为一个没有产生溢出的单长度数是很有用的(看作2倍长度数的积不会溢出)。若了解到积能用单长度数表示，我们就能做诸如用一个字节乘以另一个字节并把积加到第三个字节那样的操作了。由于这个原因，溢出和进位标志是有定义的，它们指出在把积看成单长度数的时候，该乘法指令是否产生了一个带符号或无符号溢出的结果。

为了简单起见，在执行一条除法指令之后，所有的状态标志都没有定义。

在执行一次10进制数加或减调整之后的唯一重要标志是进位标志（为多精度运算所需要）；所有的其他标志可以不加定义。但是8080/8085有一条DAA指令（8080/8085唯一的10进制指令），并且该指令设置8080/8085所有的5个状态标志(8080/8085没有溢出标志)，为了兼容，8086的DAA指令也只好这样做。为了协调，DAS，AAA和AAS也应该影响这5个标志，DAS确实是这样做了，但对AAA和AAS由于实现上的困难，符号标志、零标志和校验标志只好不加定义。进位和辅助进位对于AAM和AAD指令意味着什么还不清楚，所以这些标志也留着不加定义。

由于布尔操作不会产生溢出的结果，所以在执行这样的指令之后，溢出和进位标志都被复位为0。辅助进位标志在布尔指令之后没有用处（设置它的唯一目的就是为了10进制调整），所以它不加定义。符号、0和校验标志被设置以反映该指令的结果。

在影响标志的布尔指令表中NOT被漏掉了。NOT指令不影响标志。出现这样的情况是由于设计8086时一个疏忽的结果。

移位指令不是别的，就是以2为阶的乘或除。除了辅助进位标志没有定义(在这里移位并不参与10进制运算)和对变量移位的溢出标志也没有定义（在这种情况下检测溢出标志的机构太复杂)之外，其他状态标志均反映结果的性质。算术右移(SAR)决不会产生带符号的溢出结果，所以在执行这条指令之后，溢出标志被复位为0。

循环移位指令设计得和8080/8085的循环移位指令相兼容，并且以相同的方式影响标志。由于这个原因，它们影响进位标志而不影响辅助进位标志，符号标志，0标志和校验标志。为了一致起见，还决定循环移位指令应当和移位指令以一样的方式影响溢出标志，尽管在这种情况下还不清楚溢出究竟意味着什么。

标志恢复指令，是把标志位恢复成以前保护的值。特别是，POPF和IRET把所有的标志（状态和控制一样）恢复成被保护在堆栈中的值。SAHF是一条比较奇怪的指令(它的存在只是为了能和在8080/8085中的一条类似的指令相兼容)，它把在AH寄存器中的值恢复成8080/8085的5个状态标志。

所有的中断，都清除中断允许标志和自陷标志。如果中断允许标志不被清除，一个“并发”的外部中断将使处理器以惊人的速度持续地把CS和IP推入堆栈，从而使堆栈马上溢出。如果自陷标志不被清除，当调试程序试图以单步方式来执行被调试程序时，处理器却把调试程序本身也单步执行。

进位标志指令，方向标志指令和中断标志指令的特性是很直观的，它们的设置、清除、取补不影响任何其他标志。

§3.12 8086 指令详细解释

在这一节中，列出了8086全部指令的详细解释。为了读者查找方便，在这一节中指令是

表3.12 MCS-86符号的含义

MCS-86 符号	含义	MCS-86符号	含义
AX	累加器(16位)(8080累加器仅有8位)	w	指令中的一位字段，w=0表示字节指令，w=1表示字指令
AH	累加器(高字节)	d	指令中的一位表示方向的字段，确定指令中哪一个操作数是源，哪一个操作数是目
AL	累加器(低字节)	(…)	圆括号表示被括的寄存器或存贮器单元的内容
BX	寄存器BX(16位)(8080寄存器对HL)，它可以分开使用并能象两个8位寄存器一样寻址	(BX)	表示寄存器BX的内容，它可以作为一个存放8位操作数单元的地址，在汇编指令中用于这个目的时，BX必须用方括号括起来。
BH	寄存器BX的高字节	((BX))	表示一个由BX寄存器的内容所指示着的存贮器单元中的一个8位操作数
BL	寄存器BX的低字节	(BX)+1，(BX)	表示一个16位操作数的地址，该操作数的低8位放在由寄存器BX的内容指示的存贮器单元中，该操作数的高8位放在相邻存贮器单元(BX)+1中
CX	寄存器CX(16位)(8080寄存器对BC)，它可以分开使用并能象两个8位寄存器一样寻址	((BX)+1，(BX))	表示存放在那里的一个16位操作数，例如，((DX)+1，(DX))表示一个由两个8位字节连接起来的字，它的低字节在由DX的内容所指示的存贮器单元中，它的高字节在下一个顺序的存贮器单元中
CH	寄存器CX的高字节	offset	在存贮器中的一个字节的偏移地址(也就是相对于某一段基地址的偏移量)(16位)
CL	寄存器CX的低字节	offset-low	一个偏移地址的低字节
DX	寄存器DX(16位)(8080寄存器对DE)，它可以分开使用并能象两个寄存器一样寻址	offset-high	一个偏移地址的高字节
DH	寄存器DX高字节	offset+1: offset	开始于偏移地址offset的两个连续字节的偏移地址
DL	寄存器DX低字节	data	立即操作数(若w=0，则为8位，若w=1，则为16位)
SP	堆栈指示器(16位)	data-low	16位数据字的低字节
BP	基地址指示器(16位)	data-high	16位数据字的高字节
IP	指令指示器(8080程序计数器(16位))	disp	位移量
FLAGS	16位寄存器，其中存放9个标志位(不直接等效于8080的PSW，8080的PSW包含了5个标志位以及累加器的内容)。	disp-low	16位位移量的低字节
		disp-high	16位位移量的高字节
DI	目变址寄存器(16位)	←	赋值
SI	源变址寄存器(16位)	+	加
CS	代码段寄存器(16位)	−	减
DS	数据段寄存器(16位)	*	乘
ES	附加段寄存器(16位)	/	除
SS	堆栈段寄存器(16位)	%	模除
REG 8	一个8位CPU寄存器单元的名字或编码	&	与
REG 16	一个16位CPU寄存器单元的名字或编码	|	或
LSRC，RSRC	称为一条指令的操作数。当使用两个操作数时，一般称为左源操作数和右源操作数。左操作数也称为目操作数，右操作数也称为源操作数	‖	异或
reg	在指令说明中reg字段指定REG 8或REG 16		
EA	有效地址(16位)		
r/m	指令mod，r/m字节的第2，1，0位，这三位字段连同mode和w字段共同确定EA		
mod	指令mod，r，m字节的第7，6位。这两位字段定义寻址方式		

按字典序排列的。通过查阅本节有关各条指令的解释，一定会给读者在编制和阅读 8086 的汇编程序时带来极大的方便。表 3.12 列出了在本节中所使用的 MCS-86 符号的含义。

AAA

(Ascii adjust for addition ASCII 加法调整)

操作：若 AL 的低 4 位大于 9 或辅助进位标志置位，则对 AL 加 6 并对 AH 加 1。AF 和 CF 置位。由于上述加法，AL 的新值的高 4 位都是 0，并且低 4 位是介于 0 和 9 之间的数字。

```
if((AL) & 0FH)>9 or (AF)=1 then
    (AL)←(AL)+6
    (AH)←(AH)+1
    (AF)←1
    (CF)←(AF)
    (AL)←(AL) & 0FH
```

编码：

0 0 1 1 0 1 1 1

时钟周期：4

例：AAA；在加法之后

影响标志位：AF，CF

无定义标志位：OF，PF，SF，ZF

说明：AAA（非压缩 BCD(ASCII) 加法调整）对两个非压缩 10 进制数相加在 AL 中的结果进行修正，以产生一个非压缩 10 进制和。

AAD

(Ascii adjust for division ASCII 除法调整)

操作：累加器的高字节(AH)被乘以10，然后加到低字节(AL)。结果存放回 AL。AH 被清 0。

```
(AL)←(AH)*0AH+(AL)
(AH)←0
```

编码：

1 1 0 1 0 1 0 1	0 0 0 0 1 0 1 0

时钟周期：60

例：AAD；在除法之前

影响标志位：PF，SF，ZF

无定义标志位：AF，CF，0F

说明：AAD（非压缩 BCD (ASCII 除法调整)在把两个非压缩的 10 进制数相除的指令之前，对在 AL 中的被除数实行调整，以使除法的结果是一个非压缩的 10 进制商。

AAM

(Ascii adjust for multiply ASCII 乘法调整)

操作：AH 的内容被 AL 除以 10 的结果所代替。然后 AL 的内容被该除法的余数所代替，即被 AL mod 10 所代替。

(AH)←(AL)/0AH

(AL)←(AL)%0AH

编码：

11010100	00001010

时钟周期：83

例：AAM；　在乘法之后

影响标志位：PF，SF，ZF

无定义标志位：AF，CF，OF

说明：AAM（非压缩 BCD（ASCII）乘法调整）对两个非压缩 10 进制数相乘后放在 AX 中的结果进行修正，产生一个非压缩的 10 进制积。

AAS

（Ascii adjust for subtraction ASCII 减法调整）

操作：若 AL 的低半大于 9 或辅助进位标志置位，则从 AL 中减去 6 并从 AH 中减去 1．AF 和 CF 标志置位。AL 中的旧值，因上述的减法而被一个高 4 位都是 0 而低 4 位是介于 0 到 9 之间的数的字节所代替。

if ((AL&0FH)>9 or (AF)=1 then

(AL)←(AL)−6

(AH)←(AH)−1

(AF)←1

(CF)←(AF)

(AL)←(AL) & 0FH

编码：

00111111

时钟周期：4

例：AAS；在减法之后

影响标志位：AF，CF

无定义标志位：OF，PF，SF，ZF

说明：AAS（非压缩 BCD（ASCII）减法调整）对两个非压缩 10 进制数相减在 AL 寄存器中的结果进行修正，产生一个非压缩的 10 进制数差。

ADC

（Add with carry 带进位加）

操作：若进位标志置位，ADC 在把两个操作数相加的结果存放到目（左）操作数中去之前给该和加 1。若进位标志没有置位，即是 0，1 就不加。

if (CF)=1 then (DEST)←(LSRC)+(RSRC)+1

else (DEST)←(LSRC)+(RSRC)

编码：

存贮器或寄存器操作数与寄存器操作数：

0 0 0 1 0 0	d	w	mod	reg	r/m

if d=1 then LSRC=REG, RSRC=EA DEST=REG
else LSRC=EA, RSRC=REG DEST=EA

时钟周期：(a)寄存器加到寄存器 3
(b)存贮器加到寄存器 9+EA
(c)寄存器加到存贮器 16+EA

例：

(a) ADC AX, SI
ADC ,SI；同上
ADC DI, BX
ADC CH, BL

(b) ADC DX, MEM_WORD
ADC AX, BETA[SI]
ADC ,BETA[SI]；同上
ADC CX, ALPHA[BX][SI]

(c) ADC BETA[DI], BX
ADC ALPHA[BX][SI], DI
ADC MEM_WORD, AX

立即操作数加到累加器：

0 0 0 1 0 1 0	w	data	data, if w=1

if w=0 then LSRC=AL, RSRC=data, DEST=AL
else LSRC=AX, RSRC=data, DEST=AX

时钟周期：4

例：ADC AL, 3
ADC AL, VALUE-13-IMM
ADC AX, 333
ADC AX, IMM_VAL_777
ADC ,IMM_VAL_777；同上

立即操作数加到存贮器或寄存器操作数：

1 0 0 0 0 0	s	w	mod	0 1 1	r/m	data	data, if s:w=01

LSRC=EA, RSRC=data, DEST=EA

时钟周期：(a) 立即数加到存贮器 17+EA
(b) 立即数加到寄存器 4

例：(a) ADC BETA[SI], 4
ADC ALPHA[BX][DI], IMM 4
ADC MEM_LOC, 7396
(b) ADC BX, IMM_VAL_987

ADC DH, 65
ADC CX, 432

若一个立即数据字节要被加到一个寄存器或存贮器字，则在相加之前该字节要被符号扩展至16位。在这种情况下，指令字节应是83H(即，s:w 位均置位)。

影响标志位：AF, CF, OF, PF, SF, ZF

说明：ADC(带进位加)执行两个操作数的加法，若CF标志置位，就再加1，并把结果返回到目(左)操作数。

注意：一旦使用含有变量名的地址表达式，即一个含有数据的存贮器单元名，mod, r/m 字节后将跟着从段基地址计算出来的二字节位移量。若还用了一个字节或字的立即数据，它将跟在位移量后面。

ADD

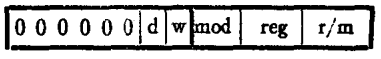
(Addition 加法)

操作：两个操作数的和被存放到目(左)操作数。

(DEST)←(LSRC) + (RSRC)

编码：

存贮器或寄存器操作数与寄存器操作数：



if d=0 then LSRC=REG, RSRC=EA, DEST=REG
else LSRC=EA, RSRC=REG, DEST=EA

时钟周期：(a) 寄存器加到寄存器 3
(b) 存贮器加到寄存器 9+EA
(c) 寄存器加到存贮器 16+EA

例：

(a) ADD AX, BX
ADD ,BX; 同上
ADD CX, DX
ADD DI, SI
ADD BX, BP

(b) ADD CX, MEM_WORD
ADD AX, BETA[SI]
ADD ,BETA[SI]; 同上
ADD DX, ALPHA[BX][DI]

(c) ADD GAMMA [BP][DI], BX
ADD BETA[DI], AX
ADD MEM_WORD, CX
ADD MEM_BYTE, BH

立即操作数加到累加器：

0 0 0 0 0 1 0	w	data	data if w=1

if w=0 then LSRC=AL, RSRC=data, DEST=AL
else LSRC=AX,RSRC=data, DEST=AX

时钟周期：4

例：

```
ADD AL, 3
ADD AX, 456
ADD AL, IMM_VAL_12
ADD AX, IMM_VAL_8529
ADD ,IMM_VAL_6AB9H; 目是 AX
```

立即操作数加到存贮器或寄存器操作数：

1 0 0 0 0 0	s	w	mod	0 0 0	r/m	data	data if s:w=01

LSRC=EA, RSRC=data, DEST=EA

时钟周期：(a) 立即数加到存贮器 17+EA
(b) 立即数加到寄存器 4

例：(a)
```
ADD MEM_WORD, 48
ADD GAMMA[DI], IMM_84
ADD DELTA[BX][SI], IMM_SENSOR_5
```
(b)
```
ADD BX, ORIG_VAL
ADD CX, STANDARD_COUNT
ADD DX, 1776
```

影响标志位：AF, CF, OF, PF, SF, ZF

说明：ADD 执行两个操作数的加法，并把结果返回到目操作数。

AND

(And: logical conjunction 逻辑与)

操作：两个操作数相与，仅当两个操作数的某一位都是 1 时，结果的这一位才是 1，其余的情况各位都是 0。把结果放回到目操作数。把进位和溢出标志复位为 0。

(DEST)←(LSRC) & (RSRC)
(CF)←0
(OF)←0

编码：

存贮器或寄存器操作数与寄存器操作数：

0 0 1 0 0 0	d	w	mod	reg	r/m

if d=1 then LSRC=REG, RSRC=EA, DEST=REG
else LSRC=EA, RSRC=REG, DEST=EA

时钟周期：(a) 寄存器与寄存器 3

(b) 存贮器与寄存器　9+EA

(c) 寄存器与存贮器　16+EA

例：

(a) AND AX, BX
AND ,BX；同上
AND CX, DI
AND BH, CL

(b) AND SI, MEM_NAME_WORD
AND DX, BETA[BX]
AND BX, GAMMA[BX][SI]
AND AX, ALPHA[DI]
AND ,ALPHA[DI]；同上
AND DH, MEM_BYTE

(c) AND MEM_NAME_WORD, BP
AND ALPHA[DI], AX
AND GAMMA[BX][DI], SI
AND MEM_BYTE, AL

立即操作数与累加器：

0 0 1 0 0 1 0	w	data	data if w=1

if w=0 then LSRC=AL, RSRC=data, DEST=AL
else LSRC=AX, RSRC=data, DEST=AX

时钟周期：立即数与寄存器　4

例：AND AL, 7AH
AND AH, 0EH
AND AX, IMM_VAL_MASK3

立即操作数与存贮器或寄存器操作数：

1 0 0 0 0 0 0	w	mod	1 0 0	r/m	data	data if w=1

LSRC=EA, RSRC=data, DEST=EA

时钟周期：(a) 立即数与寄存器　4

(b) 立即数与存贮器　17+EA

例：

(a) AND BL, 10011110B
AND CH, 3EH
AND DX, 7A46H
AND SI, 987

(b) AND MEM_WORD, 7A46H
AND MEM_BYTE, 46H
AND GAMMA[DI], IMM_MASK14

AND CHI_BYTE[BX][SI], 11100111B

影响标志位：CF, OF, PF, SF, ZF

无定义标志位：AF

说明：AND对两个操作数执行按位逻辑与，并把结果送回到目操作数中。

CALL

(Call a procedure 调用过程)

操作：如果这是一个段间调用，把堆栈指示器内容减2，把CS寄存器的内容推入堆栈。再把双字段间指示器的第2个字(段基地址)装入CS寄存器。然后把堆栈指示器的内容减2，并把指令指示器的内容推入堆栈。最后一步是用目偏移量即过程第一条指令的偏移量，替换IP的内容。段内或组内调用只做步骤2，3，4。

1) if Inter-Segment then
   (SP) ← (SP) −2
   ((SP) +1: (SP)) ← (CS)
   (CS) ←SEG
2) (SP) ← (SP) −2
3) ((SP) +1: (SP)) ← (IP)
4) (IP) ←DEST

编码：

段内或组内直接：

1 1 1 0 1 0 0 0	disp-low	disp-high

DEST= (IP) +disp

时钟周期：13+EA

例：

```
CALL NEAR_LABLE
CALL NEAR_PROC
```

段间直接：

1 0 0 1 1 0 1 0	offset-low	offset-high	seg-low	seg-high

DEST=offset, SEG=seg

时钟周期：20

例：

```
CALL FAR_LABLE
CALL FAR_PROC
```

段间间接：

1 1 1 1 1 1 1 1	mod	0 1 1	r/m

DEST= (EA), SEG= (EA+2)

时钟周期：29+EA

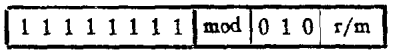
例：

```
CALL DWORD PTR [BX]
```

CALL DWORD PTR VARIABLE_NAME[SI]

CALL MEM_DOUBLE_WORD

段内或组内间接：



DEST = (EA)

时钟周期：11

例：

```
CALL WORD PTR [BX]
CALL WORD PTR VARIABLE_NAME
CALL WORD PTR [BX][SI]
CALL WORD PTR [DI]
CALL WORD PTR VARIABLE_NAME [BP][SI]
CALL MEM_WORD
CALL BX
CALL CX
```

影响标志位：无

说明：CALL 把下一条指令的偏移地址推入堆栈（在段间调用的情况下，CS 段寄存器首先被推入），然后把控制转移到目标操作数。

直接调用和跳转，只能对与 CS 相关的标号，而不是变量进行，除非在指令或在目的标号中说明是 FAR，否则总是假定为 NEAR。

如上述间接调用的例子所示，通过变量的调用，可以使用 PTR 算符来指示要被 NEAR 调用所使用的一个字，或要被对 FAR 标号或过程调用所使用的两个字。间接调用使用字寄存器（在方括号内）是 NEAR 调用所必须的。

除非使用 BP 或使用一个超越前缀，在寄存器间接调用中隐含使用的段寄存器是 DS。隐含的段寄存器用来构成地址，该地址含有调用目标的偏移量（和段基地址，如是一个“长”调用的话）。若使用了 BP，则 SS 是所使用的段寄存器。但是，若明确地指定一个段前缀字节，如 CALL WORD PTR ES:[BP][DI]，则就使用说明的寄存器（这里是 ES）。通过变量、或地址表达式、来间接调用的隐含的段寄存器，是由在源程序中的地址表达式和适当的 ASSUME 伪指令决定的。

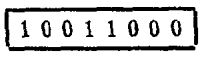
当使用 CALL 来转移控制时，隐含着一个 RET_{urn}。使用间接调用，必须确保 CALL 的类型和 RET_{urn} 的类型相匹配，否则会出现难以跟踪的错误。

CBW

(Convert byte to word 把字节转换为字)

操作：若累加器的低字节（AL）小于 80H，则使 AH 为 0。否则 AH 置为 0FFH。这等效于用 AL 的第 7 位代替 AH 中的所有位。

编码：



时钟：2

例：CBW

影响标志位：无

说明：CBW(把字节转换为字)执行一个 AL 寄存器到 AH 寄存器的符号扩展。

CLC

(Clear carry flag 清除进位标志)

操作：把进位标志复位为 0。

(CF)←0

编码：

1	1	1	1	1	0	0	0

时钟周期：2

例：CLC

影响标志位：CF

说明：CLC 清除 CF 标志。

CLD

(Clear direction flag 清除方向标志)

操作：把方向标志复位为 0。

(DF)←0

编码：

1	1	1	1	1	1	0	0

时钟周期：2

例：CLD

影响标志位：DF

说明：CLD 清除 DF 标志，使串操作把操作数指示器自动增值。

CLI

(Clear interrupt flag 清除中断标志)

操作：把中断标志复位为 0。

(IF)←0

编码：

1	1	1	1	1	0	1	0

时钟周期：2

例：CLI

影响标志位：IF

说明：CLI 清除 IF 标志，禁止出现在 8086 INTR 引脚上的可屏蔽中断(出现在 NMI 引脚上的不可屏蔽中断没有被禁止)。

CMC

(Complement carry flag 进位标志取反)

操作：若进位标志是0，则置位为1；若是1，则复位为0。

if (CF)=0 then (CF)←1 else (CF)←0

编码：

1 1 1 1 0 1 0 1

时钟周期：2

例：CMC

影响标志：CF

说明：CMC 把 CF 标志取反。

CMP

(Compare two operands 比较两个操作数)

操作：从目操作数(左)减去源操作数(右)。标志位改变但操作数不影响。

(LSRC)—(RSRC)

编码：

存贮器或寄存器操作数与寄存器操作数：

0 0 1 1 1 0	d	w	mod	reg	r/m

if d=1 then LSRC=REG, RSRC=EA

else LSRC=EA, RSRC=REG

时钟周期：(a) 寄存器与寄存器 3

(b) 存贮器与寄存器 9+EA

(c) 寄存器与存贮器 9+EA

例：

```
(a) CMP AX, DX
    CMP ,DX ;同上
    CMP SI, BP
    CMP BH, CL
(b) CMP MEM_WORD, SI
    CMP MEM_BYTE, CH
    CMP ALPHA[DI], DX
    CMP BETA[BX][SI], CX
(c) CMP DI, MEM_WORD
    CMP CH, MEM_BYTE
    CMP AX, GAMMA[BP][SI]
```

立即操作数与累加器：

0 0 1 1 1 1 0	w	data	data if w=1

if w=0 then LSRC=AL, RSRC=data

else LSRC=AX, RSRC=data

时钟周期：立即数与寄存器 4

例：

```
CMP AL, 6
CMP AL, IMM_VALUE_DRIVE 11
CMP AX, IMM_VAL_909
CMP ,999
CMP AX, 999  ;同上
```

立即操作数与存贮器或寄存器操作数：

1 0 0 0 0 0	s	w	mod	1 1 1	r/m	data	data if s:w=01

LSRC=EA, RSRC=data

时钟周期：(a) 立即数与寄存器 4

(b) 立即数与存贮器 17+EA

例：

```
(a) CMP BH, 7
    CMP CL, 19_IMM_BYTE
    CMP DX, IMM_DATA_WORD
    CMP SI, 798
(b) CMP MEM_WORD, IMM_DATA_BYTE
    CMP GAMMA[BX], IMM_BYTE
    CMP [BX][DI], 6ACEH
```

影响标志位：AF, CF, OF, PF, SF, ZF

说明：CMP (比较)指令执行对两个操作数的减法使标志位受到影响，但不返回结果。

源操作数 (右) 通常必须是和目操作数一样的类型，即字节或字。 唯一的例外是把一个立即数字节和存贮器字比较的时候，此时，该立即数字节要符号扩展(指令代码字节中的S=1)。

CMPS

(Compare byte string, compare word string 比较字节串,比较字串)

操作：从用 SI 作为变址器的左操作数中减去使用 DI 作为进入附加段的变址器的右操作数(这是 DI 变址的操作数作为右操作数的唯一的串指令)。只有标志位受到影响，操作数不受影响。若方向标志复位(0)，则 SI 和 DI 增值，(否则若 DF=1，则减值)。这样它们就指向被比较的串的下一个元素。对于字节串，增值1，对于字串增值2。

```
(LSRC) - (RSRC)
if (DF) =0 then
  (SI) ← (SI) +DELTA
  (DI) ← (DI) +DELTA
       else
```

(SI)←(SI)－DELTA
(DI)←(DI)－DELTA

编码：

1 0 1 0 0 1 1	w

if w=0 then LSRC=(SI), RSRC=(DI), DELTA=1(BYTE)
else LSRC=(SI)+1:(SI), RSRC=(DI)+1:(DI), DELTA=2(WORD)

时钟周期：22

例：

```
MOV SI, OFFSET STRING1
MOV DI, OFFSET STRING2
CMPS STRING1, STRING2
    ;在 CMPS 指令中命名的操作数，仅由汇编程序，用来验证类型和访问用到的当前段
    ;寄存器内容。CMPS 实际上只使用 SI 和 DI 来指示要比较内容的单元，而不用在源指
    ;令行中所给出的名。
```

影响标志位：AF, CF, OF, PF, SF, ZF

说明：CMPS 从由 SI 寻址的操作数中减去由 DI 寻址的字节(或字)操作数，影响标志位，但不返回结果。作为一个重复操作，它提供了对于两个串的比较。用适当的重复前缀，它能决定在哪一个串元素之后，该两个串变得不相等。从而在两个串之间建立次序。

注意，在这条指令中由 DI 变址的操作数是右操作数，并且这个操作数是使用 ES 寄存器寻址的，这个缺省规则不能被超越。

CWD

(Convert word to double word 把字变换为双字)

操作：把 AX 的最高位复制到整个 DX。

if (AX)<8000H then (DX)←0
else (DX)←0FFFFH

编码：

1 0 0 1 1 0 0 1

时钟周期：5

例：CWD

影响标志位：无

说明：CWD (把字转换为双字)把 AX 寄存器进行符号扩展到 DX 寄存器。

DAA

(Decimal adjust for addition) 10 进制数加法调整)

操作：若 AL 的低 4 位大于 9 或若辅助进位标志已经置位，则给 AL 加 6 并且置位 AF.若 AL 大于 9FH 或进位标志已经置位，则把 60H 加到 AL 并且把 CF 置位。

if ((AL)&0FH)>9 or (AF)=1 then

(AL)←(AL)+6
(AF)←1
if (AL)>9FH or (CF)=1 then
(AL)←(AL)+60H
(CF)←1

编码：

0 0 1 0 0 1 1 1

时钟周期：4

例：DAA

影响标志位：AF, CF, PF, SF, ZF

无定义标志位：OF

说明：DAA (10 进制数加法调整) 对两个压缩 10 进制操作数加法在 AL 中的结果进行修正，产生一个压缩的 10 进制和。

DAS

(Decimal adjust for substraction 10 进制数减法调整)

操作：若 AL 的低 4 位大于 9 或辅助进位标志已经置位，则从 AL 中减去 6 并且置位 AF。
若 AL 大于 9FH 或进位标志已经置位，则从 AL 中减去 60H 并且把 CF 置位。
if ((AL)&0 FH)>9 or (AF)=1 then
(AL)←(AL)−6
(AF)←1
if (AL)>9FH or (CF)=1 then
(AL)←(AL)−60H
(CF)←1

编码：

0 0 1 0 1 1 1 1

时钟周期：4

例：DAS

影响标志位：AF, CF, PF, SF, ZF。

无定义标志位：OF

说明：DAS (10 进制数减法调整) 对两个压缩 10 进制数相减在 AL 寄存器中的结果进行修正，产生一个压缩的 10 进制数差。

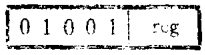
DEC

(Decrement destination by one 把目操作数减 1)

操作：把指定的操作数减 1
(DEST)←(DEST)−1

编码：

寄存器操作数：(字)



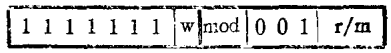
DEST=REG

时钟周期：2

例：

```
DEC AX
DEC DI
DEC SI
```

存贮器或寄存器操作数：



DEST=EA

时钟周期：寄存器　2

存贮器　15+EA

例：

```
DEC MEM_BYTE
DEC MEM_BYTE[DI]
DEC MEM_WORD
DEC ALPHA[BX][SI]
DEC BL
DEC CH
```

影响标志位：AF, OF, PF, SF, ZF

说明：DEC (减1)从操作数中减去1，并把结果返回该操作数。

DIV

(Division, unsigned, 无符号除)

操作：若除法产生一个比规定的寄存器所能容纳的值还要大的结果，则产生一个0类型中断。标志位被推入堆栈，IF 和 TF 被复位为0，CS 寄存器的内容也被推入堆栈。然后 CS 被填入在单元2中的字。当前的 IP 被推入堆栈，然后被填入在单元0中的字。这样这个序列包含了一个对于中断处理过程的长调用，该过程的段基地址和偏移量各自存放在单元2和0中。

若除法的结果能够放入到规定的寄存器中，则商存放在 AL 或 AX (对于字操作) 而余数则各自放在 AH 或 DX 中。

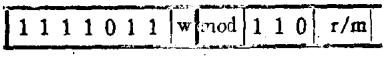
```
(temp)←(NUMR)
if (temp)/(DIVR)>MAX then the following, in sequence
        (QUO), (REM) undifined
        (SP)←(SP)−2
        ((SP)+1:(SP))←FLAGS
        (IF)←0
```

(TF)←0
(SP)←(SP)−2
((SP)+1:(SP))←(CS)
(CS)←(2) ;即存贮器单元 2 和 3 的内容。
(SP)←(SP)−2
((SP)+1:(SP))←(IP)
(IP)←(0) ;即单元 0 和 1 的内容。

else

(QUO)←(temp)/(DIVR)，这里 / 是无符号除法。
(REM)←(temp)%(DIVR)，这里%是无符号模除。

编码：



(a) if w=0 then NUMR=AX, DIVR=EA, QUO=AL, REM=AH, MAX=0FFH
(b) else NUMR=DX:AX, DIVR=EA, QUO=AX, REM=DX, MAX=0FFFFH

时钟周期： 8 位 90+EA
16 位 155+EA

例：

(a1) 一个字被一个字节除

```
MOV AX, NUMERATOR_WORD
DIV DIVISOR_BYTE ; 商在 AL 中,余数在 AH 中。
```

(a2) 一个字节被一个字节除

```
MOV AL, NUMERATOR_BYTE
CBW ;把在 AL 中的字节转换为在 AX 中的字。
DIV DIVISOR_BYTE ;商在 AL 中,余数在 AH 中。
```

(b1) 一个双字被一个字除

```
MOV DX, NUMERATOR_HI_WORD
MOV AX, NUMERATOR_LO_WORD ;商在 AX 中余数在 DX 中。
DIV DIVISOR_WORD
```

(b2) 一个字除以一个字

```
MOV AX, NUMERATOR_WORD
CWD 把字转换为双字。
DIV DIVISOR_WORD 商在 AX 中,余数在 DX 中。
```

注意： 上面的每一个存贮器操作数只要它们的类型一致，可以是任何变量或有效地址表达式。
例如，在(a1)中 NUMERATOR_WORD 可以由表达式
ARRAY_NAME[BX][SI]+67
来代替，只要 ARRAY_NAME 是 WORD 类型。类似地，
DIVISOR_BYTE 可以被 RATE_TABLE[BP][DI]
来代替，只要 RATE_TABLE 是 BYTE 类型的。

影响标志位：不产生有效的标志位

无定义标志位：AF, CF, OF, PF, SF, ZF

说明：DIV (除)执行一个对于放在累加器和它的扩展部分 (对于8位操作是 AL 和 AH,对于16位操作是 AX 和 DX) 中的双长度 NUMR 操作数，除以在指定的源操作数中的 DIVR 操作数的操作。它把单长度的商(QUO 操作数)返回到累加器(AL 或 AX)，把单长度的余数(REM 操作数)返回到累加器扩展部分(对8位操作是 AH，对16位操作是 DX)。若商大于 MAX (就如试图除以0一样)，则 QUO 和 REM 没有定义并产生一个0类型中断。在任何 DIV 操作中没有定义标志位。非整数商被舍入成整数。

ESC

(Escape 交权)

操作：

if mod≠11 then data bus←(EA)

if mod=11, no operation.

编码：

11011	x	mod	x	r/m

时钟周期：7+EA

例：

ESC EXTERNAL_OPCODE, ADDRESS

这个操作码是一个6位数，它被分裂成两个3位字段如上面X所示。

影响标志位：无

说明：ESC 指令提供了一种机能，使其他处理器能从 8086 指令流中取得它们的指令，并使用 8086 的寻址方式。8086处理器除了访问一个存贮器操作数并把它放在总线上以外，对 ESC 指令不做任何操作。

HLT

(Halt 停机)

操作：无

编码：

11110100

时钟周期：2

例：HLT

影响标志位：无

说明：HLT 指令使 8086 处理器进入它的停机状态。该停机状态由一个允许的外部中断或复位清除。

IDIV

(Integer division, signed 带符号整数除)

操作：若除法结果是一个大于规定的寄存器所能容纳的数，则产生一个类型0中断。标志位

被推入堆栈。IF 和 TF 被复位为 0。CS 寄存器的内容被推入堆栈，随后由在单元 2 的字填入。当前的 IP 被推入堆栈，随后由在单元 0 的字填入。这样，这个序列包含了一个对于中断处理程序的长调用，该过程的段基地址和偏移量各自存放在单元 2 和 0 若除法结果能放入规定的寄存器，则商放在 AL 或 AX（对字操作数）中，而余数则分别放到 AH 或 DX 中。

(temp)←(NUMR)
if (temp)/(DIVR)>0 and (temp)/(DIVR)>MAX
or (temp)/(DIVR)<0 and (temp)/(DIVR)<0−MAX−1
then
(QUO), (REM) undefined
(SP)←(SP)−2
((SP)+1:(SP))←FLAGS
(IF)←0
(TF)←0
(SP)←(SP)−2
((SP)+1:(SP))←(CS)
(CS)←(2)
(SP)←(SP)−2
((SP)+1:(SP))←(IP)
(IP)←(0)
else
(QUO)←(temp)/(DIVR)，这里/是有符号除。
(REM)←(temp)%(DIVR)，这里%是有符号模除。

编码：

1 1 1 1 0 1 1	w	mod	1 1 1	r/m

(a) if w=0 then NUMR=AX, DIVR=EA, QUO=AL, REM=AH, MAX=7FH
(b) else NUMR=DX:AX, DIVR=EA, QUO=AX, REM=DX,
MAX=7FFFH

时钟周期：8位 112+EA
16位 177+EA

例：

```
(a) MOV AX, NUMERATOR_WORD[BX]
    IDIV DIVISOR_BYTE[BX]
(b) MOV DX, NUM_HI_WORD
    MOV AX, NUM_LO_WORD
    IDIV DIVISOR_WORD[SI]
```

影响标志位：AF, CF, OF, PF, SF, ZF
无定义标志位：全体。

说明：IDIV（整数除）对放在累加器和它的扩展部分（对8位操作是AL和AH，对16位操作是AX和DX）中的双长度NUMR操作数，执行一个除以在指定的源操作数中的DIVR操作数。它把单长度的商（QUO操作数）返回到累加器（AL或AX），把单长度的余数（REM操作数）返回到累加器扩展部分（对8位操作是AH，对16位操作是DX）。若商是正的，并大于MAX，或商是负的，并小于（0-MAX-1）（如同试图除以0一样），则QUO和REM无定义，并产生一个类型0中断。标志位在任何除法操作中都没有定义。IDIV对非整数商进行舍入，并返回一个和被除数具有相同符号的余数。

IMUL

（Integer multiply accumulator by register-or-memory; signed 带符号整数累加器乘以寄存器或存贮器）

操作：累加器（若是字节为AL，若是字为AX）乘以指定的操作数。若结果的高半是低半的符号扩展，则把进位和溢出标志复位，否则把它们置位。

(DEST)←(LSRC)*(RSRC) 这里*是带符号乘

if (EXT)=sign-extentension of (LOW) then (CF)←0

else (CF)←1 ;

(OF)←(CF)

编码：

1 1 1 1 0 1 1	w	mod	1 0 1	r/m

(a) if w=0 then LSRC=AL, RSRC=EA, DEST=AX, EXT=AH, LOW=AL

(b) else LSRC=AX, RSRC=EA, DEST=DX:AX, EXT=DX, LOW=AX

时钟周期：8位 90+EA

16位 144+EA

例：

(a) MOV AL, LSRC_BYTE

IMUL RSRC_BYTE ;结果在AX中

(b1) MOV AX, LSRC_WORD

IMUL RSRC_WORD ;结果的高半在DX中，低半在AX中。

(b2) 以一个字节乘以一个字

MOV AL, MUL_BYTE

CBW ;把在AL中的字节变换成在AX中的字。

IMUL RSRC_WORD; 高半结果在DX中，低半结果在AX中。

注意：上面的任何存贮器操作数可以是一个正确的TYPE的变址地址表达式，例如若ARRAY的类型是BYTE，则LSRC_BYTE可以是ARRAY[SI]，若TABLE的类型是WORD则RSRC_WORD可以是TABLE[BX][DI]。

影响标志位：CF, OF

无定义标志位：AF, PF, SF, ZF.

说明：IMUL（整数乘）执行一个累加器（AL或AX）和源操作数的带符号乘，返回一个双长度的结果给累加器和它的扩展部分（对8位操作是AL和AH，对16位操作是AX

和 DX)。当结果的高半不是结果低半的符号扩展时，把 CF 和 OF 置位。

IN

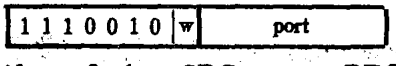
(Input byte and input word 输入字节和字)

操作：累加器的内容被指定的转接口内容所取代。

(DEST)←(SRC)

编码：

固定转接口：



if w=0 then SRC=port, DEST=AL

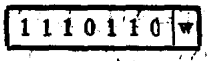
else SRC=port+1:port, DEST=AX

时钟周期：10

例：

```
IN AX, WORD_PORT ;把字输入到 AX
IN AL, BYTE_PORT ;把字节输入到 AL
;输入的目必须是 AX 或 AL，并且必须说明以使汇编程序知道输入的类型。转接口名必
;须是介于 0 和 255 之间的立即数值。 如在上面使用寄存器名 DX，它必须事先填入必
;须的转接口地址。
```

变量转接口：



if w=0 then SRC=(DX), DEST=AL

else SRC=(DX)+1:(DX), DEST=AX

时钟周期：8

例：

```
IN AX, DX ;给 AX 输入一个字
IN AL, DX ;给 AL 输入一个字节
```

影响标志位：无

说明：IN 从一个输入转接口中，把一个字节(或字)传送给 AL 寄存器(或 AX 寄存器)。该转接口或是用在命令行中的一个数据字节指定，这种方法允许固定地访问转接口 0 到 255，或是用一个在 DX 寄存器中的转接口数指定，这种方法允许可变地访问 64K 转接口。

INC

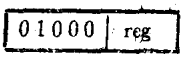
(Increment destination by1, 把目操作数加 1)

操作：把指定的操作数加 1。最高位不产生进位。

(DEST)←(DEST)+1

编码：

寄存器操作数：(字)



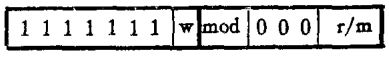
DEST=REG

时钟周期：2

例：

```
INC AX
INC DI
```

存贮器或寄存器操作数：



DEST=EA

时钟周期：

(a) 寄存器　2

(b) 存贮器　15+EA

例：

```
(a) INC CX
    INC BL
(b) INC MEM_BYTE
    INC MEM_WORD[BX]
    INC BYTE PTR [BX] ;字节在数据段的偏移量为[BX]的单元中
    INC ALPHA[DI][BX]
    INC BYTE PTR [SI][BP] ;字节在堆栈段的偏移量为 [SI+BP] 的单元中.
    INC WORD PTR [BX] ;把在数据段中偏移量为[BX]的字加 1，这样能取得进入
                      ;第 8 位的进位.
```

影响标志位：AF, OF, PF, SF, ZF

说明：INC (加 1) 执行一个操作数和 1 的加法，并把结果返回给操作数.

INT

(Interrupt 中断)

操作：把堆栈指示器 SP 减 2，把所有标志位推入堆栈. 然后把中断和自陷标志复位. 再把 SP 减 2，并把 CS 寄存器的当前内容推入堆栈，随后把中断向量双字的较高地址字，即该中断类型的中断处理过程的段基地址填入 CS.

SP 再减 2，并把指令指示器的当前值推入堆栈，随后用中断向量的较低地址字填入IP，该中断向量的绝对地址是 TYPE*4. 这样完成了一个对于过程的段间("长")调用，该过程将处理这种类型的中断.

```
(SP) ← (SP) − 2
((SP) + 1 : (SP)) ← FLAGS
(IF) ← 0
(TF) ← 0
(SP) ← (SP) − 2
((SP) + 1 : (SP)) ← (CS)
(CS) ← (TYPE*4 + 2)
```

(SP)←(SP)−2

((SP)+1:(SP))←(IP)

(IP)←(TYPE*4)

编码：

1 1 0 0 1 1 0	v	type if v=1

(a) if v=0 then TYPE=3

(b) else TYPE=type

时钟周期：52

例：

```
(a) INT  3   ;1 字节指令：11001100
(b) INT  2   ;2 字节指令：11001101 00000010
    INT 67   ;2 字节指令：11001101 01000011
    IMM_44 EQU 44
    INT IMM_44 ; 2 字节：11001101 00101100
```

注意：操作数必须是立即数而不能引用寄存器或存贮器。

影响标志位：IF，TF

说明：INT 推入标志寄存器(如象 PUSHF 一样)，清除 TF 和 IF 标志，并通过 256 个向量元素中的任一个，用间接调用转移控制。这种指令的一字节形式产生一个类型 3 中断。

INTO

(Interrupt if overflow 若溢出就中断)

操作：若溢出标志是 0，不操作。若 OF 是 1，则堆栈指示器减 2 并把所有的标志保护进堆栈。把自陷和中断标志复位，SP 再减 2 并把 CS 的内容推入堆栈。然后把类型 4 中断向量双字的第二个字(段基地址)填入 CS。

SP 再减 2，把当前的指令指示器(指着 INTO 指令后的下一条指令)推入堆栈。然后把类型 4 中断向量双字的第一个字填入 IP，该字在绝对地址 16 (10H)。这个字是处理类型 4 中断过程的偏移量。段基地址已经被放入 CS。这样就完成了对该过程的“长”调用。

```
if (OF)=1 then
    (SP)←(SP)−2
    ((SP)+1:(SP))←FLAGS
    (IF)←0
    (TF)←0
    (SP)←(SP)−2
    ((SP)+1:(SP))←(CS)
    (CS)←(12H)
    (SP)←(SP)−2
    ((SP)+1:(SP))←(IP)
```

(IP)←(10H)

编码：

1 1 0 0 1 1 1 0

时钟周期：52

例：INTO

影响标志位：无

说明：INTO 推入标志寄存器(如在 PUSHF 中所做的一样)，清除 TF 和 IF 标志。若 OF 标志置位(溢出)，则通过向量元素 4 (位置 10H)用一个间接调用转移控制。若 OF 标志被清除，则不进行操作。

IRET

(Interrupt return 中断返回)

操作：用在堆栈顶的字填入指令指示器，然后堆栈指示器加 2，并用现在在堆栈顶的字填入 CS 寄存器。这样就把控制返回到了遇到中断的那一点。

SP 再加 2，从现在在堆栈顶的字的规定位中恢复标志(和 POPF 相同)，SP 再次加 2。

(IP)←((SP)+1:(SP))

(SP)←(SP)+2

(CS)←((SP)+1:(SP))

(SP)←(SP)+2

FLAGS←((SP)+1:(SP))

(SP)←(SP)+2

编码：

1 1 0 0 1 1 1 1

时钟周期：24

例：IRET

影响标志位：全部

说明：IRET 把控制转移到以前由中断操作保护的返回地址，并且恢复保护的标志寄存器(如在 POPF 中一样)。

JA

JNBE 和 JA (Jump if not below nor equal, or jump if above 若不低于等于，或高于就跳转)

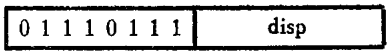
操作：若进位标志和 0 标志都是 0，则把从这条指令的末尾到目的标号之间的距离加到指令指示器来实现跳转。

若 (CF)=1 或 (ZF)=1，则不跳转。

if (CF) | (ZF) =0 then

(IP)←(IP)+disp(sign-extended to 16-bits)

编码：



时钟周期：跳转　8
　　不跳转 4

例：

JA TARGET_LABEL
JNBE TARGET_LABEL

影响标志位：无

说明：JNBE（或 JA）在不低于、等于（或高于）时，把控制转移到目标操作数。

注意：目的标号必是介于这条指令的－128 与＋127 字节之间。“高于”和“低于”是指两个无符号值之间的关系。“大于”和“小于”是指两个带符号值之间的关系。

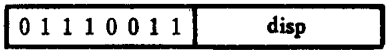
JAE

JNB 和 JAE (Jump if not below, or jump if above or equal 若不低于就跳转，或若高于等于就跳转)

操作：若进位标志是 0，则把从这条指令的末尾到目的标号之间的距离，加到指令指示器来实现跳转。若 (CF)＝1，就不跳转。

if (CF)＝0 then
(IP)←(IP)＋disp (sign-extended to 16-bits)

编码：



时钟周期：跳转　8
　　不跳转 4

例：

JNB TARGET_LABEL
JAE TARGET_LABEL

影响标志位：无

说明：JNB（或 JAE）在不低于（或高于等于）时，把控制转移到目标操作数。

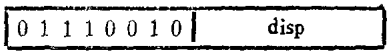
JB

JB 和 JNAE (Jump if below, or jump if not above nor equal 若低于就跳转，或若不高于等于就跳转)

操作：若进位标志是 1，则把从该条指令末尾到目的标号之间的距离，加到指令指示器来实现跳转。若 (CF)＝0，则不跳转。

if (CF)＝1 then
(IP)←(IP)＋disp (sign-extended to 16-bits)

编码：



时钟周期：跳转　8

不跳转 4

例：

JB TARGET_LABEL

JNAE TARGET_LABEL

影响标志位：无

说明：JB（或 JNAE）在低于(或不高于等于)时，把控制转移到目标操作数。

JBE

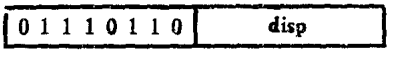
JBE 和 JNA (Jump if below or equal, or jump if not above 若低于等于就跳转，或不高于就跳转)

操作：若进位标志或 0 标志置位，则把从这条指令的末尾到目的标号之间的距离，加到指令指示器来实现跳转。若 (CF)=0 和 (ZF)=0，则不跳转。

if (CF) | (ZF)=1 then

(IP)←(IP)+disp (sign-extended to 16-bits)

编码：



时钟周期：跳转　8

不跳转 4

例：

JBE TARGET_LABEL

JNA TARGET_LABEL

影响标志位：无

说明：JBE（或 JNA）在低于等于(或不高于)时，把控制转移到目的标号。

JCXZ

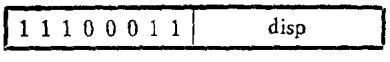
(Jump if CX is zero 若 CX 是 0 就跳转)

操作：若计数寄存器 CX 是 0，则把从这条指令的末尾到目的标号之间的距离，加到指令指示器来实现跳转。若 CX 的内容不是 0，则不跳转。

if (CX)=0 then

(IP)←(IP)+disp (sign-extended to 16-bits)

编码：



时钟周期：跳转　9

不跳转 5

例：JCXZ TARGET_LABEL

影响标志位：无

说明：JCXZ（在 CX 是 0 时跳转）在 CX 寄存器是 0 时，把控制转移到目标操作数。

JE

JE 和 JZ (Jump if equal, jump if zero 若相等就跳转，若是 0 就跳转)

操作：若上一个影响 0 标志的操作给出一个 0 结果，则(ZF)将是 1。若(ZF)=1，则把从这条指令的末尾到目的标号之间的距离，加到指令指示器来实现跳转。若(ZF)=0，则不跳转。

if (ZF)=1 then

(IP)←(IP)+disp (sign-extended to 16-bits)

编码：

01110100	disp

时钟周期：跳转 8

不跳转 4

例：

```
1)      CMP CX, DX
        JE LAB2
        INC CX
LAB2:
; 仅当 CX≠DX 时才会发生把 CX 加 1
2)      SUB AX, BX
        JZ EXACT
   ; 若结果是 0，即 AX=BX 才发生跳转。
   :
EXACT:
```

影响标志位：无

说明：在相等(或 0)时，JE (或 JZ)把控制转移到目的操作数。

JG

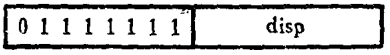
JNLE 和 JG (Jump if not less nor equal, or jump if greater 若不小于等于就跳转，或若大于就跳转)

操作：若 0 标志复位，并且符号标志等于溢出标志 (即，都是 0 或都是 1)，则把从这条指令的末尾到目的标号之间的距离，加到指令指示器来实现跳转。若 (ZF)=1 或 (SF)≠(OF)，则不发生跳转。

if ((SF) ‖ (OF)) |(ZF) =0 then

(IP)←(IP)+disp (sign-extended to 16-bits)

编码：



时钟周期：跳转 8
不跳转 4

例：

JG TARGET_LABEL
JNLE TARGET_LABEL

影响标志位：无

说明：JNLE（或 JG）在不小于等于（或大于）时，把控制转移到目标操作数。

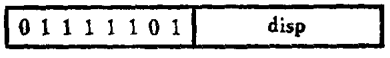
JGE

JNL 和 JGE (Jump if not less, or jump if greater or equal 若小于就跳转或若大于等于就跳转)

操作：若符号标志等于溢出标志，则把从这条指令末尾到目的标号之间的距离，加到指令指示器来实现跳转。若 (SF)≠(OF) 则不跳转。

if (SF)||(OF)=0 then
(IP)←(IP)+disp (sign-extended to 16-bits)

编码：



时钟周期：跳转 8
不跳转 4

例：

JGE TARGET_LABEL
JNL TARGET_LABEL

影响标志位：无

说明：JNL（或 JGE）在不小于（或大于等于）时，把控制转移到目标操作数。

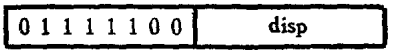
JL

JL 和 JNGE (Jump on less, or jump on not greater nor equal 小于就跳转，或不大于等于就跳转)

操作：这种跳转仅当符号标志不等于溢出标志时，即(SF)≠(OF)时才发生。(SF)异或(OF)=1 是相同的意思。若 (SF)≠(OF)，则把从这条指令的末尾到目的标号之间的距离，加到指令指示器来实现跳转。若 (SF)=(OF)，就不跳转。

if (SF)||(OF)=1 then
(IP)←(IP)+disp (sign-extended to 16-bits)

编码：



时钟周期：跳转　8
　　不跳转 4

例：

JL TARGET_LABEL
JNGE TARGET-TABEL

影响标志位：无

说明：在小于(或不大于等于)时，JL (或 JNGE) 把控制转移到目标操作数。

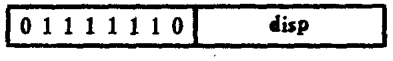
JLE

JLE 和 JNG (Jump if less or equal, or jump if not greater 若小于、等于就跳转，或若不大于就跳转)

操作：若 0 标志置位，或若符号标志不等于溢出标志，则把从这条指令末尾到目的标号之间的距离，加到指令指示器来实现跳转。若(ZF)=0，并且(SF)=(OF)，则不跳转。

if ((SF)‖(OF))|(ZF)=1 then
(IP)←(IP)+disp (sign-extended to 16-bits)

编码：



时钟周期：跳转　8
　　不跳转　4

例：

JLE TARGET_LABEL
JNG TARGET_LABEL

影响标志位：无

说明：JLE(或 JNG)在小于等于(或不大于)时，把控制转移到目标操作数。

JMP

(Jump 跳转)

操作：在所有的段间跳转和段内(或组内)间接跳转时，指令指示器的内容被目标的偏移量所替代。

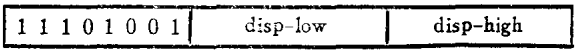
当是段内或组内直接跳转时，把从这条指令末尾到目的标号之间的距离加到 IP。

段间跳转，首先用跟在指令后面的第 2 个字(直接)，或用所指定的数据地址中的第 2 个字(间接)来替代 CS 的内容。

if Inter-Segment then (CS)←SEG
(IP)←DEST

编码：

段内或组内直接：

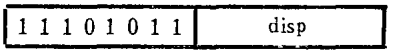


DEST = (IP) + disp

时钟周期： 7

例：JMP NEAR-LABEL

段内直接的短形式：



DEST = (IP) + disp sign extended to 16 bits

时钟周期： 7

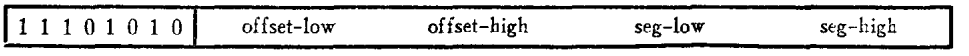
例：

JMP TARGET-LABEL

JMP SHORT NEAR_LABEL

注意：目的标号，必须是在这条指令的－128 到＋127 字节之间。

段间直接：



DEST=offset SEG=seg

时钟周期： 7

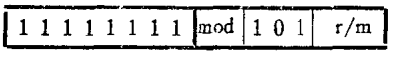
例：

JMP LABEL-DECLARED-FAR

JMP FAR PTR LABEL-NAME

JMP FAR PTR NEAR-LABEL

段间间接：



DEST = (EA)， SEG = (EA + 2)

时钟周期：16+EA

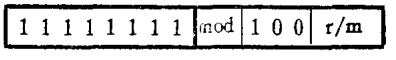
例：

JMP VAR-DOUBLEWORD

JMP DWORD PTR [BX][SI]

JMP ALPHA[BP][DI]

段内或组内间接：



DEST = (EA)

时钟周期：7+EA

例：

JMP TABLE[BX]

JMP WORD PTR [BX][DI]

```
JMP BETA_WORD
JMP AX
JMP SI
JMP BP
;这些指令用指定的寄存器的内容来替代指令指示器的值，这种跳转使用了相对于 CS
;的偏移量;因而能直接跳转到目标字节. 这不同于段内直接跳转,后者是自相关的,它
;通过给 IP 加上一个数而实现转移.
```

影响标志位：无

说明：JMP 把控制转移到目标操作数.

跳转指令,总是与在 CS 寄存器中的段基始地址有关的.

直接跳转直接使用跟随在指令字节之后的偏移量(和段基地址,若是"长"的话)字节.

间接跳转使用由跟随在指令字节之后的字节寻址的单元的内容.

JNA

JNA 和 JBE (Jump if below or equal, or jump if not above 若低于等于就跳转,或不高于就跳转)

操作：若进位标志或零标志置位，则把从指令末尾到目的标号之间的距离加到指令指示器来实现跳转. 若(CF)=0 和 (ZF)=0, 则不跳转.

```
if (CF) | (ZF)=1 then
  (IP)←(IP)+disp(sign-extended to 16-bits)
```

编码：

0 1 1 1 0 1 1 0	disp

时钟周期：跳转 8

不跳转 4

例：

```
JBE TARGET_LABEL
JNA TARGET_LABEL
```

影响标志位：无

说明：JBE (或 JNA) 在低于等于(或不高于)时,把控制转移到目标操作数.

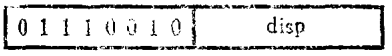
JNAE

JNAE 和 JB (Jump if below, or jump if not above nor equal 若低于就跳转,或若不不高于等于就跳转)

操作：若进位标志是1, 则把从这条指令末尾到目的标号之间的距离,加到指令指示器来实现跳转. 若(CF)=0,则不跳转.

```
if (CF)=1 then
  (IP)←(IP) +disp (sign-extended to 16-bits)
```

编码：



时钟周期：跳转　8

不跳转　4

例：

JB TARGET_LABEL

JNAE TARGET_LABEL

影响标志位：无

说明：JB（或 JNAE）在低于(或不高于等于)时，把控制转移到目标操作数.

JNB

JNB 和 JAE (Jump if not below, or jump if above or equal 若不低于就跳转，或若高于等于就跳转)

操作：若进位标志是 0，则把从这条指令末尾到目的标号之间的距离，加到指令指示器来实现跳转. 若 (CF)=1，则不跳转.

if (CF)=0 then

(IP)←(IP)+disp (sign-extended to 16-bits)

编码：

时钟周期：跳转　8

不跳转　4

例：

JNB TARGET_LABEL

JAE TARGET_LABEL

影响标志位：无

说明：JNB（或 JAE)在不低于(或大于等于)时，把控制转移到目标操作数.

JNBEJ

JNBE和JA (Jump if not below nor equal, or jump if above 若不低于等于就跳转，或高于就跳转)

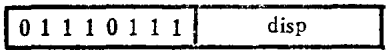
操作：若进位标志和 0 标志都不置位，则把从这条指令的末尾到目的标号之间的距离，加到指令指示器来实现跳转.

若 (CF)=1 或 (ZF)=1，就不跳转.

if (CF) | (ZF)=0 then

(IP)←(IP)+disp (sign -extended to 16-bits)

编码：



时钟周期：跳转　8

不跳转 4

例：

JNBE TARGET_LABEL

JA TARGET_LABEL

影响标志位：无

说明：JNBE (或 JA)在不低于等于(或高于)时，把控制转移到目标操作数。

JNE

JNE 和 JNZ (Jump if not equal, or jump if not zero 若不等于就跳转，或若不是0就跳转)

操作：若0标志复位，则把从这条指令末尾到目的标号之间的距离，加到指令指示器来实现跳转。若 (ZF)=1 则不跳转。

if (ZF)=0 then

(IP)←(IP)+disp(sign-extended to 16-bits)

编码：

0 1 1 1 0 1 0 1	disp

时钟周期：跳转 8

不跳转 4

例：

JNE TARGET_LABEL

JNZ TARGET_LABEL

影响标志位：无

说明：JNE (或 JNZ)在不等于(或不是零)时，把控制转移到目标操作数。

JNG

JNG 和 JLE (Jump if not greater, or jump if less or equal 若不大于就跳转，或若小于等于就跳转)

操作：若0标志置位，或若符号标志不等于溢出标志，则把从这条指令末尾到目的标号之间的距离，加到指令指示器来实现跳转。若 (ZF)=0，并且 (SF)=(OF)，则不跳转。

if ((SF) ‖ (OF)) ‖ (ZF)=1 then

(IP)←(IP)+disp (sign-extended to 16-bits)

编码：

0 1 1 1 1 1 1 0	disp

时钟周期：跳转 8

不跳转 4

例：

JLE TARGET_LABEL
JNG TARGET_LABEL

影响标志位：无

说明：JLE（或 JNG）在小于等于(或不大于)时，把控制转移到目标操作数。

JNGE

JL 和 JNGE (Jump if less, or jump if not greater nor equal 若小于就跳转，或若不大于等于就跳转)

操作：若符号标志不等于溢出标志，则把从这条指令的末尾到目的标号之间的距离，加到指令指示器来实现跳转。

若 (SF) = (OF)，则不跳转。

if (SF) ‖ (OF) = 1 then

(IP) ← (IP) + disp (sign-extended to 16-bits)

编码：

0 1 1 1 1 1 0 0	disp

时钟周期：跳转 8

不跳转 4

例：

JL TARGET_LABEL
JNGE TARGET_LABEL

影响标志位：无

说明：JL（或 JNGE）在小于(或不大于等于)时，把控制转移到目标操作数。

JNL

JNL 和 JGE (Jump if not less, or jump if greater or equal 若不小于就跳转，或若大于等于就跳转)

操作：若符号标志和溢出标志相等，则把从这条指令的末尾到目的标号之间的距离，加到指令指示器来实现跳转。

若 (SF) ≒ (OF)，则不跳转

if (SF) ‖ (OF) = 0 then

(IP) ← (IP) + disp (sign-extended to 16-bits)

编码：

0 1 1 1 1 1 0 1	disp

时钟周期：跳转 8

不跳转 4

例：

JGE TARGET_LABEL
JNL TARGET_LABEL

影响标志位：无

说明：JNL（或 JGE）在不小于(或大于等于)时，把控制转移到目标操作数。

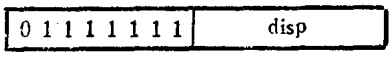
JNLE

JNLE 和 JG (Jump if not less nor equal, or jump if greater 若不小于等于就跳转，或若大于就跳转)

操作：若 0 标志复位，并且符号标志等于溢出标志（即，都是 0 或 1），则把从这条指令末尾到目的标号之间的距离，加到指令指示器来实现跳转。若 (ZF)=1 或 (SF)≠(OF)，则不跳转。

if ((SF) ‖ (OF)) | (ZF)=0 then
(IP)←(IP)+disp (sign-extended to 16-bits)

编码：



时钟周期：跳转 8
不跳转 4

例：
JG TARGET_LABEL
JNLE TARGET_LABEL

影响标志位：无

说明：JNLE（或 JG）在不小于等于(或大于)时，把控制转移到目标操作数。

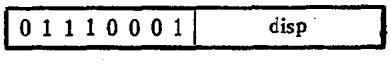
JNO

(Jump on not overflow 不溢出跳转)

操作：若溢出标志是 1，则不跳转。若 (OF)=0，则把从这条指令末尾到目的标号之间的距离，加到指令指示器来实现跳转。

if (OF)=0 then
(IP)←(IP)+disp (sign-extended to 16-bits)

编码：



时钟周期：跳转 8
不跳转 4

例：JNO TARGET_LABEL

影响标志位：无

说明：JNO 在无溢出时，把控制转移到目的操作数。

JNP

JNP 和 JPO (Jump on no parity or jump if parity odd 奇偶性为非偶跳转，或奇偶性为奇跳转)

操作：若奇偶标志是1，意味着上一次影响 PF 的操作的结果的奇偶性为偶，则不跳转。若(PF)=0，则把从这条指令的末尾到目的标号之间的距离，加到指令指示器来实现跳转。

if (PF)=0 then

(IP)←(IP)+disp (sign-extended to 16-bits)

编码：

01111011	disp

时钟周期：跳转 8

不跳转 4

例：

(1) JNP TARGET_LABEL

(2) JPO TARGET_LABEL

影响标志位：无

说明：JNP (或 JPO) 在奇偶性为非偶(或奇偶性为奇)时，把控制转移到目标操作数。

JNS

(Jump on not sign, jump if positive 无符号跳转，为正跳转)

操作：若符号标志没有置位，则把从这条指令末尾到目的标号之间的距离，加到指令指示器来实现跳转。若 (SF)=1，则不跳转。

if (SF)=0 then

(IP)←(IP)+disp (sign-extended to 16-bits)

编码：

01111001	disp

时钟周期：跳转 8

不跳转 4

例：

JNS TARGET_LABEL

影响标志位：无

说明：JNS 在无符号时把控制转移到目标操作数。

JNZ

JNE 和 JNZ (Jump if not equal,or jump if not zero 若不等于就跳转，或若不是0就跳转)

操作：若0标志复位，则把从这条指令末尾到目的标号之间的距离，加到指令指示器来实现跳转。若 (ZF)=1，则不跳转。

if (ZF)=0 then

(IP)←(IP)+disp(sign-extended to 16-bits)

编码：

01110101	disp

时钟周期：跳转　8
　　不跳转　4

例：

(1) JNE TARGET_LABEL

(2) JNZ TARGET_LABEL

影响标志位：无

说明：JNE（或 JNZ）在不相等(或不是0)时，把控制转移到目标操作数。

JO

(Jump on overflow 溢出就跳转)

操作：若溢出标志是1，则把从这条指令末尾到目的标号之间的距离，加到指令指示器来实现跳转。若 (OF)=0，则不跳转。

if (OF)=1 then

(IP)←(IP)+disp(sign-extended to 16-bits)

编码：

01110000	disp

时钟周期：跳转　8
　　不跳转　4

例：

JO TARGET_LABEL

影响标志位：无

说明：JO 在溢出时，把控制转移到目标操作数。

JP

JP 和 JPE (Jump on parity, or jump if parity even 奇偶性为偶跳转，或若奇偶校验为偶就跳转)

操作：若奇偶标志为1，则把从这条指令末尾到目的标号之间的距离，加到指令指示器来实现跳转。若 (PF)=0，则不跳转。

if (PF)=1 then

(IP)←(IP)+disp(sign-extended to 16-bits)

编码：

01111010	disp

时钟周期：跳转　8
　　不跳转　4

例：

(1) JP TARGET_LABEL

(2) JPE TARGET_LABEL

影响标志位：无

说明：JP（或 JPE）在奇偶性为偶(或校验为偶)时，把控制转移到目标操作数。

JPE

JP 和 JPE (Jump on parity, or jump if parity even 奇偶性为偶跳转，或若奇偶校验为偶就跳转)

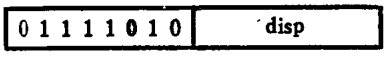
操作：若奇偶标志为 1，则把从这条指令末尾到目的标号之间的距离，加到指令指示器来实现跳转。

若 (PF)=0，则不跳转。

if (PF) =1 then

(IP)←(IP)+disp(sign-extended to 16-bits)

编码：



时钟周期：跳转 8

不跳转 4

例：

(1) JP TARGET_LABEL

(2) JPE TARGET_LABEL

影响标志位：无

说明：JP（或 JPE）在奇偶性为偶(或校验为偶)时，把控制转移到目标操作数。

JPO

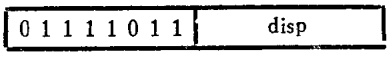
JNP 和 JPO (Jump on no parity or jump if parity odd 奇偶性为非偶跳转，或奇偶性为奇跳转)

操作：若奇偶标志为 1，意味着从上次影响 PF 的操作中出现了奇偶性为偶的结果，则不跳转。若 (PF)=0，则把从这条指令末尾到目的标号之间的距离，加到指令指示器来实现跳转。

if (PF)=0 then

(IP)←(IP)+disp (sign-extended to 16-bits)

编码：



时钟周期：跳转 8

不跳转 4

例：

(1) JNP TARGET_LABEL

(2) JPO TARGET_LABEL

影响标志位：无

说明：JNP（或 JPO）在奇偶性为非偶(或奇偶性为奇)时，把控制转移到目标操作数。

JS

(Jump on sign 符号标志置位跳转)

操作：若符号标志为 1，则把从这条指令末尾到目的标号之间的距离加到指令指示器来实现跳转。若 (SF)=0，则不跳转。

if (SF)=1 then

(IP)←(IP)+disp (sign-extended to 16-bits)

编码：

01111000	disp

时钟周期：跳转 8

不跳转 4

例：

JS TARGET_LABEL

影响标志位：无

说明：JS 在符号标志置位时，把控制转移到目标操作数。

JZ

JZ 和 JE (Jump if equal, jump if zero 若相等就跳转，或若是 0 就跳转)

操作：若影响 ZF 标志的上一次操作给出一个为 0 的结果，则(ZF)将是 1。若(ZF)=1，则把从这条指令的末尾到目的标号之间的距离，加到指令指示器来实现跳转。若(ZF)=0，则不跳转。

if (ZF)=1 then

(IP)←(IP)+disp (sign-extended to 16-bits)

编码：

01110100	disp

时钟周期：跳转 8

不跳转 4

例：

```
(1)        CMP CX, DX
           JE LAB2
           INC CX;
     LAB2;
     ;在 CX=DX 时才发生 CX 加 1
(2)        SUB AX, BX
           JZ EXACT
           ;若结果是零，即 AX=BX，跳转才发生。
           :
     EXACT;
```

影响标志位：无

说明：JE（或 JZ）在相等(或是0)时，把控制转移到目标操作数。

LAHF

(Load AH from flags 把标志位装入 AH)

操作：AH 的指定位被从下列标志中装入：符号标志填入第7位。0标志填入第6位。辅助进位标志填入第4位。奇偶校验标志填入第2位。进位标志填入第0位。AH 的第1、3、5 位不定，即它们某些时候是1在另一些时候为0。

AH←(SF):(ZF):X:(AF):X:(PF):X:(CF)

编码：

1 0 0 1 1 1 1 1

时钟周期：4

例：LAHF

影响标志位：无

说明：LAHF（把标志位装入 AH)）把标志寄存器 SF、ZF、AF、PF 和 CF（当 8080 代码翻译成 8086 代码时，它们是 8080 的标志），传送入 AH 寄存器的指定位，标着“X”的位没有指定。

LDS (Load data segment register 装入数据段寄存器)

操作：

1) 指定寄存器的内容，被双字存贮器操作数的较低地址字所替代。
(REG)←(EA)
2) DS 寄存器的内容，被双字存贮器操作数的较高地址字所替代。
(DS)←(EA+2)

编码：

1 1 0 0 0 1 0 1	mod	reg	r/m

mod≠11 (若 mod=11，则不定义操作)

时钟周期：16+EA

例：

```
LDS BX, ADDR_TABLE[SI]
LDS SI, NEWSEG[BX]
```

影响标志位：无

说明：LDS（把指示器装入 DS)把一个“指示器目标”(即一个含有一个偏移地址和一个段地址的32位目标)，从源操作数(它必须是一个双字存贮器操作数)传送到一对目寄存器。段地址被传送到 DS 段寄存器中。偏移地址被传送到所指定的任何通用指示器或变址器寄存器中(不能是段寄存器)。

LEA

(Load effective address 装入有效地址)

操作：指定寄存器的内容，被所指定的变量、或标号、或地址表达式的偏移量所替代。

(REG)←EA

编码：

1 0 0 0 1 1 0 1	mod	reg	r/m

mod≠11 (若 mod=11 则不定义操作)

时钟周期：2+EA

例：

```
LEA BX, VARIABLE-7
LEA DX, BETA[BX][SI]
LEA AX, [BP][DI]
```

影响标志位：无

说明：LEA (装入有效地址) 把源操作数的偏移地址，传送到目操作数。源操作数必须是一个存贮器操作数，而目操作数可以是任何 16 位的通用，指示器或变址器寄存器。LEA 允许源是变址的。使用带 OFFSET 算术符号的 MOV 指令不允许这样做。同样后者的操作，固定地使用变量在它被定义的段中的偏移量，但是，若组是通过最近的 ASSUME 伪指令的唯一可能的访问通道，LEA 将取得一个组偏移量。

LES

(Load extra-segment register 装入附加段寄存器)

操作：

1) 指定寄存器的内容，被双字存贮器操作数的较低地址字所替代。
   (REG)←(EA)
2) ES 寄存器的内容，被双字存贮器操作数的较高地址字所替代。
   (ES)←(EA+2)

编码：

1 1 0 0 0 1 0 0	mod	reg	r/m

mod≠11 (若 mod=11，则不定义操作。)

时钟周期：16+EA

例：

```
LES BX, ADDR_TABEL[SI]
LES DI, NEWSEG[BX]
```

影响标志位：无

说明：LES (把指示器装入 ES) 把一个“指示器目标”(即一个含有一个偏移地址和一个段地址的 32 位目标)，从一个源操作数(它必须是双字存贮器操作数)，传送到一对目寄存器。段地址传送到 ES 段寄存器。偏移地址可以传送到一个 16 位的通用、指示器或变址器寄存器(但不能是段寄存器)。

LOCK

(LOCK 封锁)

操作：无

编码：

1 1 1 1 0 0 0 0

时钟周期：2

例：LOCK

影响标志位：无

说明：这个特殊的一字节封锁前缀，可以放在任何指令之前。它使处理器在这条指令的操作期间发出总线封锁信号。在有共享资源的多处理器系统中，提供加强对那些资源的访问控制的机构是必须的。这种机构通过软件操作系统提供时需要硬件支持。足以完成这一任务的机构是封锁交换(也称为测试——设置封锁)。

假定外部硬件在接收这个信号后，在信号的有效期间，将禁止其他的总线主设备进行总线访问。

这条指令在寄存器和存贮器交换时很有用。下列代码序列可以组成一个简单的软件循环：

```
Check: MOV  AL, 1       ;把 AL 设置为 1 (暗示封锁)。
LOCK   XCHG Sema, AL    ;测试并设置封锁。
       TEST AL, AL      ;根据 AL 设置标志。
       JNZ  Check       ;若封锁已经设置,再测试。
        ⋮
       MOV  Sema, 0     ;做完以后清除封锁。
```

LOCK 前缀可以同段超越和/或重复前缀组合(同重复前缀的组合会发生一些问题，详见 REP 指令前缀)。

LODS

(Load byte or word string 装入字节或字串)

操作：若方向标志复位((DF)=0)，则把源字节(或字)装入 AL (或 AX)，再把源变址器加 1 (对字串加 2)；否则把 SI 减 1 (或 2)。

(DEST)←(SRC)

if (DF)=0 then (SI)←(SI)+DELTA

else (SI)←(SI)−DELTA

编码：

1 0 1 0 1 1 0	w

1) if w=0 then SRC=(SI), DEST=AL, DELTA=1

2) else SRC=(SI)+1:(SI), DEST=AX, DELTA=2

时钟周期：12

例：

```
1) CLD      ;清方向标志使 SI 为增加
   MOV SI, OFFSET BYTE_STRING
   LODS BYTE_STRING   ;SI←SI+1
    ⋮
2) STD      ;设置 DF 使 SI 为减少
   MOV SI, OFFSET WORD_STRING
   LODS WORD_STRING; SI←SI-2
   ;DF=1 隐含着变量 WORD_STRING 指示着串中的最后一个或最高地址字。在
   ;LODS 指令中的操作数，名仅由汇编程序用当前段寄存器的内容来验证类型和可访问
   ;性。LODS 实际上只使用 SI 去指出其内容要被装入到累加器中去的单元，而不使用
   ;在源指令中给出的名。
```

影响标志位：无

说明：LODS 把一个由 SI 寻址的源操作数字节，传送到累加器 AL (或 AX)，并以 DELTA 调整 SI 寄存器。这个操作通常不重复进行。

LOOP

(LOOP, or iterate instruction sequence until count complete 循环，或重复指令序列直到完成计数)

操作：把计数寄存器 (CX) 减 1。若新 CX 不是 0，则把从这条指令的末尾到目的标号之间的距离，加到指令计数器来实现跳转。若 CX=0，则不能跳转。

(CX) ← (CX) −1

if (CX) ≠0 then

(IP) ← (IP) +disp (sign-extended to 16-bits)

编码：

11100010	disp

时钟周期：跳转 9

不跳转 5

例：

下面是一个计算一个非 0 数组的 16 位累加和的序列：

```
(1)     MOV CX, LENGTH ARRAY
        MOV AX, 0
        MOV SI, AX
NEXT:   ADD AX, ARRAY[SI]
        ADD SI, TYPE ARRAY
        LOOP NEXT
        MOV CKS, AX
(2)     MOV AX, 0
        MOV BX, 1
```

```
        MOV CX, N;项的个数
        MOV DI, AX
FIB:    MOV SI, AX
        ADD AX, BX
        MOV BX, SI
        MOV FIBONACCI[DI], AX
        ADD DI, TYPE FIBONACCI
LL:     LOOP FIB
        ;从 FIB 到 LL 的指令将执行 N 次，并把该序列的前 N 项存放到 FIBONACCI
        数组中去,即 1,1,2,3,5,8,13,21,……
```

影响标志位：无

说明：LOOP 把 CX (计数)寄存器减 1，并且在 CX 不是 0 时，把控制转移到目标操作数(标号)。

LOOPE

LOOPZ 和 LOOPE (LOOP on equal, or loop on zero 相等时循环,或为 0 时循环)

操作：把计数寄存器 (CX) 减 1。若 0 标志置位,并且 (CX) 还不是 0，则把从这条指令末尾到目的标号之间的距离,加到指令指示器来实现跳转。若 (ZF) = 0 或 (CX) = 0，则不跳转。

(CX) ← (CX) − 1

if (ZF) = 1 and (CX) ≠ 0 then

(IP) ← (IP) + disp (sign-extended to 16-bits)

编码：

11100001	disp

时钟周期：跳转 11

不跳转 5

例：

下列序列在一个字节数组中找到第一个非 0 项：

```
        MOV CX, LENGTH ARRAY
        MOV SI, -1
NEXT:   INC SI
        CMP ARRAY[SI], 0
        LOOPE NEXT
        JNE OKENTRY
        ⋮                       ;若数组全为 0 才到达这里
OKENTRY:                        ;SI 告知哪一项是非 0
```

影响标志位：无

说明：LOOPE 也称为 LOOPZ (为 0 或相等时循环) 把 CX 寄存器减 1，并且若 CX 是非

0 和 ZF 标志置位就转移。

LOOPNE

LOOPNZ 和 LOOPNE (LOOP on not zero,or loop on not equal 非0或不相等时循环)

操作：把计数寄存器(CX)减 1。若新的(CX)为非 0，并且 0标志复位，则把从这条指令末尾到目的标号之间的距离，加到指令指示器来实现跳转。若(CX)=0 或 (ZF)=1，则不跳转。

(CX)←(CX)−1

if (ZF)=0 and (CX)≠0 then

(IP)←(IP)+disp (sign-extended to 16-bits)

编码：

11100000	disp

时钟周期：　跳转　11

　　不跳转　5

例：

下列序列将计算 2 个字节数组的和，每个的长度为 N，仅当在两个数组中同时遇到 0 项时才停止。在该点表达式 SI-1 将给出非 0 和数组的长度。

```
        MOV AX, 0
        MOV SI,-1
        MOV CX, N
NONZER: INC SI
        MOV AL, ARRAY1[SI]
        ADD ,ARRAY2[SI]
        MOV SUM[SI], AX
        LOOPNZ NONZER
```

下列序列将沿着一个连接表搜索最后一个元素。这个元素在通常放下一个元素的地址的字中放一个 0,这个字总是放在从每一个表元素的头上开始的相同字节数的位置。LINK 是该绝对字节数的名,例如：

```
        LINK EQU 7
        MOV AX, OFFSET HEAD_OF_LIST
        MOV CX, 1000 ; 最多搜索 1000 项。
NEXT:   MOV BX, AX
        MOV AX,[BX]+LINK
        CMP AX, 0
        LOOPNE NEXT
```

影响标志位：无

说明：LOOPNZ 也称为 LOOPNE(当非 0 或不相等时循环)把 CX 寄存器减 1,并且在 CX 非 0 和 ZF 被清除时转移。

LOOPNZ

LOOPNZ 和 LOOPNE (Loop on not zero, or loop on not equal 非0或不等循环)

操作：把计数寄存器(CX)减1,若新的(CX)为非0,并且0标志复位,则把从这条指令末尾到目的标号之间的距离，加到指令指示器来实现跳转。若 (CX)=0 或(ZF)=1,则不跳转。

(CX)←(CX)−1

if(ZF)=0 and (CX)≠0 then

(IP)←(IP)+disp (sign-extended to 16-bits)

编码：

1 1 1 0 0 0 0 0	disp

时钟周期：跳转 11

不跳转 5

例：

下列序列计算两个字节数组的和,每个数组长 N，仅当在两个数组中同时遇到0项时才停止。在该点,表达式 SI-1 将给出非0和数组的长度。

```
        MOV AX, 0
        MOV SI,-1
        MOV CX, N
NONZER: INC SI
        MOV AL, ARRAY1[SI]
        ADD ,ARRAY2[SI]
        MOV SUM[SI], AX
        LOOPNZ NONZER
```

下列序列将沿着一个连接表搜索最后一个元素，这个元素在通常放下一个元素的地址的字中放一个0。这字总是放在从每一个表元素的头上开始的相同字节数的位置。LINK是该绝对字节数的名,例如：

```
        LINK EQU 7
        MOV AX, OFFSET HEAD_OF_LIST
        MOV CX, 1000     ;最多搜索1000个项
NEXT:   MOV BX, AX
        MOV AX,[BX]+LINK
        CMP AX, 0
        LOOPNE NEXT
```

影响标志位：无

说明：LOOPNZ,也称为 LOOPNE(在非0或不等于时循环),把 CX 寄存器减1，并当 CX 为非0和 ZF 标志被清除时转移。

LOOPZ

LOOPZ 和 LOOPE (Loop on equal, or loop on zero 相等,或为0循环)

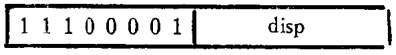
操作：把计数寄存器(CX)减1，若0标志置位，并且(CX)还不是0，则把从这条指令末尾到目的标号之间的距离，加到指令指示器来实现跳转。若 (ZF)=0 或 (CX)=0，则不跳转。

(CX)←(CX)−1

if (ZF)=1 and (CX)≠0 then

(IP)←(IP)+disp (sign-extended to 16-bits)

编码：



时钟周期：跳转 11

不跳转 5

例：

下列序列在一个字节数组中找到第一个非0项：

```
        MOV CX, LENGTH ARRAY
        MOV SI, -1
NEXT:   INC SI
        CMP ARRAY[SI], 0
        LOOPZ NEXT
        JNE OKENTRY
        :           ;若整个数组都是0，才到达这里。
OKENTRY:            ;SI 告知那一项是非0。
```

影响标志位：无

说明：LOOPZ，也称为 LOOPE (在为0或相等时循环)，把 CX 寄存器减1，并若 CX 为非0和 ZF 标志置位就转移。

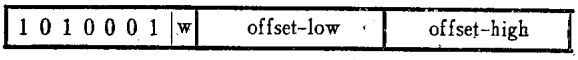
MOV

(Move 传送)

有七种不同类型的传送指令如下所示。

每一种类型具有多种用途，并且根据被传送的数据和数据的位置进行编码。汇编程序根据这两个因素产生正确的编码。若目是一个寄存器，表示为“d”的位将是1，否则是0，若类型是字，表示为“w”的位将是1，否则是0。

类型 1：从累加器送到存贮器



if w=0 then SRC=AL, DEST=offset else SRC=AX, DEST=offset+1: offset

时钟周期： 10+EA

例：

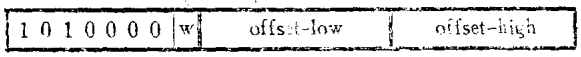
```
MOV ALPHA_MEM, AX
MOV GAMMA_BYTE, AL
```

MOV CS:DATUM_BYTE, AL

MOV ES: ARRAY[BX][SI], AX

(前缀字节，例如，ES: 将放在指令字节之前)

类型 2：从存贮器送到累加器



if w=0 then SRC=offset, DEST=AL else SRC=offset+1: offset, DEST=AX

时钟周期： 8+EA

例：

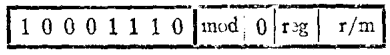
MOV AX, BEAT_MEM

MOV AL, GAMMA_BYTE

MOV AX, ES: ARRAY[BX][SI]

MOV AL, SS: OTHER_BYTE

类型 3：从存贮器或寄存器操作数送到段寄存器



if reg≠01 then SRC=EA, DEST=REG else undefined operation

时钟周期：寄存器送到寄存器 2

存贮器送到寄存器 8+EA

例：

MOV ES, DX

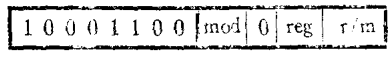
MOV DS, AX

MOV SS, BX

MOV ES, SS:NEW_WORD[DI]

注意：CS 在这里作为目是非法的。

类型 4：从段寄存器到存贮器或寄存器



SRC=REG DEST=EA, (DEST)←(SRC)

时钟周期：从存贮器送到寄存器 9+EA

从寄存器送到寄存器 2

例：

MOV DX, DS

MOV BX, ES

MOV ARRAY[BX][SI], SS

MOV BETA_MEM_WORD, DS

MOV GAMMA, CS; 注意 CS 在这里作为源是非法的。

类型 5：(a) 从寄存器送到寄存器

(b) 从存贮器或寄存器操作数送到寄存器

(c) 从寄存器送到存贮器或寄存器操作数

1 0 0 0 1 0	d	w	mod	reg	r/m	disp-low*	disp-high*

if d=1 then SRC=EA, DEST=REG else SRC=REG, DEST=EA

* 这些字节在从寄存器到寄存器的传送中省去,即当 mod=11, MOV CX, DX 和当存贮器的地址表达式是无变量名位移量的寄存器间接方式时,即,

MOV [BX][SI], DX

MOV AX, [BP][DI]

时钟周期: (a) 2

(b) 8+EA

(c) 9+EA

例:

(a) MOV AX, BX
MOV CL, DH
MOV CX, DI

(b) MOV AX, MEM_VALUE
MOV DX, ARRAY[SI]
MOV DI, MEM[BX][DI]

(c) MOV ARRAY[DI], DX
MOV MEM_VALUE, AX
MOV [BX][SI], DI

类型 6: 立即数送到寄存器

1 0 1 1	w	reg	data	data-high*

SRC=data, DEST=REG

*仅当 w=1 时出现.

时钟周期: 4

例:

MOV AX, 77
MOV BX, VALUE_14_IMM
MOV SI, EQU_VAL_9
MOV DI, 618

类型 7: 立即数送到存贮器或寄存器操作数

1 1 0 0 0 1 1	w	mod	0 0 0	r/m	data	data-high*

SRC=data, DEST=EA

* 仅当 w=1 时才出现

时钟周期: 10+EA

例:

MOV ARRAY[BX][SI], DATA_4
MOV MEM_BYTE, IMM_BYTE_3

```
MOV BYTE PTR [DI], 66
MOV MEM_WORD, 1999
MOV BX, 84
MOV DS:MEM_WORD[BP], 3989
```

(前缀字节,例如 DS: 00111110 要放在上面的 1100011w 前面.)

影响标志位：无

注意：指令中直接引用段寄存器,使用一个 2 位 reg 字段

MOVS (Move byte string or move word string 传送字节串或传送字串)

操作：把偏移量在源变址器中的源串,传送到偏移量在目变址器中位于附加段的单元中去.若方向标志是 0, SI 和 DI 都增加,若 (DF)=1,则都减少. 对于字节串,增加或减少 1,对于字串增加或减少 2.

```
(DEST)←(SRC)
if (DF)=0 then
    (SI)←(SI)+DELTA
    (DI)←(DI)+DELTA
else
    (SI)←(SI)−DELTA
    (DI)←(DI)−DELTA
```

编码：

1	0	1	0	0	1	0	w

if w=0 then SRC=(SI), DEST=(DI), DELTA=1

else SRC=(SI)+1:(SI), DEST=(DI)+1:(DI), DELTA=2

时钟周期： 17

例：

```
    MOV SI, OFFSET SOURCE
    MOV DI, OFFSET DEST
    MOV CX, LENGTH SOURCE
REP MOVS DEST, SOURCE
    ;上面的序列,把一个源串(在任何由当前段寄存器可到达的段中),全部传送到在附加
    ;段的目的单元中 (ES 寄存器在串操作中总是为 DI 操作数所使用). 在串操作中命
    ;名的操作数,仅由汇编程序用当前段寄存器的内容来验证类型和可访问性. MOVS
    ;实际上把由 SI 所指示的字节, 传送到在 ES 中由 DI 所指示的字节,而并不使用
    ;在源 MOVS 指令中给出的名.
```

影响标志位：无

说明：MOVS 把由 SI 寻址的一个字节 (或字) 源操作数, 传送到由 DI 寻址的目操作数中去,并且用 DELTA 调整 SI 和 DI 寄存器.在重复操作时,就把一个串从存贮器中的

一个区域传送到另一个区域。

MUL

(Multiply accumulator by register-or-memory; unsigned 累加器乘以寄存器或存贮器；无符号)

操作：累加器 (若是字节,为 AL，若是字,为 AX) 乘以指定的操作数，若结果的高半是0，则把进位和溢出标志复位,否则,把它们置位。

(DEST)←(LSRC)*(RSRC)，　这里*是无符号乘

if (EXT)=0 then (CF)←0 else (CF)←1

(OF)←(CF)

编码：

1 1 1 1 0 1 1	w	mod	1 0 0	r/m

(a) if w=0 then LSRC=AL, RSRC=EA, DEST=AX, EXT=AH

(b) eles LSRC=AX, RSRC=EA, DEST=DX:AX, EXT=DX

时钟周期：　8位　71+EA

16位　124+EA

例：

(a) MOV AL, LSRC_BYTE
MUL RSRC_BYTE ;结果在 AX 中

(b1) MOV AX, LSRC_WORD
MUL RSRC_WORD ;高半结果在 DX 中,低半在 AX 中。

(b2) 以一个字乘以一个字节：
MOV AL, MUL_BYTE
CBW ;把在 AL 中的字节转换为在 AX 中的字
MUL RSRC_WORD

注意：上列任何存贮器操作数,可以是一个正确 TYPE 的变址地址表达式。例如，LSRC_BYTE 在 ARRAY 是类型 BYTE 时,可以是 ARRAY[SI]；若 TABLE 是 WORD 类型时，RSRC_WORD 可以是 TABLE[BX][DI]。

影响标志位：CF, OF

无定义标志位：AF, PF, SF, ZF

说明：MUL (乘) 对累加器和源操作数执行无符号相乘，返回一个双倍长度的结果给累加器和它的扩展部分(对8位操作,是 AL 和 AH，对16位操作,是 AX 和 DX)。若结果的高半部分是非0，则 CF 和 OF 置位。

NEG

(Negate, or form 2's complement 取补,或形成2的补码)

操作：从全1 (对字节是 0FFH，对字是 0FFFF)中减去指定的操作数,并加上1，把结果存放回给定的操作数。

(EA)←SRC－(EA)

(EA)←(EA)＋1　(影响标志位)

编码：

1 1 1 1 0 1 1	w	mod	0 1 1	r/m

if w=0 then SRC=0FFH

else SRC=0FFFFH

时钟周期：寄存器　3

存贮器　16+EA

例：

(1) 若 AL 含有 13H (00010011)，则 NEG AL 使 AL 含有 －13H 或 0EDH (11101101)。

(2) 若 MEM_BYTE 含有 0AFH (10101111)，则 NEG MEM_BYTE 使 MEM_BYTE 含有 －0AFH 或 51H (01010001)。

(3) 若 SI 含有 2FC3H，则 NEG SI 使 SI 含有 0D03DH。

影响标志位：AF，CF，OF，PF，SF，ZF

说明：NEG (取补) 执行一个从0中减去操作数，并把结果返回到操作数的操作。这样形成了指定操作数的2的补码。

NOP

(No operation 空操作)

操作：无

编码：

1 0 0 1 0 0 0 0

时钟周期：3

例：

```
NOP
```

影响标志位：无

说明：NOP 引起空操作，并占3个时钟周期，然后顺序执行下面的指令。

NOT

(Not, or from 1's complement 非，或形成1的补码)

操作：从 0FFH (在字情况下，是 0FFFFH) 中减去指定的操作数，并把结果放回到给定的操作数。

(EA)←SRC－(EA)

编码：

1 1 1 1 0 1 1	w	mod	0 1 0	r/m

if w=0 then SRC=0FFH

else SRC=0FFFFH

时钟周期：寄存器　3

存贮器　16+EA

例：

(1) 若 AH 含有 13H (00010011)，则 NOT AH 使 AH 含有 0ECH (11101100)。

(2) 若 MEM_BYTE 含有 0AFH(10101111)，则 NOT MEM_BYTE 使 MEM_BYTE 含有 50H (01010000)。

(3) 若 DX 含有 2FC3H，则 **NOT DX** 使 DX 含有 0D03CH。

影响标志位：无

说明：NOT 形成操作数的 1 的补码(反码)，并把结果返回到操作数。标志位不受影响。

OR

(Or, inclusive 或，包含或)

操作：除非目操作数和源操作数在某一位都是 0，目 (左) 操作数中的该位将变成 1。换句话说，若任一个操作数在某位有一个 1，该位的结果就是 1；若都是 0 则结果是 0。进位和溢出标志都复位。

(DEST) ← (LSRC) ∣ (RSRC)

(CF) ←0

(OF) ←0

编码：

存贮器或寄存器操作数：

0 0 0 0 1 0	d	w	mod	reg	r/m

if d=1 then LSRC=REG, RSRC=EA, DEST=REG

else LSRC=EA, RSRC=REG, DEST=EA

时钟周期：(a) 寄存器或寄存器　3

(b) 存贮器或寄存器　9+EA

(c) 寄存器或存贮器　16+EA

例：

```
(a) OR AH, BL ;结果在 AH 中，BL 不改变
    OR SI, DX  ;结果在 SI 中，DX 不改变
    OR CX, DI  ;结果在 CX 中，DI 不改变
(b) OR AX, MEM_WORD
    OR CL, MEM_BYTE[SI]
    OR SI, ALPHA[BX][SI]
(c) OR BETA[BX][DI], AX
    OR MEM_BYTE, DH
    OR GAMMA[DI], BX
```

立即操作数或累加器：

0 0 0 0 1 1 0	w	data	data if w=1

(a) if w=0 then LSRC=AL, RSRC=data, DEST=AL

(b) else LSRC=AX, RSRC=data, DEST=AX

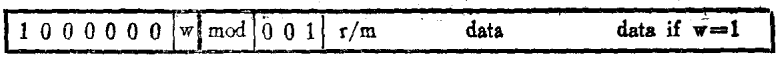
时钟周期：立即操作数或寄存器　4

例：

(a) OR AL, 11110110B
OR AL, 0F6H

(b) OR AX, 23F6H
OR AX, 75Q
OR ,23F6H

立即操作数或存贮器或寄存器操作数：



LSRC=EA, RSRC=data, DEST=EA

时钟周期：(a) 立即数或寄存器　4
(b) 立即数或存贮器　17+EA

例：

(a) OR AH, 0F6H
OR CL, 37
OR DI, 23F5H

(b) OR MEM_BYTE, 3DH
OR GAMMA[BX][DI], 0FACEH
OR ALPHA[DI], VAL_EQUD_33H

影响标志位：CF, OF, PF, SF, ZF

无定义标志位：AF

说明：OR 把两个操作数按位进行逻辑或并把结果放回到目操作数。

OUT

(Output byte and output word 输出字节和输出字)

操作：目标转接口的内容被累加器的内容替代。
(DEST)←(SRC)

编码：

固定转接口：

if w=0 then SRC=AL, DEST=port
else SRC=AX, DEST=port+1:port
(0<port<255)

时钟周期：　10

例：

OUT BYTE_PORT_VAL, AL　;从 AL 中输出一个字节。
OUT WORD_PORT_VAL, AX　;从 AX 中输出一个字。

OUT 44, AX ;通过转接口 44 从 AX 中输出一个字。

可变转接口：

1 1 1 0 1 1	w

if w=0 then SRC=AL, DEST=(DX)

else SRC=AX, DEST=(DX)+1:(DX)

时钟周期：8

例：

OUT DX, AL ;通过在 DX 中的可变转接口,从 AL 中输出一个字节。

OUT DX, AX ;通过在 DX 中的可变转接口,从 AX 中输出一个字。

影响标志位：无

说明：OUT 从 AL 寄存器(或 AX 寄存器)中向一个输出转接口传送一个字节(或字)。转接口或是在指令行的数据字节中指定,以便对 0 到 255 号转接口进行访问,或把一个转接口数放在 DX 寄存器中,以便对 64K 输出转接口进行变量访问。

POP

(POP word off stack into destination 把字从堆栈中弹入到目操作数)

有三种 POP 指令的类型,用于不同的目的。

操作：

(1) 用在堆栈顶的字替代目操作数的内容。

(DEST)←((SP)+1:(SP))

(2) 把堆栈指示器加 2

(SP)←(SP)+2

影响标志位：无

类型 1：

寄存器操作数：

0 1 0 1 1	reg

DEST=REG

时钟周期：8

例：

POP CX

汇编程序产生 01011001

POP DX

汇编程序产生 01011010

类型 2：

段寄存器：

0 0 0	reg	1 1 1

if reg≠01 then DEST=REG

else undefined operation

注意：POP CS 是非法的

时钟周期：8

例：

POP SS

汇编程序产生00010111

POP DS

汇编程序产生00011111

类型 3：

存贮器或寄存器操作数：

1 0 0 0 1 1 1 1	mod	0 0 0	r/m

DEST=EA

时钟周期：存贮器 17+EA

寄存器 8

例：

```
POP ALPHA
; 汇编程序产生1000111100000110 ALPHA offset-low ALPHA offset-high
POP ALPHA[BX]
; 汇编程序产生10001111 10000111 ALPHA disp low ALPHA disp-high
```

说明：POP 从由 SP 寄存器寻址的堆栈元素中传送一个字到目的操作数，然后把 SP 加2。

POPF

(Pop flags off stack 把标志从堆栈中弹出)

操作：

Flags←((SP)+1:(SP))

(SP)←(SP)+2

把在堆栈顶上的字的规定的位，填入到标志寄存器，即

overflow	flag←bit 11
direction	flag←bit 10
interrupt	flag←bit 9
trap	flag←bit 8
sign	flag←bit 7
zero	flag←bit 6
auxiliary carry	flag←bit 4
parity	flag←bit 2
carry	flag←bit 0

然后把堆栈指示器加 2。

编码：

1 0 0 1 1 1 0 1

时钟周期：8

例：

POPF

影响标志位：全部

说明：POPF (弹出标志)把由 SP 寄存器寻址的堆栈元素的指定位传送到标志寄存器，然后把 SP 加2。

PUSH

(Push word onto stack 把字推入堆栈)

依据提供的不同的操作数种类，有3种类型的 PUSH 指令。

操作：

(1) 把堆栈指示器(SP)减2

(SP)←(SP)−2

(2) 把指定操作数的内容，放到由 SP 指示着的堆栈顶单元中去。SP的内容用作相对于在 SS 段寄存器中的堆栈的基地址的偏移量。

((SP+1):(SP))←(SRC)

影响标志位：无

类型 1：

寄存器操作数(字)：

0 1 0 1 0	reg

时钟周期：10

例：

PUSH AX (产生：01010000)

PUSH SI (产生：01010110)

类型 2：

段寄存器：

0 0 0	reg	1 1 0

时钟周期：10

例：

PUSH SS (产生：00010110)

PUSH ES (产生：00000110)

PUSH DS

注意：PUSH CS 是合法的。

类型 3：

存贮器或寄存器操作数：

1 1 1 1 1 1 1 1	mod	1 1 0	r/m

时钟周期：存贮器　16+EA

寄存器　10

例：

```
            PUSH BETA
11111111 00 110 110 Beta offset-low Beta offset-high
            PUSH BETA[BX]
11111111 10 110 111 Beta disp-low Beta disp-high
            PUSH BETA[BX][DI]
11111111 10 110 001 disp-low Beta disp-high
```

说明：PUSH 把堆栈指示器 SP 减 2，然后把一个字从源操作数传送到由 SP 寻址的堆栈单元中去。

PUSHF

(Push flags onto stack 把标志位推入堆栈

操作：把堆栈指示器减 2，然后用标志位替代在堆栈顶的字的规定位。

(SP)←(SP)−2

((SP)+1:(SP))←Flags

编码：

1 0 0 1 1 1 0 0

时钟周期：10

例：

```
PUSHF
```

影响标志位：无

说明：PUSHF 把 SP 寄存器减 2，然后把所有的标志寄存器传送到由 SP 寻址的字的规定位中去。

RCL

(Rotate left through carry 通过进位位左环移)

操作：指定的目(左)操作数通过进位位左环移若干次 (COUNT)，这个次数或是准确地一次由一个绝对数值 1 指定，或是放在 CL 寄存器中，由一个 CL 右操作数指定。

环移一直进行到 COUNT 为 0。CF 被环移入目操作数的第 0 位。目操作数的最高位被环移入 CF。若 COUNT 是 1 并且原来目操作数的最高 2 位值是不相等的 (一个是 0，一个是 1)，则溢出标志置位，若它们相等，OF 复位。若 COUNT 不是 1，则 OF 无定义，并且没有可靠值。

```
(temp)←COUNT
do while (temp)≠0
(tmpcf)←(CF)
(CF)←high-order bit of (EA)
(EA)←(EA)*2+(tmpcf)
(temp)←(temp)−1
if OCUNT=1 then
```

if high-order bit of (EA) ≠ (CF) then (OF) ←1
else (OF) ←0
else (OF) undefined

编码：

1 1 0 1 0 0	v	w	mod	0 1 0	r/m

if v=0 then COUNT=1
else COUNT=(CL)

时钟周期：(a) 寄存器一位 2
(b) 存贮器一位 15+EA
(c) 寄存器可变位 8+4/位
(d) 存贮器可变位 20+EA+4/位

例：

```
(a) RCL AH, 1
    RCL BL, 1
    RCL CX, 1
    VAL_ONE EQU 1
    RCL DX, VAL_ONE
    RCL SI, VAL_ONE
(b) RCL MEM_BYTE, 1
    RCL ALPHA [DI], VAL_ONE
(c) MOV CL, 3
    RCL DH, CL ;左环移3位.
    RCL AX, CL
(d) MOV CL, 6
    RCL MEM_WORD, CL ;环移6次
    RCL GANDALF_BYTE, CL
    RCL BETA[BX][DI], CL
```

影响标志位：CF, OF

说明：RCL (通过进位位左环移)把操作数通过 CF 标志寄存器左环移 COUNT 位.

RCR

(Rotate right through carry 通过进位位右环移)

操作：指定的目(左)操作数通过进位位右环移若干次 (COUNT)，这个次数或是准确地一次由一个绝对数值 1 指定，或是放在 CL 寄存器中，由一个 CL 右操作数指定. 环移一直进行到 COUNT 为 0. CF 被环移入目操作数的最高位. 目操作数的最低位，被环移入 CF. 若 COUNT 是 1，并且目操作数最高 2 位的值现在不相等(一个是 0 一个是 1)，则溢出标志置位. 若它们相等则 OF 复位. 若 COUNT 不是 1，则 OF 无定义，并且没有可靠值.

```
(temp) ←COUNT
do while (temp) ≠0
  (tempcf) ← (CF)
  (CF) ←low-order bit of (EA)
  (EA) ← (EA) /2
  high-order bit of (EA) ← (tmpcf)
  (temp) ← (temp) -1
if COUNT=1 then
  if high-or der bit of (EA) ≠next-to- high-order bit of (EA)
      then (OF) ←1
    else (OF) ←0
  else (OF) undefined
```

编码：

1 1 0 1 0 0	v	w	mod	0 1 1	r/m

if v=0 then COUNT=1
else COUNT= (CL)

时钟周期：(a) 寄存器一位 2
(b) 存贮器一位 15+EA
(c) 寄存器可变位 8+4/位
(d) 存贮器可变位 20+EA+4/位

例：

```
(a) RCR AH, 1
    RCR BL, 1
    RCR CX, 1
    VAL__ONE EQU 1
    RCR DX, VAL__ONE
    RCR SI, VAL__ONE
(b) RCR MEM__BYTE, 1
    RCR ALPHA [DI], VAL__ONE
(c) MOV CL, 3
    RCR DH, CL ;右环移3位
    RCR AX, CL
(d) MOV CL, 6
    RCR MEM__WORD, CL ;环移6位
    RCR GANDALF__BYTE, CL
    RCR BETA[BX][DI], CL
```

影响标志位：CF, OF

说明：RCR (通过进位位右环移)把操作数通过 CF 标志寄存器右环移 COUNT 位。

REP

REP/REPZ/REPE/REPNE/REPNZ (Repeat string operation 重复串操作)

操作：把指定的串操作执行若干次，即，直到(CX)变成0。在每次重复后，把 CX 减1。当这个指令字节的0位值与0标志不相等时，比较和扫描串操作终止循环。

```
do while (CX) ≠0
    service pending interrupt (if any)
    execute primitive string operation in succeeding byte
    (CX) ← (CX) -1
    if primitive operation is CMPS
      or SCAS and (ZF) ≠z then exit from while loop
```

编码：

1 1 1 1 0 0 1	z

时钟周期：6/循环

例：

(1) REP MOVS DEST, SOURCE ;参见 MOVS

(2) REPE CMPS DEST, SOURCE

;只要(ZF)=1 循环将在(CX)=0 之前终止，即，仅当在(DI)中的字节等于在(SI)中

;的字节时就终止。

(3) REPNZ SCAS DEST ;参见 SCAS

;仅当(ZF)=1，即，(AL)=DEST，这个循环将在(CX)=0 之前终止。

(4) REPNZ (非0)=REPNE (不相等)

REPZ (0)=REPE (相等)

影响标志位：见各个串操作。

说明：REP(重复)使后面的基本串操作在(CX)不是0时重复执行。在 CMPS 和 SCAS 的情况下，若在任何基本操作的重复之后 ZF 标志不同于该重复前缀的 z 位，重复将终止。这个前缀可以与段超越前缀和/或 LOCK 前缀组合。在多前缀情况下，必须不允许中断，因为从一个中断的返回将把控制返回到被中断的指令或最多返回到该指令的一个前缀字节。

RET

(Return from procedure 从过程中返回)

操作：用在堆栈顶的字替代指令指示器(堆栈顶的偏移量在堆栈指示器中)，把 SP 加2。对于段间返回，用当前堆栈顶的字替代 CS 段寄存器，并再次把 SP 加2。若在 RET 语句中还指定了一个立即数，则把这个数加到 SP。

```
(IP) ← ((SP) +1: (SP))
(SP) ← (SP) +2
if inter-segment then
```

(CS)←((SP)+1:(SP))

(SP)←(SP)+2

if Add Immediate to Stack pointer then (SP)+data

编码：

段内：

11000011

时钟周期：8

例：RET

段内并给堆栈指示器加立即数：

11000010	data-low	data-high

时钟周期：12

例：

```
RET  4
RET 12
;这些值把事先存放在堆栈中的2个或6个参数丢弃掉。由于绝大多数堆栈操作,是在字
;上进行的,所以这些值通常是偶数(每个字2字节)。
```

段间：

11001011

时钟周期：18

例：

```
RET
```

段间并给堆栈指示器加立即数：

11001010	data-low	data-high

时钟周期：17

例：

```
RET 2 ;段间返回先恢复 IP，再恢复 CS
RET 8
```

影响标志位：无

说明：RET 把控制返回到以前 CALL 操作推入的返回地址，并且任选地给 SP 寄存器加一个立即常数来丢弃堆栈中的参数。若这是一个段间返回，即它是在一个标有 FAR 的过程下汇编的，则它将用在堆栈顶的两个字来替代 IP 和 CS。否则只用堆栈顶的一个字替代 IP。当使用间接调用时，程序员必须仔细地确保在过程中 CALL 的类型和 RET 的类型一致。例如：

```
CALL WORD PTR [BX]
必定不调用一个 FAR 过程,而
CALL DWORD PTR [BX]
必定不调用一个 NEAR 过程
```

ROL

(Rotate left 左环移)

操作：把指定的目(左)操作数左环移 COUNT 次。它的高位替代进位标志，而进位标志的原来值被丢失。在目操作数中的所有其他位"上移"一个位置，例如，第3位的值被第2位的值所替代。空出来的第0位由新的 CF 值，即原来的最高位填入。

环移一直进行到 COUNT 为0。若 COUNT 是1并且新的 CF 值不等于最高位的值，则溢出标志置位；若(CF)等于该最高位值，OF 变成0。但是，若 COUNT 不是1，则 OF 无定义，并且没有可靠值。

```
(temp)←COUNT
do while (temp)≠0
  (CF)←high-order bit of (EA)
  (EA)←(EA)*2+(CF)
  (temp)←(temp)-1
if COUNT=1 then
  if high-order bit of (EA)≠(CF) then (OF)←1
    else (OF)←0
else (OF) undefined
```

编码：

1 1 0 1 0 0	v	w	mod	0 0 0	r/m

```
if v=0 then COUNT=1
       else COUNT=(CL)
```

时钟周期：(a) 寄存器一位 2

(b) 存贮器一位 15+EA

(c) 寄存器可变位 8+4/位

(d) 存贮器可变位 20+EA+4/位

例：

```
(a) ROL AH, 1
    ROL BL, 1
    ROL CX, 1
    VAL_ONE EQU 1
    ROL DX, VAL_ONE
    ROL SI, VAL_ONE
(b) ROL MEM_BYTE, 1
    ROL ALPHA [DI], VAL_ONE
(c) MOV CL, 3
    ROL DH, CL ;左环移3位
    ROL AX, CL
(d) MOV CL, 6
```

```
ROL MEM_WORD, CL ;环移6次
ROL GANDALF_BYTE, CL
ROL BETA [BX] [DI], CL
```

影响标志位：CF，OF

说明：ROL(左环移)把操作数左环移 COUNT 位

ROR (Rotate right 右移环)

操作：把指定的目(左)操作数右环移 COUNT 次。它的最低位替代进位标志，该最低位的值丢失。在目操作数中的所有其他位都"下移"一个位置，例如第2位的值被第3位的值所替代。空出的最高位被新的 CF 值，即第0位的原来值填入

循环一直进行到 COUNT 为0。若 COUNT 为1，并且新的最高位值不等于老的最高位值，则溢出标志置位；若它们相等，则(OF)=0。但是，若 COUNT 不等于1，则 OF 无定义，并且没有可靠的值。

```
(temp)←COUNT
do while (temp)≠0
  (CF)←low order bit of (EA)
  (EA)←(EA)/2
  high-order bit of (EA)←(CF)
  (temp)←(temp)-1
if COUNT=1 then
   if high-order bit of (EA) ≠next-to-high-order bit of (EA)
      then (OF)←1
     else (OF)←0
   else (OF) undefinde
```

编码：

1 1 0 1 0 0	v	w	mod	0 0 1	r/m

```
if v=0 then COUNT=1
   else COUNT=(CL)
```

时钟周期：(a) 寄存器一位 2

(b) 存贮器一位 15+EA

(c) 寄存器可变位 8+4/位

(d) 存贮器可变位 20+EA+4/位

例：

```
(a) ROR AH, 1
    ROR BL, 1
    ROR CX, 1
    VAL_ONE EQU 1
```

```
    ROR DX, VAL_ONE
    ROR SI, VAL_ONE
(b) ROR MEM_BYTE, 1
    ROR ALPHA [DI], VAL_ONE
(3) MOV CL, 3
    ROR DH, CL ;右环移3位
    ROR AX, CL
(c) MOV CL, 6
    ROR MEM_WORD, CL ;环移6次
    ROR GANDALF_BYTE, CL
    ROR BETA[BX][DI], CL
```

影响标志位：CF，OF

说明：ROR(右环移)把操作数右环移 COUNT 位。

SAHF

操作：如下所示的5个标志位被累加器的高字节 AH 中的规定位所替代。

(SF)←bit 7
(ZF)←bit 6
(AF)←bit 4　　of AH
(PF)←bit 2
(CF)←bit 0

(SF):(ZF):X:(AF):X:(PF):X:(CF)←(AH)

编码：

10011110

时钟周期：4

例：

```
SAHF
```

影响标志位：AF，CF，PF，SF，ZF

说明：SAHF 把 AH 寄存器的指定位，传送到标志寄存器 SF，ZF，AF，PF 和 CF。在操作中由 X 所指的 AH 位被忽略。

SAL

SHL 和 SAL (Shift logical left and shift arithmetic left 逻辑左移和算术左移)

操作：把指定的目(左)操作数左移 COUNT 次。用它的最高位替代进位标志，进位标志的原来值丢失。在目操作数中的所有其他位"上移"一个位，例如第3位的值被第2位的值所替代。空出的最低位被填入0。

移位一直进行到 COUNT 为0。若 COUNT 是1，并且 CF 的新值不等于新的最高位的值，则溢出标志置位；若(CF)等于该最高位的值，(OF)变为0。但是，若 COUNT

不为 1,则 OF 无定义,且没有可靠值。

```
(temp) ←COUNT
do while (temp) ≠0
    (CF) ←high-order bit of (EA)
    (EA) ← (EA) *2
    (temp) ← (temp) -1
  if COUNT=1 then
    if high-order bit of (EA) ≠ (CF) then (OF) ←1
    else (OF) ←0
    else (OF) undefined
```

编码:

1 1 0 1 0 0	v	w	mod	1 0 0	r/m

```
if v=0 then COUNT=1
else COUNT= (CL)
```

时钟周期: (a) 寄存器一位 2
(b) 存贮器一位 15+EA
(c) 寄存器可变位 8+4/位
(d) 存贮器可变位 20+EA+4/位

例:

```
(a) SHL AH, 1
    SHL BL, 1
    SHL CX, 1
    VAL_ONE EQU 1
    SHL DX, VAL_ONE
    SHL SI, VAL_ONE
(b) SHL MEM_BYTE, 1
    SHL ALPHA[DI], VAL_ONE
(c) MOV CL, 3
    SHL DH, CL ;左移3位
    SHL AX, CL
(d) MOV CL, 6
    SHL MEM_WORD, CL ;左移6次
    SHL GANDALF_BYTE, CL
    SHL BETA[BX][DI], CL
```

影响标志位: CF, OF, PF, SF, ZF

无定义标志位: AF

说明: SHL(逻辑左移)和 SAL(算术左移)把目操作数左移 COUNT 位,在低位移入 0。

SAR

(shift arithmetic right 算术右移)

操作：把指定的目(左)操作数右移 COUNT 次。它的最低位替代进位标志，进位标志的原来值丢失。在目操作数中的所有其他位"下移"一个位置，例如第 2 位的值被第 3 位的值替代。空出来的最高位保持它原来的值，即若原来最高位值是 0，就移入 0，若该值原来是 1，就移入 1。

移位一直进行到 COUNT 为 0。由于最高位始终保持不变，所以(OF)=0

```
(temp)←COUNT
do while (temp)≠0
  (CF)←low-order bit of (EA)
  (EA)←(EA)/2，这里/相当于带符号除并舍入，最高位保持原来的值
(temp)←(temp)-1
(OF)←0
```

编码：

1 1 0 1 0 0	v	w	mod	1 1 1	r/m

```
if v=0 then COUNT=1
  else COUNT=(CL)
```

时钟周期：(a) 寄存器一位 2
(b) 存贮器一位 15+EA
(c) 寄存器可变位 8+4/位
(d) 存贮器可变位 20+EA+4/位

例：

```
(a) SAR AH, 1
    SAR BL, 1
    SAR CX, 1
    VAL_ONE EQU 1
    SAR DX, VAL_ONE
    SAR SI, VAL_ONE
(b) SAR MEM_BYTE, 1
    SAR ALPHA[DI], VAL_ONE
(c) MOV CL, 3
    SAR DH, CL ;右移 3 位
    SAR AX, CL
(d) MOV CL, 6
    SAR MEM_WORD, CL ;移位 6 次
    SAR GANDALF_BYTE, CL
    SAR BETA[BX][DI], CL
```

影响标志位：CF, OF, PF, SF, ZF

无定义标志位：AF

说明：SAR(算术右移)把目操作数右移 COUNT 位，把原来操作数最高位的值移入该最高位。

SBB

(Subtract with borrow 带借位减)

操作：从目(左)操作数中减去源(右)操作数。若进位标志置位，则从上述结果中减去1。该结果替代原来的目操作数。

if (CF)=1 then (DEST)←(LSRC)-(RSRC)-1
else (DEST)←(LSRC)-(RSRC)

编码：

存贮器或寄存器操作数和寄存器操作数：

0 0 0 1 1 0	d	w	mod	reg	r/m

if d=1 then LSRC=REG, RSRC=EA, DEST=REG
else LSRC=EA, RSRC=REG, DEST=EA

时钟周期：(a) 寄存器减寄存器 3
(b) 寄存器减存贮器 9+EA
(c) 存贮器减寄存器 16+EA

例：

```
(a) SBB AX, BX
    SBB CH, DL
(b) SBB DX, MEM_WORD
    SBB DI, ALPHA[SI]
    SBB BL, MEM_BYTE[DI]
(c) SBB MEM_WORD, AX
    SBB MEM_BYTE[DI], BX
    SBB GAMMA[BX][DI], SI
```

寄存器减立即操作数：

0 0 0 1 1 1 0	w	data	data if w=1

(a) if w=0 then LSRC=AL, RSRC=dada, DEST=Al
(b) else LSRC=AX, RSRC=data, DEST=AX

时钟周期：寄存器减立即数 4

例：

```
(a) SBB AL, 4
    VAL_SIXTY EQU 60
    SBB AL, VAL_SIXTY
(b) SBB AX, 660
    SBB AX, VAL_SIXTY*6
```

SBB ,6606

存贮器或寄存器操作数减立即数：

1 0 0 0 0 0	s	w	mod	0 1 1	r/m	data	data if s:w=0 1

LSRC=EA, RSRC=data, DEST=EA

时钟周期：(a) 寄存器减立即数　4

(b) 存贮器减立即数　17+EA

例：

(a) SBB BX, 2001
SBB CL, VAL_SIXTY
SBB SI, VAL_SIXTY*9

(b) SBB MEM_BYTE, 12
SBB MEM_BYTE[DI], VAL_SIXTY
SBB MEM_WORD[BX], 79
SBB GAMMA[DI][BX], 1984

若要从一个寄存器或存贮器字中减去一个立即数字节，则在相减之前，该字节要符号扩展至16位。在这种情况下，指令字节是83H。

影响标志位：AF, CF, OF, PF, SF, ZF

说明：SBB(带借位减)执行两个操作数的减法，若 CF 标志是置位的则再减去1，并把结果返回到目操作数。

注意：一旦使用一个包括变量名的地址表达式，即一个包含数据的存贮单元的名，mod，r/m字节之后将有2字节的从段基地址计算出来的位移量。若还使用一个字节或字的立即数，则立即数将跟在位移量之后。

SCAS

(Scan byte string or scan word string)

操作：从在累加器的值中，减在当前附加段中由 DI 指定的串元素，但该操作只影响标志位。然后目变址器增加(若方向标志是0)或减少(若(DF)=1)。对于字节串增加或减少1，对于字串则增加或减少2。

(LSRC)-(RSRC)
if (DF)=0 then (DI)←(DI)+DELTA
else (DI)←(DI)-DELTA

编码：

1 0 1 0 1 1 1	w

if w=0 then LSRC=AL, RSRC=(DI), DELTA=1
else LSRC=AX, RSRC=(DI)+1:(DI), DELTA=2

时钟周期：　15

例：

(1) CLD ;清除 DF，使 DI 增加
```
MOV DI, OFFSET DEST_BYTE_STRING
MOV AL, 'M'
SCAS DEST_BYTE_STRING
```
(2) STD ;置位 DF，使 DI 减少
```
MOV DI, OFFSET WORD_STRING
MOV AX, 'MD'
SCAS WORD_STRING
```
;在 SCAS 指令中的操作数名，仅由汇编程序用当前段寄存器的内容来验证类型和
;可访问性。这条指令的实际操作，使用 DI 来指出要扫描的单元，并不使用在指令
;行中命名的操作数。

影响标志位：AF，CF，OF，PF，SF，ZF

说明：SCAS 从 AL(或 AX)中减去由 DI 寻址的目操作数字节(或字)操作数，但不返回结果。在重复操作的时候，就可以扫描一个给定的值在串中出现或不出现。

SHL

SHL 和 SAL (shift logical left and shift arithmetic left 逻辑左移和算术左移)

操作：把指定的操作数左移 COUNT 次。它的最高位取代进位标志，而进位标志的原来值丢失。在目操作数中的所有其他位都"上移"一位，例如第 3 位的值被第 2 位的值取代。空出的最低位被填入 0。

移位一直进行到 COUNT 为 0。若 COUNT 为 1，并且新的 CF 值不等于新的最高位的值，则溢出标志置位；若(CF)等于该最高位，OF 变为 0。但是，若 COUNT 不为 1，则 OF 无定义，并且没有有效值。

```
(temp)←COUNT
do while (temp)≠0
  (CF)←high-order bit of (EA)
  (EA)←(EA)*2
  (temp)←(temp)-1
if COUNT=1 then
  if high-order bit of (EA)≠(CF) then (OF)←1
        else (OF)←0
  else (OF) undefinde
```

编码：

1 1 0 1 0 0	v	w	mod	1 0 0	r/m

```
if v=0 then COUNT=1
  else COUNT=(CL)
```

时钟周期：(a) 寄存器一位　　2

(b) 存贮器一位　　15+EA

(c) 寄存器可变位　　8+4/位

(d) 存贮器可变位　　20+EA+4/位

例：

```
(a) SHL AH, 1
    SHL BL, 1
    SHL CX, 1
    VAL_ONE EQU 1
    SHL DX, VAL_ONE
    SHL SI, VAL_ONE
(b) SHL MEM_BYTE, 1
    SHL ALPHA[DI], VAL_ONE
(c) MOV CL, 3
    SHL DH, CL ;左移3位
    SHL AX, CL
(d) MOV CL, 6
    SHL MEM_WORD, CL ;移位6次
    SHL GANDALF_BYTE, CL
    SHL BETA[BX][DI], CL
```

影响标志位：CF，OF，PF，SF，ZF

无定义标志位：AF

说明：SHL(逻辑左移)和 SAL(算术左移)把目操作数左移 COUNT 位，最低位移入0。

SHR

(shift logical right 逻辑右移)

操作：把指定的目(左)操作数右移 COUNT 次。它的最低位取代进位标志，而原来的进位标志值丢失。在目操作数中的所有其他位都"下移"一位，例如，第2位的值被第3位的值所替代，空出来的最高位被填入0。

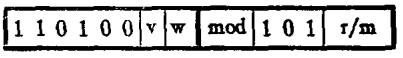
移位一直进行到 COUNT 为0。若 COUNT 为1，并且新的最高位的值不等于下一位的值，则溢出标志置位；若它们相等，则(OF)=0。但是，若 COUNT 不为1，则 OF 无定义，并且没有可靠的值。

```
(temp)←COUNT
do while (temp)≠0
  (CF)←low-order bit of (EA)
  (EA)←(EA)/2, 这里/等价于无符号除。
  (temp)←(temp)−1
if COUNT=1 then
  if high-order bit of (EA)≠next-to-high-order bit of (EA)
    then (OF)←1
```

else (OF)←0

else (OF) undefined

编码：



if v=0 then COUNT=1

else COUNT=(CL)

时钟周期：(a) 寄存器一位　　2

(b) 存贮器一位　　15+EA

(c) 寄存器可变位　　8+4/位

(d) 存贮器可变位　　20+EA+4/位

例：

```
(a) SHR AH, 1
    SHR BL, 1
    SHR CX, 1
    VAL_ONE EQU 1
    SHR DX, VAL_ONE
    SHR SI, VAL_ONE
(b) SHR MEM_BYTE, 1
    SHR ALPHA[DI], VAL_ONE
(c) MOV CL, 3
    SHR DH, CL ;左移3位
    SHR AX, CL
(d) MOV CL, 6
    SHR MEM_WORD, CL ;移位6次
    SHR GANDALF_BYTE, CL
    SHR BETA[BX][DI], CL
```

影响标志位：CF, OF, PF, SF, ZF

无定义标志位：AF

说明，SHR(逻辑右移)把目操作数右移 COUNT 位，最高位移入0。

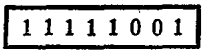
STC

(Set carry flag 置位进位标志)

操作：把进位标志设置为1

(CF)←1

编码：



时钟周期：2

例：

STC

影响标志位：CF

说明：STC 设置 CF 标志位。

STD

(set direction flag 置位方向标志)

操作：把方向标志设置为 1

(DF)←1

编码：

1 1 1 1 1 1 0 1

时钟周期：2

例：

STD

影响标志位：DF

说明：STD 置位 DF 标志，使串操作自减操作数变址器。

STI

(Set interrupt flag 置位中断标志)

操作：把中断标志设置为 1。

(IF)←1

编码：

1 1 1 1 1 0 1 1

时钟周期：2

例：

STI ;允许中断

影响标志位：IF

说明：STI 置位 IF 标志，在下一条指令执行之后，允许可屏蔽外部中断。

STOS (store byte string or store word string 存放字节串或存放字串)

操作：用在 AL (或 AX) 中的字节(或字)，取代在附加段中由 DI 指示着的字节(或字)的内容。若方向标志是 0，则 DI 增加；若(DF)=1 则 DI 减少，对于字节，改变量为 1；对于字，改变量是 2。

(DEST)←(SRC)

if (DF)=0 then (DI)←(DI)+DELTA

else (DI)←(DI)−DELTA

编码：

1 0 1 0 1 0 1	w

if w=0 then SRC=AL, DEST=(DI), DELTA=1
else SRC=AX, DEST=(DI)+1:(DI), DELTA=2

时钟周期：10

例：

```
(1) MOV DI, OFFSET BYTE_DEST_STRING
    STOS BYTE_DEST_STRING
(2) MOV DI, OFFSET WORD_DEST
    STOS WORD_DEST
```

影响标志位：无

说明：STOS 把在 AL(或 AX)中的一个字节(或字)，传送到由 DI 寻址的目操作数，并用 DELTA 调整 DI 寄存器。在重复操作时，这条指令提供了把给定值填入一个串的手段。在 STOS 指令中的操作数名只由汇编程序用当前段寄存器的内容来验证类型和可访问性。指令的实际操作，只使用由 DI 指示着的要被存入的单元。

SUB

(subtract 减)

操作：从目(左)操作数中减去源(右)操作数，并把结果存放到目操作数单元。

(DEST)←(LSRC)←(RSRC)

编码：

存贮器或寄存器操作数和寄存器操作数：

0 0 1 0 1 0	d	w	mod	reg	r/m

if d=1 then LSRC=REG, RSRC=EA, DEST=REG
else LSRC=EA, RSRC=REG, DEST=EA

时钟周期：(a) 寄存器减寄存器　3
(b) 寄存器减存贮器　9+EA
(c) 存贮器减寄存器　16+EA

例：

```
(a) SUB AX, BX
    SUB CH, DL
(b) SUB DX, MEM_WORD
    SUB DI, ALPHA [SI]
    SUB BL, MEM_BYTE[DI]
(c) SUB MEM_WORD, AX
    SUB MEM_BYTE[DI], BL
    SUB GAMMA[BX][DI], SI
```

累加器减立即操作数：

0 0 1 0 1 1 0	w	data	data if w=1

(a) if w=0 then LSRC=AL, RSRC=data, DEST=AL

(b) else LSRC=AX, RSRC=data, DEST=AX

时钟周期：寄存器减立即数　4

例：

(a) SUB AL, 4
VAL_SIXTY EQU 60
SUB AL, VAL_SIXTY

(b) SUB AX, 660
SUB AX, VAL_SIXTY*6
SUB ,6606

存贮器或寄存器操作数减立即操作数：

1 0 0 0 0 0	s	w	mod	1 0 1	r/m	data	data if s:w=01

LSRC=EA, RSRC=data, DEST=EA

时钟周期：(a) 寄存器减立即数　4
(b) 存贮器减立即数　17+EA

例：

(a) SUB BX, 2001
SUB CL, VAL_SIXTY
SUB SI, VAL_SIXTY*9

(b) SUB MEM_BYTE, 12
SUB MEM_BYTE[DI], VAL_SIXTY
SUB MEM_WORD[BX], 79
SUB GAMMA[DI][BX], 1984

若要从一个寄存器或存贮器字中减去一个立即数字节，则在相减之前，该字节要被符号扩展至16位。在这种情况下，指令字节是 83H。

影响标志位：AF, CF, OF, PF, SF, ZF

说明：SUB 从目(左)操作数中减去源(右)操作数，并把结果返回到目操作数。

TEST

(Test, or logical compare 测试，或逻辑比较)

操作：把两个操作数相与以影响标志位，但两个操作数都不改变。进位标志和溢出标志被复位。
(LSRC) & (RSRC)
(CF)←0
(OF)←0

编码：

存贮器或寄存器操作数与寄存器操作数：

1 0 0 0 0 1 0	w	mod	reg	r/m

LSRC=REG, RSRC=EA

时钟周期：(a) 寄存器与寄存器　3

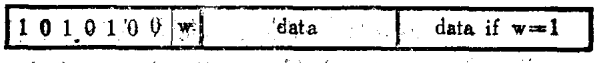
(b) 存贮器与寄存器　9+EA

例：

(a) TEST AX, DX
TEST ,DX ;同上
TEST SI, BP
TEST BH, CL

(b) TEST MEM_WORD, SI
TEST MEM_BYTE, CH
TEST ALPHA[DI], DX
TEST BETA[BX][SI], CX
TEST DI, MEM_WORD
TEST CH, MEM_BYTE
TEST AX, GAMMA[BP][SI]

立即操作数与累加器：



(a) if w=0 then LSRC=AL, RSRC=data

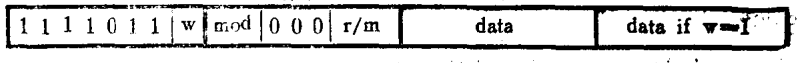
(b) else LSRC=AX, RSRC=data

时钟周期：立即数与寄存器　4

例：

TEST AL, 6
TEST AL, IMM_VALUE_DRIVE11
TEST AX, IMM_VAL_909
TEST ,999
TEST AX, 999 ;同上

立即操作数与存贮器或寄存器操作数：



LSRC=EA, RSRC=data

时钟周期：(a) 立即数与寄存器　4

(b) 立即数与存贮器　10+EA

例：

(a) TEST BH, 7
TEST CL, 19_IMM_BYTE
TEST DX, IMM_DATA_WORD
TEST SI, 798

(b) TEST MEM_WORD, IMM_DATA_BYTE
TEST GAMMA[BX], IMM_BYTE
TEST [BP][DI], 6ACEH

影响标志位：CF，OF，PF，SF，ZF

无定义标志位：AF

说明：TEST 对两个操作数执行按位逻辑与，使标志位受影响，但不返回结果。

源(右)操作数通常必须同目(左)操作数类型相同，即同是字节或字。对于 TEST，唯一的例外是一个立即数字节同一个存贮器字相与。

WAIT

(wait 等待)

操作：无

编码：

1	0	0	1	1	0	1	1

时钟周期：3

例：

```
WAIT
```

影响标志位：无

说明：若在 TEST 引脚上不出现信号，则 WAIT 指令使处理器进入等待状态。等待状态可以被一个允许的外部中断所中断。当中断发生时，被保护的指令单元就是 WAIT 指令的单元，以便在中断返回时再次进入等待状态。当 TEST 信号出现时，等待状态被清除，执行重新开始。重新开始的执行在下一条指令被执行之前不允许中断。这条指令使处理器实现与外部硬件同步。

XCHG

(Exchange 交换)

有两种 XCHG 指令，一种是把累加器的内容同某些通用字寄存器的内容相交换，另一种是把一个寄存器的内容和一个存贮器或寄存器操作数的内容相交换。

操作：

(1) 把目(左)操作数的内容，临时存放到一个内部工作寄存器

(temp)←(DEST)

(2) 用源(右)操作数的内容，替代目操作数的内容。

(DEST)←(SRC)

(3) 把目操作数原来的内容，从工作寄存器移送到源操作数。

(SRC)←(temp)

影响标志位：无

类型 1：

寄存器操作数与累加器：

1	0	0	1	0	reg

SRC=REG，DEST=AX

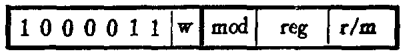
时钟周期：3

例：

```
XCHG AX, BX
XCHG SI, AX
XCHG CX, AX
```

类型 2：

存贮器或寄存器操作数与寄存器操作数：



SRC=EA, DEST=REG

时钟周期：存贮器与寄存器 17+EA

寄存器与寄存器 4

例：

```
XCHG BETA_WORD, CX
XCHG BX, DELTA_WORD
XCHG DH, ALPHA_BYTE
XCHG BL, AL
```

说明：XCHG 把源和目操作数的字节或字相交换，段寄存器不能是 XCHG 的操作数。

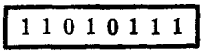
XLAT

(Translate 翻译)

操作：累加器 AL 的内容被来自一个表中的字节所取代，该表的起始地址，事先已被送入 BX 寄存器。AL 的原先的内容，是从起始地址开始所越过的字节数，在那儿可以找到所要翻译的字节，并用它替代 AL 的内容。

(AL)←((BX)+(AL))

编码：



时钟周期：11

例：

```
MOV BX, OFFSET TABLE_NAME
XLAT TABLE_NAME      ;关于指令中的操作数的说明参见 LODS 指令的例子
```

影响标志位：无

说明：XLAT 完成在一个表中查找一个字节的翻译工作，AL 寄存器用来作为进入由 BX 寄存器寻址的一个表的变址器。这样寻址到的字节就被送到 AL 中去。

XOR

(Exclusive or 异或)

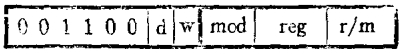
操作：若在两个操作数中的对应位相等，则目(左)操作数中的该位被设置为 0。若它们不相等，则该位被设置为 1。

(DEST)←(LSRC)‖(RSRC)

(CF)←0

(OF)←0

编码：存贮器或寄存器异或寄存器：



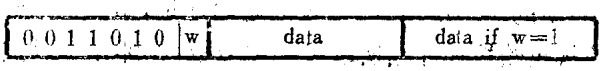
if d=1 then LSRC=REG, RSRC=EA, DEST=REG
else LSRC=EA, RSRC=REG, DEST=EA

时钟周期：(a) 寄存器异或寄存器　3
(b) 存贮器异或寄存器　9+EA
(c) 寄存器异或存贮器　16+EA

例：

```
(a) XOR AH, BL ;结果在 AH 中, BL 不改变
    XOR SI, DX  ;结果在 SI 中, DX 不改变
    XOR CX, DI  ;结果在 CX 中, DI 不改变
(b) XOR AX, MEM_WORD
    XOR CL, MEM_BYTE[SI]
    XOR SI, ALPHA[BX][SI]
(c) XOR BETA[BX][DI], AX
    XOR MEM_BYTE, DH
    XOR GAMMA[DI], BX
```

立即操作数异或累加器：



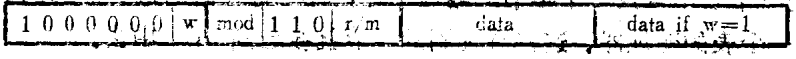
if w=0 then LSRC=AL, RSRC=data, DEST=AL
else LSRC=AX, RSRC=data, DEST=AX

时钟周期：立即数异或寄存器 4

例：

```
(a) XOR AL, 11110110B
    XOR AL, 0F6H
(b) XOR AX, 23F6H
    XOR AX, 75Q
    XOR ,23F6H    ;AX 是目.
```

立即操作数异或存贮器或寄存器操作数：



LSRC=EA, RSRC=data, DEST=EA

时钟周期：立即数异或寄存器　4
立即数异或存贮器　17+EA

例：

```
(a) XOR AH, 0F6H
    XOR CL, 37
    XOR DI, 23F5H
(b) XOR MEM_BYTE, 3DH
```

XOR GAMMA[BX][DI], 0FACEH

XOR ALPHA[DI], VAL__EQUD__33H

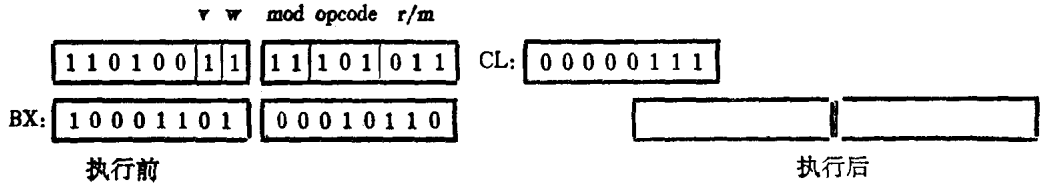
影响标志位：CF，OF，PF，SF，ZF

无定义标志：AF

说明：XOR（异或）把两个操作数按位进行异或操作，并把结果送回到目操作数。

习　题

1. 数据传送指令，可以把段寄存器作为目操作数，但又规定 CS 不能作为目操作数，为什么？
2. 已知堆栈段寄存器 SS 的内容是 0F0A0H，堆栈指示器 SP 的内容是 00B0H，先执行两条把 8057H 和 0F79BH 分别推入堆栈的 PUSH 指令，然后执行一条 POP 指令，试画出示意图，说明堆栈及 SP 内容的变化过程。
3. 写出从第 25 号转接口输入的指令。若要从第 512 号转接口输入，指令序列应是怎样的？试把它写出来。
4. 已知一个关于 0-9 的数字的 ASCII 码表的首地址是当前数据段的0A80H，现在要找出数字 5 的 ASCII 代码，试写出用指令 XLAT 进行翻译的指令序列(提示：0A80H 可作为立即数)。
5. 某一个子程序的功能是把一个变量的值乘以 10，该子程序被设计得总是从 BX 中取得该变量的偏移地址，今设该偏移地址为 7F80H，在某程序调用该子程序之前要用一条立即数指令作准备，试写出该条指令。
6. 设在当前数据段的偏移地址 2000H，含有一个内容为 0FF1CH 和 8000H 的指示器，它指着一个 16 位变量，设当前数据段寄存器的值为 1B00H，试写出把该变量值装入 AX 的指令序列。并以简图表示之。
7. 8086 的除法指令对于余数的规定是怎样的？试写出 −37 除以 +5，+37 除以 −5 的商和余数。
8. 试写出 13H，29H，45H，54H，76H，98H 的压缩 BCD 表示形式。
9. 试写出 27+36，35+48，29+28，2985+4936 的 BCD 加法的过程，并写出有关标志位的内容。
10. 有一个压缩的 BCD 数 47 在 AL 中，试验证可以利用除 16 的办法把它转换成非压缩的 BCD 数放在 AH 和 AL 中。
11. 为什么非压缩 BCD 的被除数、除数、乘数、被乘数的高 4 位必须都是 0，而对于加法和减法这一点则不是必须的？
12. 试写出一个多个数字 BCD 乘数的乘法的算法。
13. EPROM 是一种可重写的 ROM，它在照紫外光后所有的位都是“1”，通过写入装置，可以把“1”写成“0”，但若已经是“0”，则不能再被写成“1”。在写入程序中有一段判断 EPROM 的要写入字节是否能被正确写入，即判断要写入“1”的位是否已是 0？若是，该字节就不能被写入。设要写入的字节在 CL 中，未写入前 EPROM 的字节内容被事先读出放在 BL 中，试利用 8086 的逻辑指令设计一个测试方法。
14. 若要检查 BX 寄存器中第 13 位是否为“0”，该用什么指令序列，若要检查是否为“1”呢？
15. 使用移位指令来做乘以 2 和除以 2 是很方便的。试把 +53、−49 乘以 2，它们各应用什么指令，得到的结果各是什么？若除以 2 呢？
16. 有这样一条移位/环移指令，请写出 BX 的最终内容。



17. 利用串基本指令把存贮器中自 10000H 到 1FFFFH 的单元全部置成 FF，试写出指令序列(DS,ES 的内容可以自己设)。
18. 利用串比较指令，比较字符串 abcde, abcce; efgh, efghij按字典序，哪一个应放在前面，设 SI 的值为 1000H，DI 的值为 2000H，试描述一下指令执行的过程，写出各寄存器和标志位在每一步的变化情况，并据此决定哪一串比较大。
19. 要把一个字序列的值乘以 10，假设不会溢出，设该序列的第一个字的偏移量在 SI 中，已乘 10 的序列的第一个字存放的偏移地址在 DI 中，CX 含有该序列中字数的计数值，写出执行这一操作的步骤，看哪些可用串指令来加速，并把它写出来。
20. 直接跳转指令不用目的偏移量，而用该指令和目的偏移量之差来表示跳转目的有什么好处？
21. 处理器有几种中断引脚，它们的功用如何？设处理器在 INTR 引脚上收到一个中断请求，并且中断是允许的(IF=1)时，处理器将做些什么。在 IF=0 时，处理器接收到一个在 NMI 引脚上的中断时，它将做些什么？
22. 8086 是利用什么机构来处理被 0 除的？
23. 为什么使用 CALL 的方法调用中断程序不是一种好的方法？INT 指令的功能是什么？
24. 使用 INT 指令来设置断点有什么好处，为什么必须要用 1 字节 INT 指令，用 2 字节 INT 指令有什么坏处，试举例说明之。
25. TF=1 使程序进入单步执行，试说明是什么机构使程序得以单步执行的。
26. 在用调试程序调试一个程序时，从地址 100 开始希望用单步方式执行，试叙述全部过程。
27. 协处理器是怎样获得所要进行的操作、操作数及地址的？
28. 举出一个不使用 LOCK 前缀就会发生冲突的例子。
29. 8086 设计中有一个缺陷，就是当其他前缀和重复前缀配合做串操作时，中断以后，前一个前缀会丢失，为什么会丢失？怎样做才能不丢失？
30. INC, DEC 指令实质是加法和减法指令，但它们却被设计得不影响进位标志，为什么？
31. 8086 是用什么途径来更新 CS 和 IP 的值的？有哪些指令可以用于这个目的？

第四章　8086 系统设计

在微处理器出现之前，计算机用户往往只能买一个完整的系统。这种系统一般都作为通用计算机系统来使用，以解决各种各样的问题。同时，这种系统常常是太大和太贵，因此很难专用于某种单一系统。微处理器的小体积和低价格，使得构造专用于某种特殊单一的系统成为可能。例如，一台现金出纳机，能够由安装在该出纳机机箱中的一个专门设计的计算机系统来控制。用户再也用不到去买一个完整的大系统了。相反，他可以买到能构成专用系统的器件，然后把这些器件按他的特殊需要装配起来即可。这和立体声爱好者，按照他的需要来装配一组音响设备的情形很相似。

这一章将介绍能用在 8086 系统中的一组器件系列，并且将论述这些器件怎样装配起来，才能构成一个完整的系统。

除了基本逻辑零件——AND 门、OR 门和反向器的知识以外，只要很少的数字设计知识就行了。

§4.1 总 线 结 构

在第一章中，我们已经知道，8086 是一个微处理器，而不是一台微型计算机。正如我们所知道的，这两者之间的区别在于微处理器并不包含任何存贮器单元或输入/输出转接口。形象地说，微处理器能够思考，但不能记忆，也不能听或说。这就要求用一些附加的部件和微处理器一起，才能构成一台可用的微型计算机。图 4.1 说明了一个微型计算机系统所必须的四大部件。

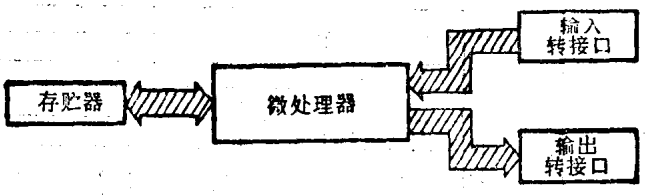


图 4.1　一个微型计算机系统

信息(数据)沿着称为数据总线的通路，从微型计算机系统中的一个部件，传送到另一个部件。一种具有代表性的结构是，只有一条数据总线，并且它是与系统中所有部件共享的，微型计算机发出允许各种部件轮流使用数据总线的控制信号。图 4.2 说明了这种情况。

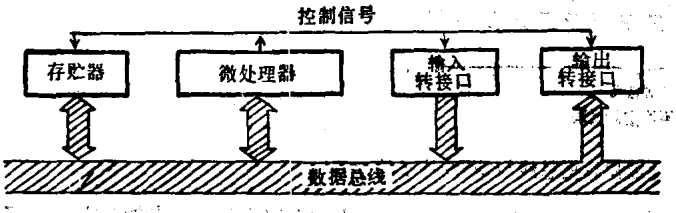


图 4.2　一个单总线系统

只告诉一个部件，如存贮器，它可以使用数据总线还不足以启动它正常工作。存贮器必须知道是哪一个单元要与数据总线打交道。微处理器发出该存贮器单元的地址，并把它放在第二条被称为地址总线的公共总线上。因而形成了一个带有数据总线、地址总线和控制信号的微处理机系统。图 4.3 表示了这种系统。

数据总线、地址总线和控制信号都来源于微处理器。因此，有必要仔细地考察一下 8086

微处理器，看看它有些什么种类的总线和控制信号。由于8086是一个16位处理器，它应该有一个16位宽的数据总线，因而在一次存贮器访问中，它能一次读或写一个完整的字（16位）。另外，由于8086能寻址2^{20}（大约1兆）字节，所以就需要一个20位宽的地址总线。由于8086被封装在一个有40条引脚的双列直插式管壳内，所以处理器和系统中的其他部件之间，只能有40个连接点。如果这些连接点中的36个被地址和数据总线占用了，剩下的4条引脚对于所有的控制信号，电源和接地的连接就不够了。为了减少地址和数据总线使用的引脚数，就设法把这些总线通过如图4.4

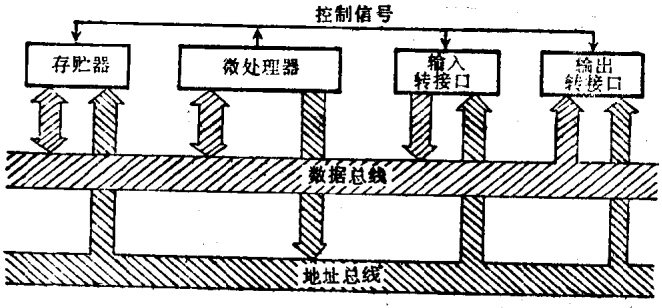


图4.3 带有地址总线和数据总线的完整的微型计算机系统

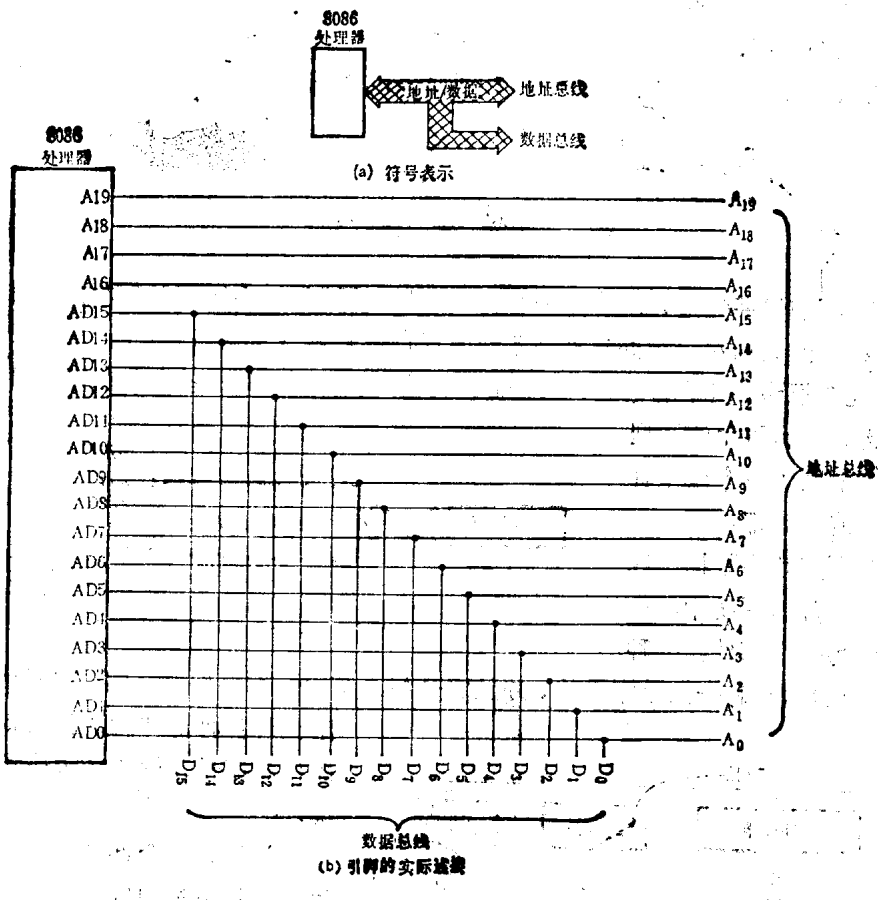


图4.4 共享地址和数据总线与8086片子的连接

所示的一集公用引脚，从处理器中引出来。这样，虽然需要把地址锁存（在下一节叙述）而给系统的其余部分稍微增加了一点复杂性，但却能使引脚数压缩在40条以内，从而降低了8086的成本。另外从第二章中知道8086具有预取指令队列，因此这样安排对运算速度也不会带来很大的影响。

§4.2 地址锁存

让我们考虑数据是怎样从处理器发送到存贮器的。在某一时刻，处理器把某个特定存贮器单元的地址，发送到地址总线上。在稍后的时刻，处理器又把数据发送到数据总线上。但是由于这两个总线共享着某些引脚，所以处理器无法在发送出数据的同时，又发送出地址。因此除非想办法把地址摘记下来，否则它必然会丢失，从而造成数据将不知道向哪里传送。

8086 系列中有一个称为 8282 的锁存器，它能用来担当"摘记"的任务，即用来存贮那些将被丢失的地址。8282 有 8 个数据输入引脚和 8 个数据输出引脚。当在进行地址锁存时，8282 将"记住"在输入引脚上的数据。在一个允许进行地址锁存的信号被送到它的选通输入端 STB (strobe)上时，8282 就能完成锁存输入端数据的任务。另外，当把一个信号送给它的 OE (output enable)，即允许输出控制引脚时，8282 就会把锁存的内容传送到输出引脚上。8282 的数据流向表示在图 4.5 中。

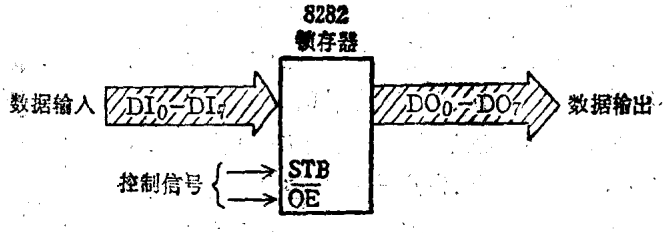


图 4.5 8282 锁存器的符号表示

严格地说，在 8282 片子上的引脚应标记为 $\overline{OE}$，而不是 OE，这表示该引脚的功能是低电平有效。为了简单起见，这种细节一般在原理性介绍中省略(图中不省略)。

显然，这种 8282 锁存器，正是我们需要用来使地址保持不丢失的中规模集成电路器件。在处理器正把一个地址，送到公用的地址/数据总线上时，8086 在它的允许地址锁存引脚 ALE (address latch enable) 上发出一个控制信号，从而把这件事通知每一个部件。这个 ALE 信号正是 8282 为了锁存一个地址所需要的选通信号，8086 和 8282 之间的连接表示在图 4.6 中。

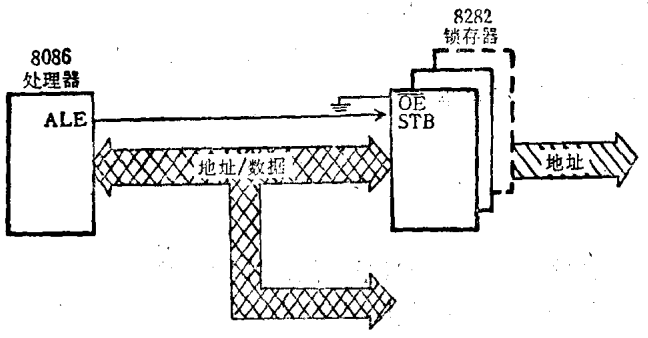


图 4.6 使用锁存器把地址从公用的地址/数据总线上分离出来

图 4.6 中使用了三个 8282 锁存器，因为 8086 的地址是 20 位的，而一个 8282 只能存贮 8 位。在只有小容量存贮器的系统中，并不是所有 20 个地址位都被使用的，于是，地址锁存器中的某一个可以省去。

§4.3 数据功率放大

地址锁存是必不可少的，因为当处理器到了读或写数据的时候，它不可能再发送出地址，但是，锁存数据则没有必要。值得注意的是，处理器的输出或接收数据的负载能力也是有限的。例如，如果有太多的部件挂在数据总线上，而每个部件都想要接收数据时，8086 就可能没有足够的功率把数据送给它们全体了(如果不使用地址锁存器，对于地址也将碰上类似的问题)。解决的办法是使用一个数据功率放大器，用它来接收数据，经过放大以后再传送给任何需要该

数据的部件。使用这种功率放大的唯一困难是，这种放大必须是能双向进行的：因为数据既要从处理器流向系统的其余部分，也要从系统的其余部分流回处理器，能在两个方向进行传送和接收的放大器称为收发器。

8086 系列中有被称为收发器的中规模集成电路片子 8286。8286 有 8 个用作数据输入的引脚和另外 8 个用作数据输出的引脚。但收发器可以交换这两组引脚的任务，以便数据能以任一方向通过该收发器。这种片子也有两个控制引脚，其中一个是允许输出引脚 OE (output enable)，它控制收发器 8286 何时传递数据；另一个是传送引脚 T (transmit)，它控制 8286 中的数据按哪个方向传送。8286 的方框图表示在图 4.7 中。

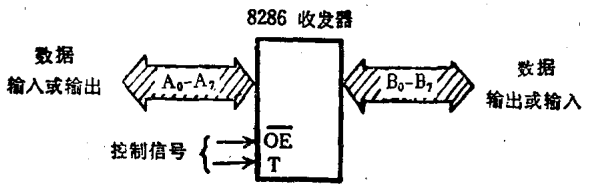


图 4.7 8286 收发器的符号表示

显然，8286 就是我们要用来把数据总线上的信息进行功率放大的理想的器件。当 8086 正在公用的地址/数据总线上传递数据的时候，8086 在它的数据允许引脚 DEN (data enable) 上，发送出一个控制信号，以通知在系统中的其他器件，它正在传递数据。在此同时，8086 还在它的数据传送/接收引脚 DT/$\overline{R}$ (data transmit/receive) 上，发出另一个控制信号，指明数据是从处理器流向系统的其余部分还是从系统的其余部分流向处理器。8086 处理器，8282 地址锁存器和 8286 收发器之间的连接表示在图 4.8 中。收发器用虚线表示着，因为在较小的系统中，它可能不需要。

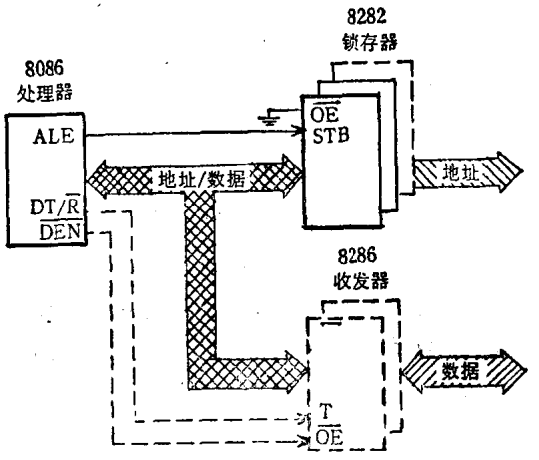


图 4.8 使用收发器推动数据总线

§4.4 定　　时

由 8086 处理器执行的每一种功能，其执行的时序是相当重要的。让我们稍微详细地考察一下地址的锁存过程。8086 处理器把某个地址发送到总线上，并通过发送 ALE 信号把这件事通知 8282 (一个 ALE 信号，实际上是在 ALE 引脚上发送一个脉冲，但是不要让我们纠缠在这样的细节上)。如果处理器同时发送 ALE 信号和地址，锁存器在接收到 ALE 信号后，就会在所有的地址位上锁存尚未完全到达稳定状态的地址码。所以，在处理器把地址送到总线上去的时刻和发送出 ALE 信号的时刻之间必须要有些延迟。这个延迟无疑是很短的 (比一个微秒还要短得多)，然而为了保证地址的稳定却是必须的。

处理器用时钟脉冲来计量延迟。所谓时钟脉冲是从一个被称为时钟发生器的时序电路中取得的。就象一个节拍器一样，时钟脉冲为计量时间提供了一种参考的基本单位。如果时钟脉冲以每秒钟一个的速率到达，则 3 个时钟脉冲的延迟将是 3 秒。但是，如果使用一个较快的时钟发生器，使得在 1 秒钟中能到达 1 兆个时钟脉冲，则 3 个时钟脉冲的延迟将只有 3 个微

秒．由此可见，时钟越快，延迟就将越短．如果我们把时钟不断地加快，就必然会到达这样一个临界点，在这点之前，系统能正确地运行，在这点之后，系统则不能正确地运行．这一点的时钟频率就是系统的理论最高时钟频率（实际最高时钟频率当然要稍低些）．使 8086 系统仍能正确地运行的最高时钟频率是 8 兆赫兹，也就是每秒钟大约有 8 兆个时钟脉冲．

8284 时钟发生器，是一个产生时钟脉冲的集成电路．脉冲发生的速率由一个连接到 8284 两个引脚上的石英晶体来决定（象使用在电子手表中的情形一样）．为了可靠，8284 对每三个来自晶体的脉冲，产生一个时钟脉冲．这样，为了产生每秒 8 兆的时钟脉冲，就要使用 24 MHz（兆赫兹或兆周/秒）的晶体．图 4.9 表示了 8284 时钟发生器如何连接到 8086 中去的情况．

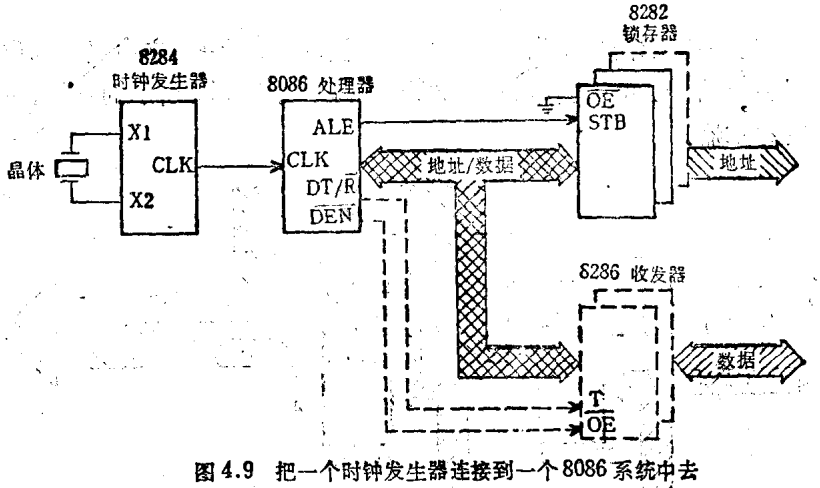


图 4.9 把一个时钟发生器连接到一个 8086 系统中去

§4.5 存贮器部件

我们已经讨论了地址总线和数据总线，下面让我们看看如何把一些存贮器挂到总线上去．我们已经知道，存贮器可以分为两种：一种是只读存贮器，这种存贮器单元中的信息是预先"烧入"的．因此，这种存贮器只能被读出而不能被写入，简写为 ROM 的只读存贮器的名称就是这么得来的；另一种存贮器可以随机地读出或写入信息，所以被称为读写存贮器或简写为 RWM．由于这种表达的发音困难，某些人就决定称它们为随机访问存贮器，并把它们简写为 RAM．请注意，不要让这一点迷惑了，实际上两种存贮器都可以被随机地访问！

让我们把 2716 只读存贮器作为 ROM 的一个例子．这种存贮器片子含有 2K (K=1024) 个字节单元，每个字节含有8位．因此，它有时被称为 2K×8 ROM 片子．这种片子含有 11 个地址引脚和 8 个数据引脚．在接到命令时，这种片子将取出由地址引脚指定的字节单元中的内容，并且把该信息送到数据引脚上．2716 有一对控制引脚，一个是片子允许引脚 CE (chip enable)，而另一个是输出允许引脚 OE (output enable)．这种片子的方框图表示在图 4.10 中．

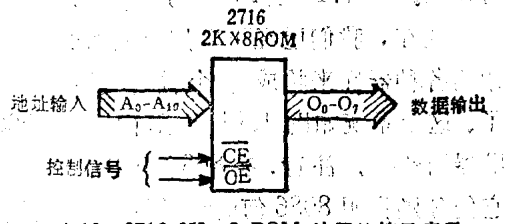


图 4.10 2716 2K×8 ROM 片子的符号表示

较大容量存贮器（较多的存贮单元），可以把若干个 2716 组合在一起而得到．例如，一个 4K 字节的存贮器，可由两片 2716 组成．在这种

情况下，地址码应该是 12 位长。显然，应把低 11 位地址同时送到两个片子上去，但在同一时刻，只能有一个片子被选中。该地址的最高位就可以用来决定选择哪一个片子。这些情况可用图 4.11 来说明。同理，更大的存贮器可以组合更多的 2716 片子来获得。在这种情况下，高于第 11 位的那些高位地址，可能被用来决定选择哪一个片子。这种选择称为地址译码。

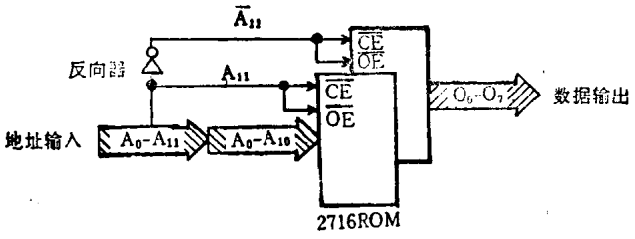


图 4.11 组合两个 2K×8 存贮器以形成一个 4K×8 存贮器

存贮器也可以用增加存贮器片子的办法来加大数据的宽度(每个字由更多的位组成)。例如，2 片 2716 可以组合成 4K ×8 的存贮器，同样也可以如图 4.12 所示的那样，用 4 片 2716 来构成 4K ×16 的存贮器。

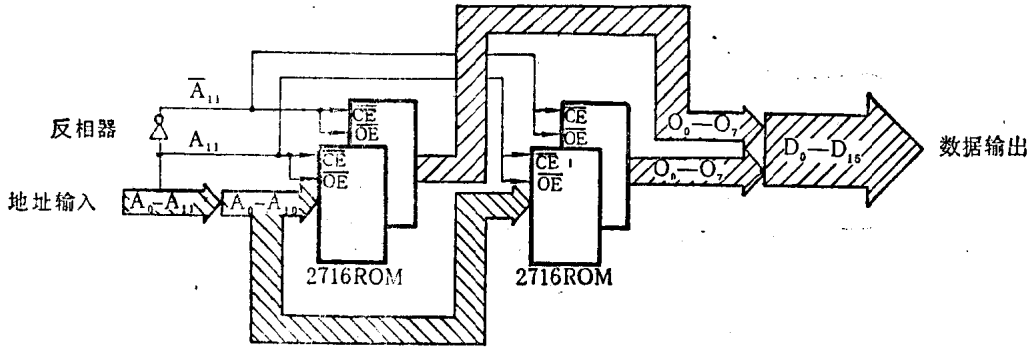


图 4.12 组合 4 个 2K×8 的存贮器构成一个 4K×16 的存贮器

我们把 2142 存贮器片子作为 RAM 的例子。2142 存贮器片子含有 2^{10}(1024)个单元，每个单元 4 位。因此这种片子被称为 1K ×4 的 RAM。这种 RAM 片子含有 10 个地址引脚，4 个数据引脚，还有若干个控制引脚。控制引脚中的一个片选引脚 CS (chip select) 用来选择片子。一个没有被选中的片子只是空挂在总线上而已。另一个控制引脚，即允许写引脚 WE (write enable)，它使数据引脚上的内容，被写入由地址引脚所指定的存贮单元。还有一个不允许输出控制引脚 OD (output disable)，它用来决定是否把被选中单元中的内容，输送到数据引脚上去。WE 在写入时使用，而 OD 则在读出时使用。图 4.13 是 2142 的方框图。

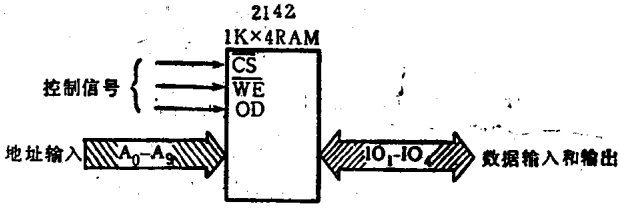


图 4.13 2142 1K×4 RAM 片子的符号表示

现在，我们已经可以用上面所介绍的各种器件来构成一个简单系统了。这个系统如图 4.14 所示，由一个 8086、16 位的 4K ROM 存贮器和 16 位的 4K RAM 存贮器所组成。注意，在 8086 上出现了两个新的输出控制信号，即 RD(read) 和 WR (write)。它们分别指明 8086 何时读数据总线上的内容，以及何时把信息写到数据总线上去。这些信号是用来控制存贮器的。还有另一个控制信号 BHE，留待后面解释。

现在让我们进一步分析，看看当处理器执行一条指令时，会发生些什么。考虑一条把 AL

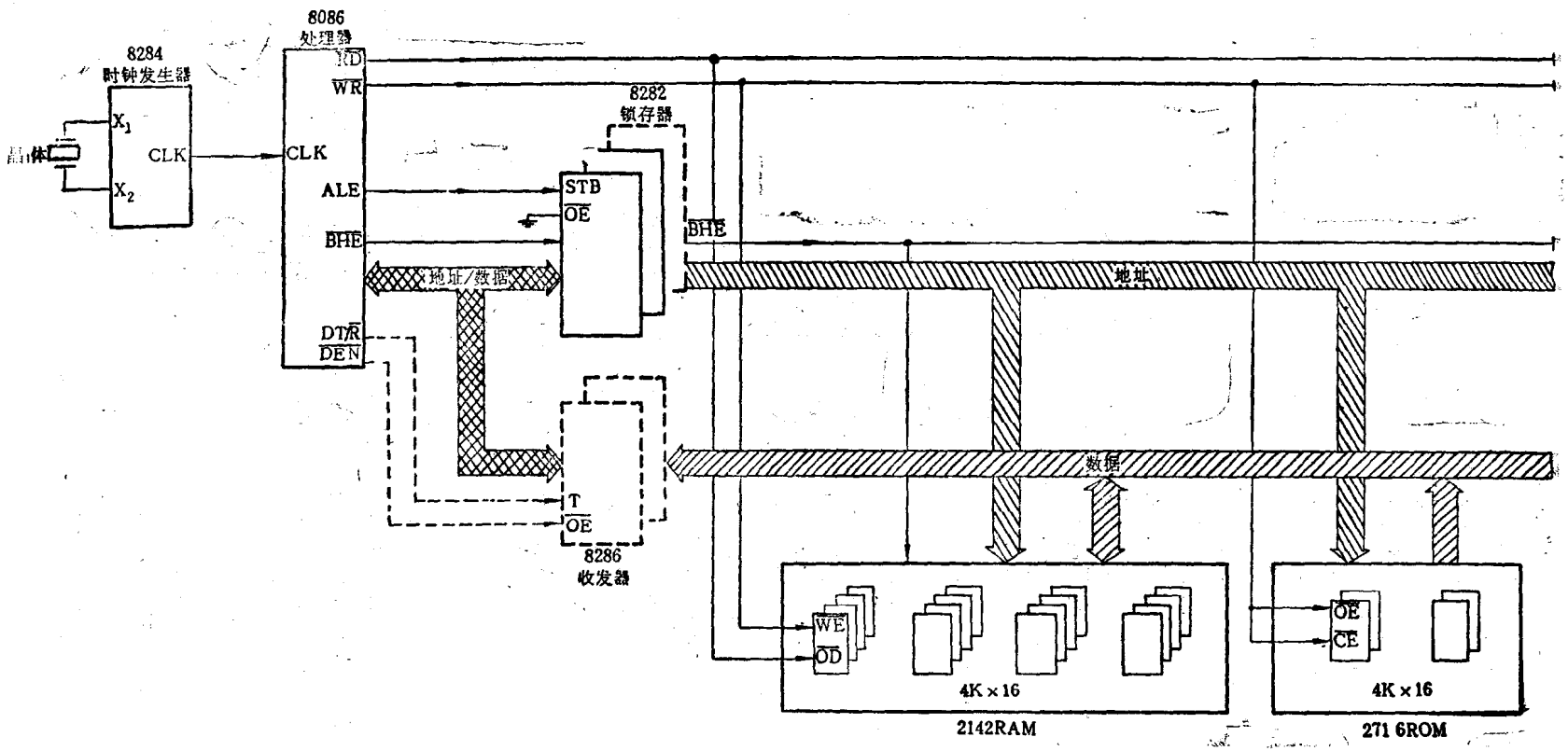


图 4.14　一个带有存贮器的 8086 系统

寄存器中的内容，传送到地址为 0AF0H 的存贮器单元中去的指令。假定 AX 中的内容是 0F307H，则意味着 AL 中的内容是其中的"07"，而 AH 中的内容则是其中的"F3"。首先，处理器把地址 0AF0H 送到公共的地址/数据总线上。然后，处理器再在其 ALE (地址锁存允许)引脚上输出一个信号，以通知地址锁存器锁存该值。这样，0AF0H 就被保存在锁存器中，同时在地址总线上发出相应的地址码电平。这时，处理器可以从公共总线上去掉 0AF0H，并把 0F307H 送到其上。接着，处理器在其 WR (写)引脚上输出一个信号，以通知存贮器去取得公共总线上的低 8 位数据，并且把它们写入在地址总线上的地址所指的那个单元中去。这样，存贮器就把 07 写入地址为 0AF0 的字节单元中去了。

存贮器是怎么知道处理器正在执行一个字节传送而不是字传送的呢？特别是，存贮器是怎么会知道不把 F3 写入地址为 0AF1H 的字节单元中去的？答案很简单：还有一个我们没有解释的控制信号。这个控制信号来自处理器的总线高 8 位允许引脚 BHE (bus high enable)。这个信号由 8086 在把地址送到公共总线上的同时发出，并且象地址一样，它也被地址锁存器所接收并锁存起来。BHE 信号的作用是通知存贮器是否要访问数据总线的高 8 位。在前面的例子中，BHE 引脚没有送出信号，因此存贮器只接收并写入低 8 位"07"。

BHE 信号，只在处理器写入存贮器的时候才需要。当处理器在读存贮器时，存贮器根本不必知道处理器是在执行一条字节指令还是字指令。因为存贮器一次总是送出一个 16 位的字，而要使用该字的哪个部分，则由处理器决定的。这样，正如我们在图 4.14 中所看到的那样，对 ROM 存贮器就不必发送 BHE 控制信号了。

显然，8086 并不需要一个总线低 8 位允许信号 BLE (bus low enable)，因为地址最低位的反码，就可以充当这个角色。换一句话说，发送一个奇地址，就将禁止存贮器访问在数据总线上的低 8 位，而发送一个偶地址则不禁止。如果要把一个字节传送到存贮器的奇地址单元中去，则处理器必须发送出该奇地址(不允许访问数据总线的低 8 位)；并发送一个 BHE 信号(允许访问数据总线的高 8 位)；还要把所需要的数据发送到数据总线的高 8 位。同理，处理器发送一个偶地址 (允许访问数据总线的低 8 位)；但不发送 BHE 信号 (不允许访问数据总线的高 8 位)；并且把所需要的数据送到数据总线的低 8 位即可向存贮器中的偶地址传送一个字节。对于 16 位数据的偶地址传送，就比较简单。处理器发送出一个偶地址 (允许访问数据总线上的低 8 位)；并发送 BHE 信号 (允许访问数据总线的高 8 位)；再把所需要的 16 位数据发送到并联的两个 8 位数据总线上去，就能把一个整字传送到存贮器的偶地址字单元中去。

最后，让我们考虑处理器怎样把一个 16 位字，传送到存贮器的奇地址字单元中去。这个字传送，需要两次存贮器访问。第一次访问时，由处理器发送出该奇地址 (不允许访问数据总线的低 8 位)，并送出 BHE 信号 (允许访问数据总线的高 8 位)；然后由处理器把该字的低 8 位传送到数据总线的高 8 位(注意，发生了字节调换)。在该次访问完成之后，处理器再发送出一个由把该奇地址加 1 而获得的偶地址 (允许访问数据总线的低 8 位)，但不发送 BHE 信号(不允许访问数据总线的高 8 位)；并且把该字的高 8 位发送到数据总线的低 8 位(又是字节调换)。这样，经过二次存贮器访问，才把一个 16 位字写到了存贮器的奇地址字单元。由此可见，在条件允许的情况下，我们应当尽量避免这种传送。图 4.15 表示了把信息传送给存贮器的各种方法。

象 BHE 信号一样，地址的最低位也不需要发送给 ROM 存贮器。

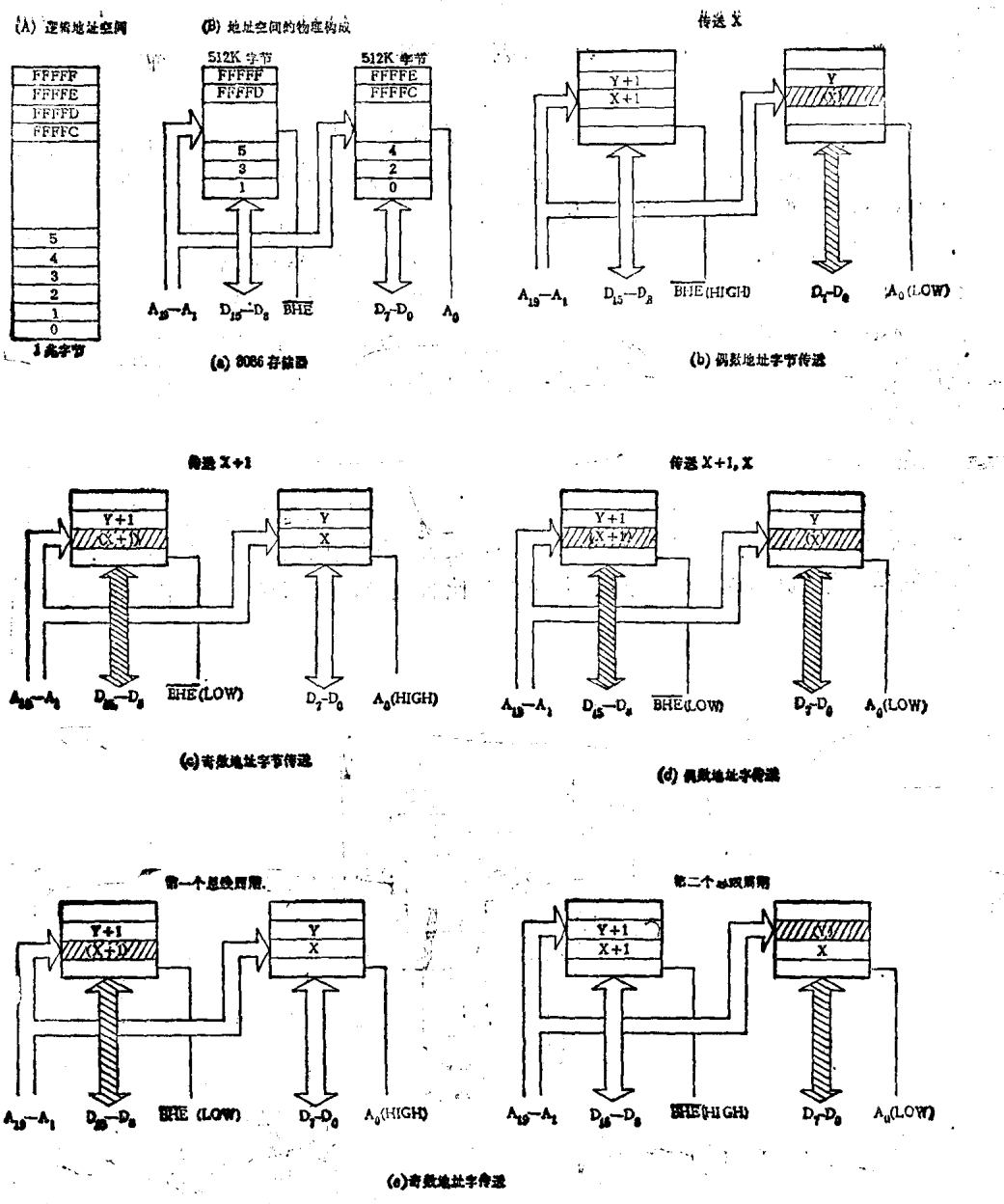


图 4.15　向存贮器奇地址或偶地址传送字节或字

§4.6　输入/输出转接口

现在到了把一些输入/输出转接口引入系统的时候了。输入/输出转接口就象存贮器一样，挂在地址和数据总线上。这就意味着处理器能够把地址发送到地址总线上来，以选择一个特定的转接口，并能通过数据总线和这些转接口互相传送数据。8086 用一个 $M/\overline{IO}$ 控制信号来区分存贮器指令，如 MOV 和输入/输出指令、IN 和 OUT。但是，正如在第二章中已经提到过的，在 8086 的小系统中，把某些存贮器地址转换为输入或输出转接口地址，然后用存贮器

指令，而不是输入/输出指令来与这些转接口交换信息却更简单。这种方式常常称为存贮器映象输入/输出。

我们用一个控制交通灯的 8086 系统作为存贮器映象输入/输出结构的例子。这个 8086 系统控制着两组交通灯。其中，每组交通灯由一盏红灯，一盏黄灯，一盏绿灯和一个左转弯箭头所组成。这样，就必须有四种信号从 8086 系统通向每组交通灯的控制部件。换句话说，就是需要一个输出转接口去控制交通灯。在这里，所谓的输出转接口，就是一种存贮由系统发来的对交通灯的控制信息，并能持续不断地把这种被存贮的信息馈送给交通灯控制部件的设备。在这种应用中，8282 锁存器可以成为满足这种要求的理想的输出转接口片子。假定系统中的存贮器使用 2^{10} (1024)个存贮器地址。换句话说，就是存贮器的最低地址是 0，而最高地址是 1023。这样，我们就可以把输出转接口放在存贮器地址 1024。当 1024 ($A_{10}=1$，其余 A_0—A_9 都是 0) 出现在地址总线上时，对 1024 响应的译码器，就会通知输出转接口去取得在数据总线上的数据。显然，在这种具体应用中，把 $A_{10}=1$ 作为该转接口的地址，就可以使结构简化。但是，这样一来，必然会浪费 $A_{10}=1$ 的大量地址，或者说把所有这些地址都贡献给一个转接口了。由于在这个简单的系统中，这些地址实际上并没有什么用处，所以就这么用了。上述例子中的系统表示在图 4.16 中。

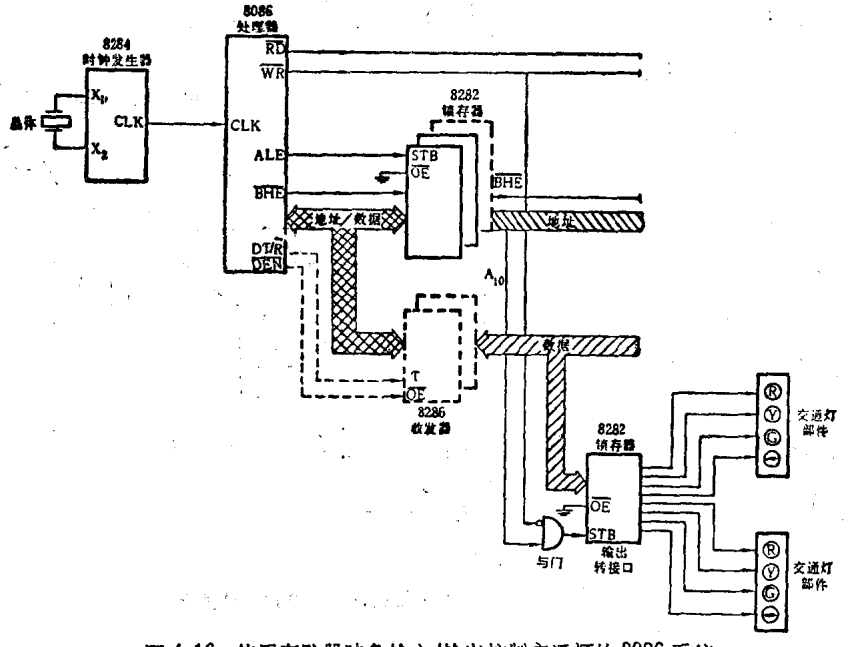


图 4.16 使用存贮器映象输入/输出控制交通灯的 8086 系统

注意那个连接到输出转接口的 STB (strobe)引脚的 AND 门。它的作用是，仅当地址线 $A_{10}=1$，并且 8086 正发送出一个 WR 信号时，才产生一个信号送给 STB 引脚。

上述例子说明了存贮器映象输出的用途。程序将使用存贮器指令去改变交通灯的设置。例如一条把存贮器地址 1024 作为目的操作数地址的 OR 指令，可以用来改变一组交通灯的设置情况，并可做到使另一组不改变。当然两个都改变也是可以的。如果我们想使用 OUT指令 (不用存贮器映象输出)，就需要借助 $M/\overline{IO}$ 输出信号(图中没有表示出)，并把它馈送入 AND 门。

§4.7 中断服务

8086处理器片子，有两条能被外部设备用来进行中断的引脚。这些引脚中的一条是INTR（普通中断，也称可屏蔽中断），形象地描述，该引脚被外部设备用来对处理器说："你有空时能为我服务吗？"。另一条引脚，NMI（不可屏蔽中断），则被外部设备用来对处理器说："现在必须响应我，否则就太迟了！"这两种中断的详细描述可在第三章的关于中断指令的描述中找到。

现在让我们更详细地考察普通中断过程。外部设备在它需要服务的时候，把一个信号送给8086的INTR引脚。一旦8086能够响应（IF=1，并且处理器的现行指令已经执行完），它就对外部设备说："要我为你做什么？"。8086是用把一个信号从它的中断响应引脚INTA（interrupt acknowledge）上输出的方法来说出这句话的。随后，外部设备就把一个在0和255之间的中断类型代码送给数据总线，从而告诉8086要做些什么。经过这样的对话，中断服务就开始了。

这种服务方式在只有一台外部设备时，不会发生什么问题。但是，若有两台或更多的外部设备，而它们中的任何一台都可以请求处理器中断时，就会出现问题。若它们中的两台恰好在同一时刻都要求中断服务，即它们同时把中断请求信号发送到8086的INTR引脚上，由于8086只依据在INTR引脚上是否出现信号（IF=1时）来决定是否响应中断，它就会对两台外部设备都发出INTA信号来响应它们的请求。当然，两台外部设备在收到INTA信号后，都会把自己的中断类型代码放到数据中线上。这样，8086最终取得的就是一个混淆了的中断代码，从而无法进行正常的中断服务。

由此可见，在有多个外部设备的情况下，需要某种能决定哪一个外部设备的中断请求更紧迫，并能把它的请求优先传递给8086的仲裁器（称为中断控制器）。在8086的外围系列片中，这种仲裁器就是8259A。有了中断控制器之后，外部设备只和8259A通讯，然后由8259A再和8086通讯。它们之间的关系表示在图4.17中。

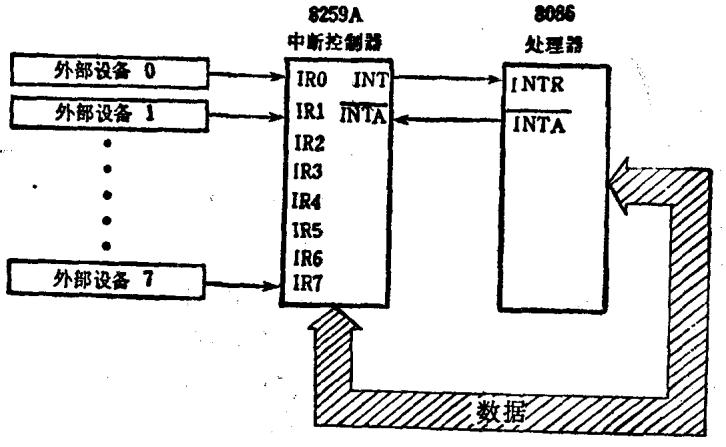


图4.17 8259A作为一个仲裁器

使用多片8259A可以处理多于8个的外部设备。图4.18是怎样处理多至64个外部设备的示意图。从图中可见，一个8259A处于更高一层，它称为主中断控制器，而其余的8259A

则称为从中断控制器。这样，外部设备仅与从中断控制器通讯，而从中断控制器又仅与主中断控制器通讯，后者再与 8086 通讯。

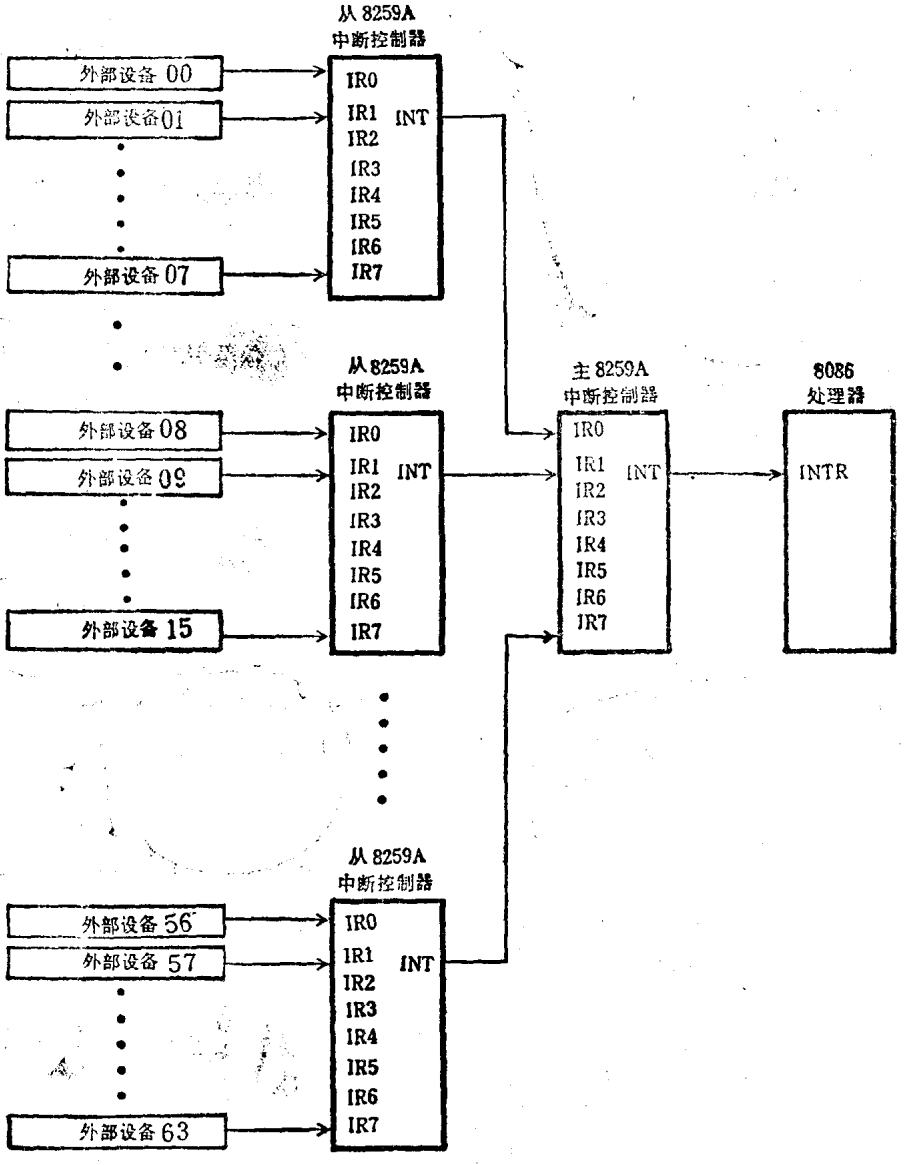


图 4.18　处理多于 8 个外部设备的结构

要使 8259A 执行它的职能，它必须知道与它相连的有哪些外部设备，这些外部设备中的哪一些比较重要，这样才能合理地解决中断时外部设备之间的冲突。另外，它必须知道每台外部设备产生中断的原因，从而能向 8086 传递正确的中断类型代码(即中断原因)。实际上，所有这些信息，都是由 8086 事先以程序的形式写入 8259A 形成的。这就表明 8259A 必定是个十分复杂的大规模集成电路片子。8086 要通过数据总线把信息送给 8259A 来为它“编程”，而 8259A 又要把中断类型代码送给 8086，这就是数据总线被指明是 8259A 的输入/输出通道的原因。关于对 8259A 编程的详细情况，将不在这里介绍，因为它至少需要一章的篇幅才能说清楚，读者可参阅有关的资料。

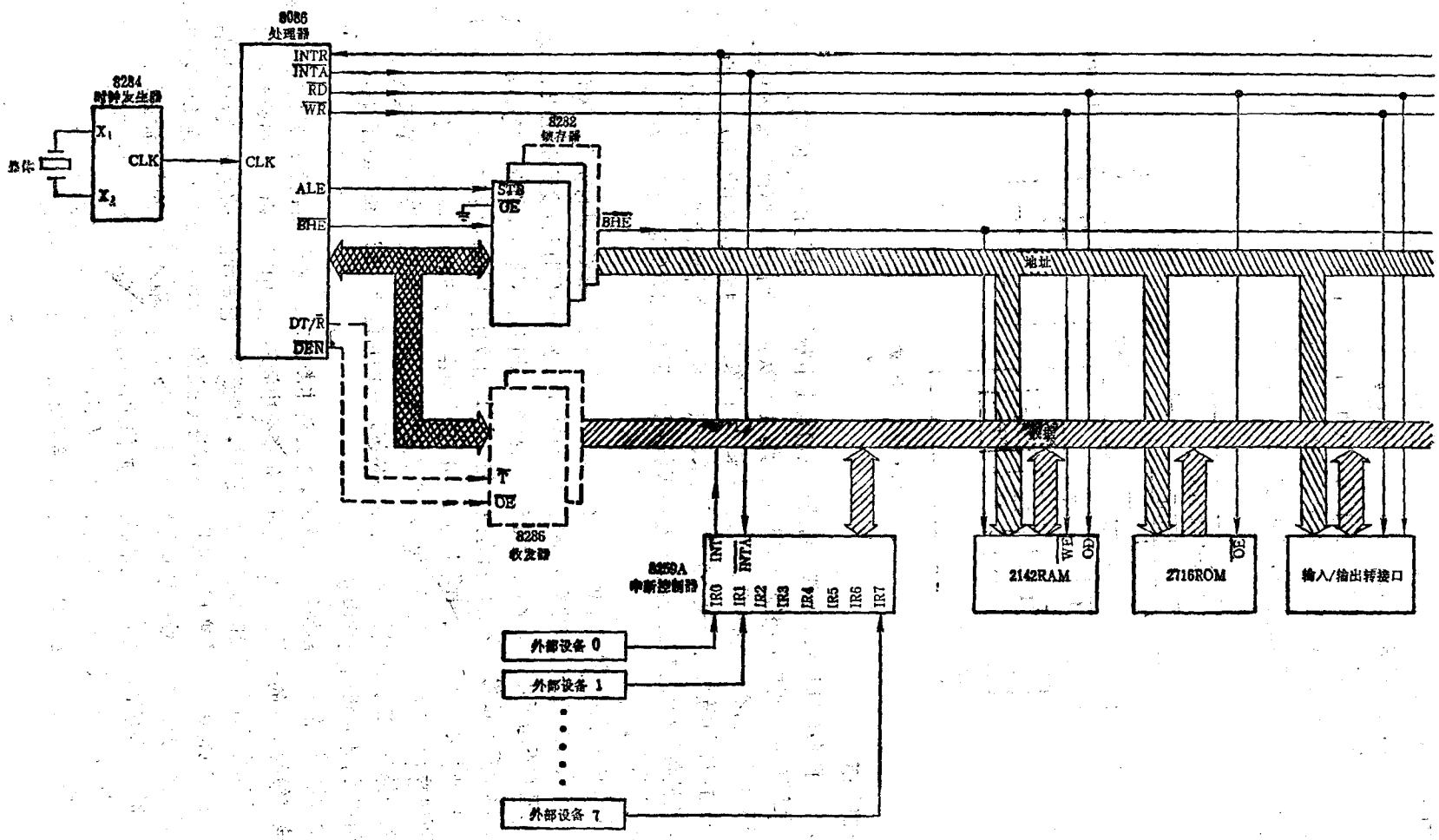


图 4.19　具有中断控制器的 8086 系统

图 4.19 表示了 8259A 如何同我们已经了解的部件进行配合的情形。

现在，我们可以对 8086 的中断机构作一个总结了。8086 能处理 256 种不同类型的中断。中断可以由处理器以外的设备启动；也可由软件中断指令触发（软中断）；在某些情况下，还可以由处理器本身发出。图 4.20 表示了各种中断与处理器的关系，图 4.21 则说明了 8086 对中断的基本响应过程。下面就 8086 的中断机构进行总结。

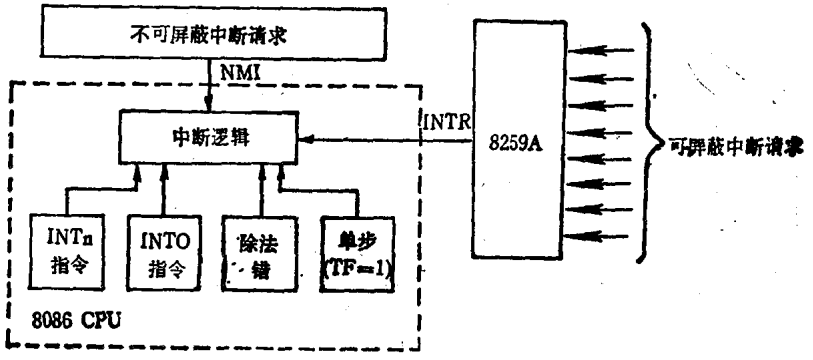


图 4.20 中 断 源

一、外部中断

8086 有两条中断引脚，外部设备通过它们发出中断信号（INTR 和 NMI）。INTR 引脚通常由 8259A 可编程中断控制器（PIC）驱动，它依次把需要请求中断的设备接上去。8259A 是由8086 的软件命令控制的（PIC 对软件来说，好象是一组输入/输出通道）。它的主要作用是接收来自与它相连的外部设备的中断请求，决定哪个请求中断的设备具有最高的优先权级别，若所选设备的优先权高于当前正在被服务的设备时（假如有的话），则向 8086 的 INTR 引脚发送一个信号。

当 INTR 有效时，处理器根据中断标志位（IF）的状态，采取不同的动作。然而，这些动作只能在当前执行的指令完成之后才能发生。如果 IF 被清除（即屏蔽或禁止 INTR 发出的中断），则处理器不考虑中断请求，继续处理下条指令。处理器不锁存 INTR 信号，因此，该信号必须在请求被接受或撤销之前保持有效。如 INTR 上发出的中断被允许（IF＝1），则处理器将响应该中断请求并处理它。INTR 上的中断请求可由执行 STI 指令（置位 IF 为 1）而被允许，也可由执行 CLI 指令（复位 IF 为 0）而被禁止，还可以利用给 8259A 编程的方法，有选择地屏蔽一部分中断。应当注意，为了减少超越堆栈的可能性，STI 和 IRET 指令只能在下条指令结束后，才可重新允许中断。

有少数情况，在下一条指令完成之后，才允许中断请求。重复、封锁和段超越前缀被看作是带有该前缀的指令的一部分，在执行前缀和指令中间不允许中断。MOV（传送）指令和 POP 指令在执行向段寄存器传送的任务时也是这样，即要等在它后面的一条指令执行完以后才允许中断。这种机构保护正在变更到新堆栈的程序（通过修改 SS 和 SP）。假如在 SS 改变后，而 SP 尚未改变时，允许中断，处理器就会把标志位、CS 和 IP 推入错误的存贮器区域。由此可知，无论何时，段寄存器和其他的值都必须一道修改，一般应是先执行修改段寄存器的指令，接着执行修改其他值的指令。还有两种情况，即 WAIT 指令和重复串操作指令，中断请求允许在指令执行中间发生。在这种情况下，待任何完整的基本操作或等待检测周期完了以后，中断

就被接受。

8086 通过执行两个邻接的中断响应 (INTA) 总线周期来响应中断。如果在 INTA 周期内，来了总线保持请求(通过 HOLD 或请求/应答线)，则它要到该周期完成时才能获得认可。此外，若 8086 以最大模式构成系统，则它将在这些周期内激活 $\overline{\text{LOCK}}$ 信号，以通知其他处理

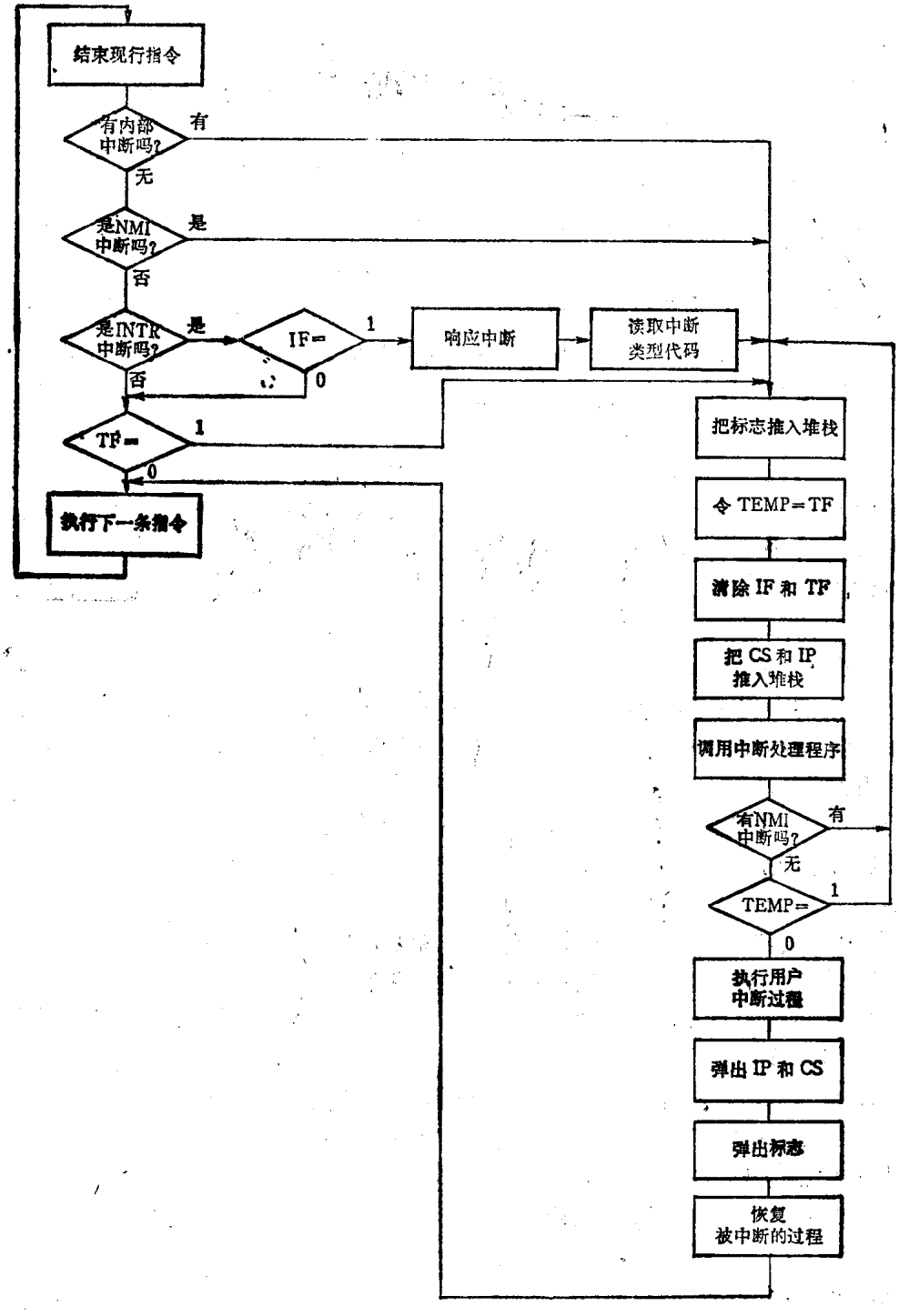


图 4.21 8086 的中断处理顺序

器不要试图获得总线。第一个 INTA 周期给 8259A 一个信号，通知它总线已获认可。在第二个 INTA 周期内，8259A 通过在数据总线上放一个字节来响应，这个字节的内容就是中断类型(0—255)。8086 读出这个类型代码，并用它去调用相应的中断处理过程。这种中断响应的时序在图 4.22 中表示。

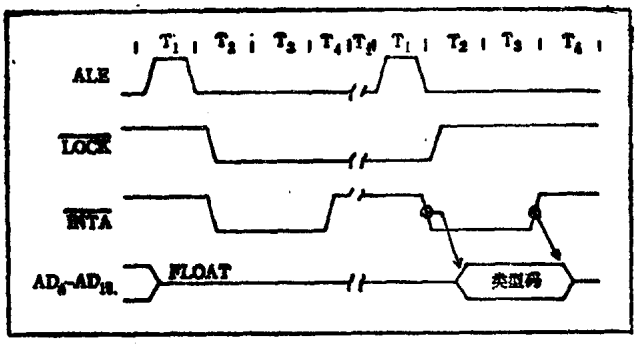


图 4.22 中断响应序列

外部中断请求信号也可由 NMI (不可屏蔽中断) 引脚到达处理器。这条引脚是边沿触发的 (INTR 是电平触发的)，一般用于向处理器发出“灾难性”事件的信号，如电源掉电，存贮器检验错或总线奇偶错。由 NMI 引脚来的中断请求不能禁止，而由处理器锁存，它比 INTR 上的中断请求具有更高的优先权。如果在指令执行期间，两条引脚上同时来中断请求时，NMI 将首先被认可。由于非屏蔽中断预先确定为类型 2，所以，处理器无需配备类型码去调用 NMI 过程，对 NMI 上中断请求的响应也不运行 INTA 总线周期。

处理器认可外部中断请求所需的时间 (中断延迟)，取决于执行完当前指令还差多少时钟周期。一般说来，当执行乘、除、变量移位或变量循环移位指令时，如有中断请求到来，中断延迟最长 (参见 § 3.12 中的有关指令)。个别情况下，最长的延迟就是两条指令而不是一条指令的时间。

二、内部中断

当 INT (中断) 指令执行完了时，立即产生一个中断。指令的中断类型码为处理器提供调用这个中断处理过程所需要的类型码。因为可以规定任何类型码，故用软件中断的办法可以调用所有的外部设备的中断处理程序。如果溢出标志 (OF) 置 1，则 INTO (溢出中断) 指令执行后立即产生类型 4 中断。若 DIV 或 IDIV (除、整除) 指令试图以 0 作为除数时，处理器立即自动地产生一个类型 0 中断。若自陷标志 (TF) 置 1，处理器在每执行一条指令后自动产生类型 1 中断。

所有内部中断 (INT，INTO，除法错，单步) 具有如下特性：

1. 中断类型码包含在指令内，或者是隐含规定的；
2. 不运行 INTA 总线周期；
3. 除单步中断外，其他内部中断不能禁止；
4. 除单步中断外，任何内部中断都比外部中断的优先权高 (见表 4.1)。如果引起内部中断 (如除法错) 的指令执行期间，NMI 和 INTR 引脚上有中断请求时，则优先处理内部中断。

表 4.1 中断优先权

中　　断	优 先 权
除法错、INTn INTO	最 高
NMI	
INTR	
单　步	最 低

三、中断指示器表

8086 的中断指示器 (中断向量) 表，是中断类型码和与该中断类型相对应的中断处理程序

之间的过渡表。中断指示器表占用存贮器的低地址区域。表中有256个入口，对应每个中断类型在系统中的处理程序的入口。每个入口都是一个双字指示器，指示器的较高地址字是处理程序的段基地址，而较低地址字则是从段基地址开始的偏移量。由于每个入口是4字节长，处理器通过简单的乘操作（类型码乘以4），就能正确地计算得到某一中断类型的入口的地址。图4.23是图3.42和3.43的综合，表示了中断指示器表的结构。

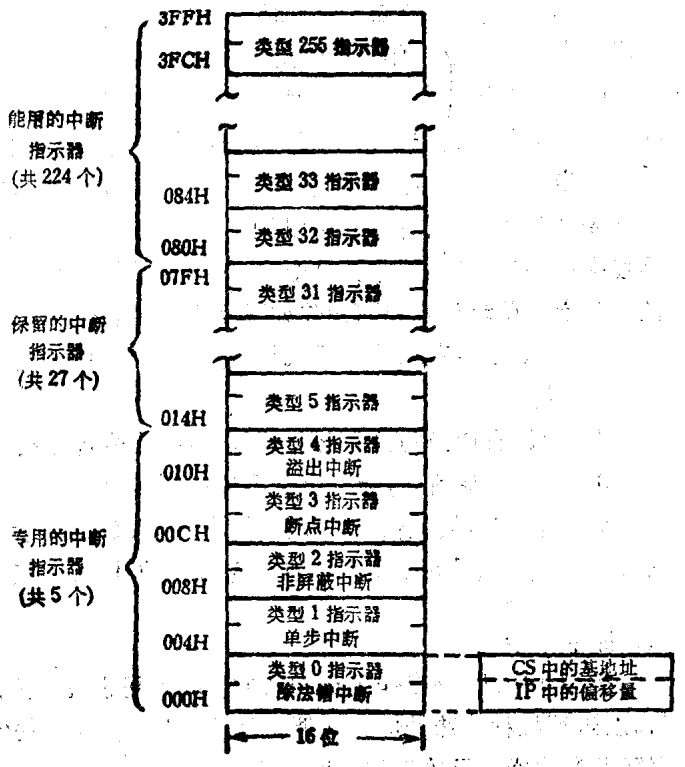


图4.23 中断指示器表

中断指示器表的高端空间，在一个给定不会发生中断的应用中，可以用作其他目的。然而，中断指示器表的专用和保留部分（000H到07FH），却不能用作其他目的，以保证系统的合理操作和对Intel未来硬件和软件产品的兼容性。

四、中断过程

在中断服务开始时，8086把标志位，CS，IP推入堆栈，清除TF，IF。然后用STI指令置位中断允许标志，从而使即将开始的中断处理过程可被来自INTR引脚上的请求所中断（注意，实际上，只有跟在STI后面的指令被执行之后，才能允许中断）。普通的中断处理过程总是可由NMI引脚上的请求所中断。软件或处理器启动的中断也可中断正在执行的优先权比它低的中断处理的过程。必须注意，在正服务于某种中断类型的中断处理过程中，不能再次发生同种类型的中断。例如，在除法错中断过程中，试图用0除会导致无终止的再引入过程。堆栈的空间必须足够大，以适应系统中发生的最大嵌套深度。

象所有的过程一样，中断处理过程对所要使用的所有寄存器的值都应该进行保护，并在处理过程结束前把它们恢复。

所有中断处理过程应当用IRET（中断返回）指令结束。IRET指令保证了堆栈处于和过程引入时相同的条件。IRET从堆栈顶弹出三个字进入IP、CS和标志位，这样就返回到了启动中断时的现场，处理器得以继续执行被中断的主程序。

软件启动的中断过程，可用作系统中其他程序的服务程序（"管理调用"）。在此情况下，当某个程序需要服务时，才启动中断过程，而不是由外部设备启动的（这里所说的"服务"，就是指为文件找一个记录，将信息送给另一程序，请求分配存贮器自由空间等等）。软件中断过程在系统中使用很方便，它在程序执行过程中能够动态地浮动。因为中断指示器表是在固定的存贮单元，过程可以利用软件中断指令，通过中断指示器表相互调用。这种过程之间的通讯，是与过程本身的地址无关的。只要中断指示器表总是由中断类型码来查找，以提供对调用程序

的连接，那么中断处理过程本身就可以移动。

五、单步(自陷)中断

当 TF (自陷标志位) 为 1 时，8086 处于单步工作方式，这时处理器在每条指令执行后自动产生类型 1 中断。作为中断处理的一部分，处理器自动把标志位推入堆栈，然后清除 TF 和 IF。因此，当引入单步中断处理过程时，处理器并不是单步工作方式，而是正常运行的。当中断处理过程终止时，老标志从堆栈中得到恢复，重新置处理器于单步方式。

单步是很有价值的调试手段，它使单步处理过程起到系统"窗口"的作用，通过它来进行逐条指令地观察操作。例如，可以利用单步处理过程打印或显示寄存器的内容、指令指示器 (在堆栈中) 的值以及存贮器关键变量，等等。故用这种方法，能够详细跟踪指令流，并能确定错误的发生点。单步过程还可用作如下几个目的：

1. 当规定的存贮器单元或输入/输出转接口改变其值 (或等于规定的值) 时，写入一个信息。
2. 提供有选择的诊断 (例如只对某些指令地址)。
3. 在提供诊断前，让一个子程序执行多次。

8086 没有直接置位 TF 和清除 TF 的指令。TF 是通过修改堆栈中的镜象标志来改变的。当 TF 的镜象置 1 后，执行单步处理过程的 IRET 指令，处理器就进入了单步工作方式。

六、断点中断

类型 3 中断，是专作断点中断的。断点可设在程序的任何地方。此时，阻止程序的正常执行，以进行某种特殊处理。在调试期间，常在程序的关键点上引入断点，作为显示在该点的寄存器及存贮器单元的值的一种手段。

INT 指令 (类型 3 中断指令) 是一字节长指令，这样容易使它设置在程序的任何地方。

断点指令还可以用来"修补"程序 (插入新的指令)，而不必重新编译或重新汇编。这可以通过保护某一指令字节，并用 INT 机器指令代替它。断点处理过程包含着新的机器指令，处理过程在返回之前，恢复被保护的指令字节，再把保护在堆栈上的 IP 值减 1。在处理过程返回之后，就从被恢复的指令字节开始执行。这样，通过设置断点，利用断点中断处理程序，就实现了在断点所在指令前插入一段新的指令的目的。但是要注意，修补程序只是在特殊情况下采取的措施，另外，修补一个程序需要用机器指令进行程序设计，试图校正现有程序时，很容易加进新的故障。

§4.8 较大系统

从 8086 的能力来说，它可以做许多事，但它只能以40条引脚来做，这是一个矛盾。解决这个矛盾的一个方法是使 8086 收缩它的功能，即不去做它所能做的每一件事，以便与它的较少引脚数相协调。另一个解决办法是给 8086 一些附加的引脚，以便去做它所能做的每一件事，从而充分发挥 8086 的潜力。我们就采用这种方法。8086 运行的模式是由一条称为 MN/$\overline{\text{MX}}$ (minimum/maximum) 的输入控制引脚决定的，该引脚在一个给定的系统中或是永久接地，或是永久接高电平，从而使 8086 处于两种不同的系统工作状态。当这个引脚接高电平时，8086 处在功能收缩的最小模式下运行。实际上，图 4.19 就是一种典型的 8086 最小模式系统。

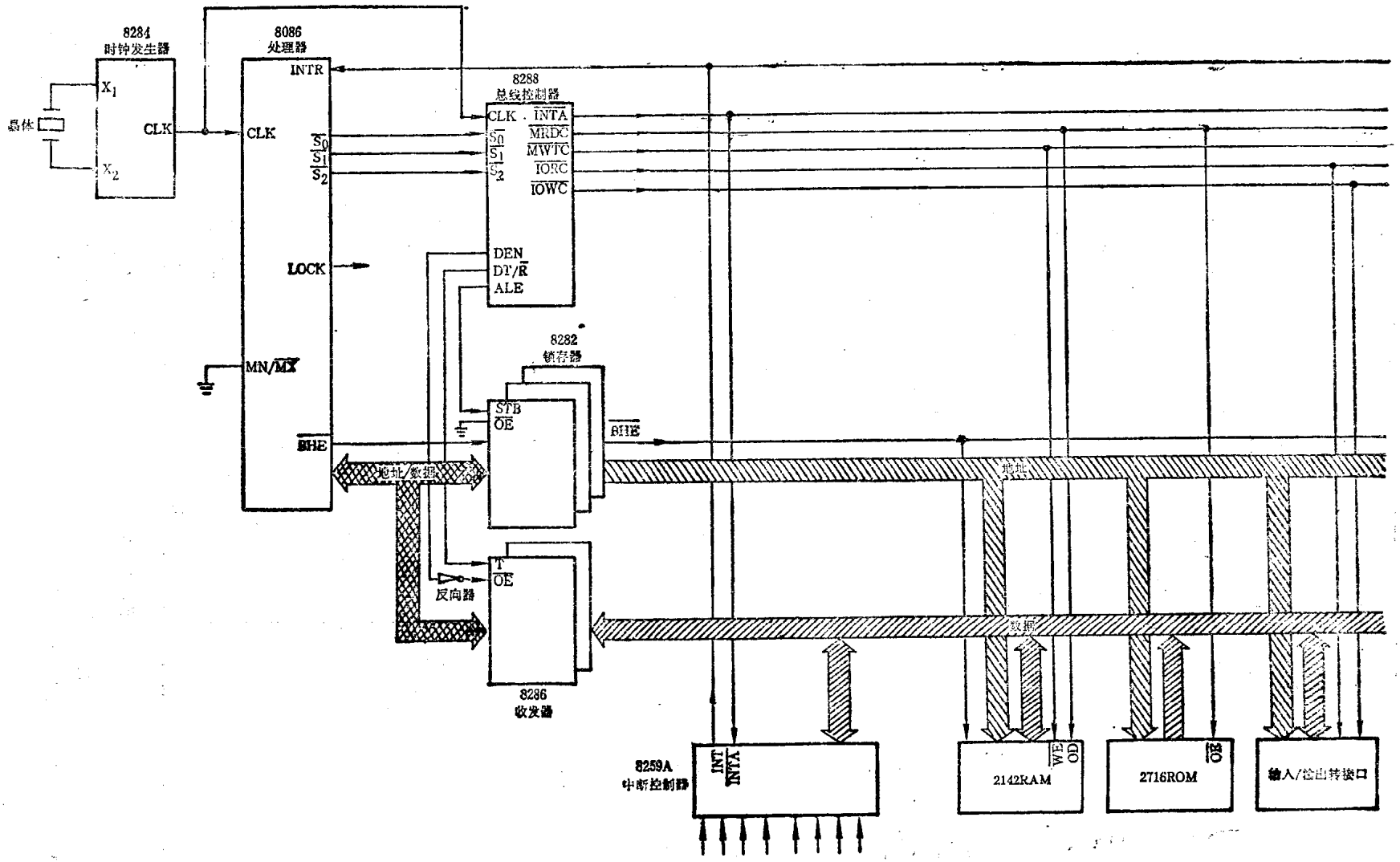


图 4.24　一个最大模式的 8086 系统

当 MN/$\overline{MX}$ 接地时，就表示 8086 有另外一批引脚可以利用，从而扩大了 8086 系统的工作能力。

附加的引脚集实际上需要一片称为 8288 的总线控制器来过渡。8288 接收 8086 发来的 3 个控制信号，通过译码，将它们变换成 8 个控制信号，用来代替某些原来 8086 的引脚功能，从而让那些引脚空出来去执行 8086 的其他功能。例如，ALE(地址锁存允许)信号，现在由 8288 产生，因此，8086 可以使用它原来的 ALE 引脚去实现一个以前不得不隐藏起来的功能。又例如，使若干个处理器能在同一总线上同时运行，而不互相影响的 LOCK 信号 (在第三章中讨论)，是 8086 在最大方式下才能发送出来的信号中的一个，在最小方式下，这个信号是无法发送的。

图 4.24 表示了 8086 的最大模式系统的例子。8086 使用引脚 S_0、S_1 和 S_2 来通知 8288 发送什么控制信号。注意，ALE、DT/R、DEN 和 INTA 信号，它们在最小模式系统中直接来自 8086，现在却来自 8288.另外，来自 8086 的 M/$\overline{IO}$、RD 和 WR 信号已被来自 8288 的存贮器读命令 MRDC (memory read command)，存贮器写命令 MWTC (memory write command)，输入/输出读命令 IORC (input/output read command) 和输入/输出写命令 IOWC (input/output write command) 所强化。这种存贮器读和写信号与输入/输出读和写信号的分离，使得区别存贮器指令和输入/输出指令更加容易。

8284 时钟发生器的输出被馈送入 8288，使 8288 能在地址或数据被放到总线上以后，以恰当的时钟脉冲数定时以后，产生如 ALE 或 DEN 等信号。S_0、S_1 及 S_3 信号使 8288 在规定的时刻，把这些信号放到总线上。

数据收发器和第三个地址锁存器，在最大系统中通常不是任选的，因而在图 4.24 中用实线而不是虚线画出。

§4.9 多处理器系统

随着微处理器价格的下降，多处理器系统 (在系统中运用两个以上的处理器协调工作) 就日益引人注目了。把系统任务分配到各个处理器上并同时进行处理，能大大改善系统性能。这一点对我们来说并不陌生，在第三章关于 ESC 指令的描述中介绍的 8086 和浮点处理器 8087 相互配合进行工作的系统，实际上就是一种多处理器系统。多处理器系统的出现，促进了模块化设计，使系统更易维护和扩充。例如在图 4.25 所示的多处理器系统中，输入/输出操作已由 8089 输入/输出协处理器承担。如果要改变系统中的输入/输出设备 (例如用硬盘代替软盘)，则只要对输入/输出子系统作必要的修改，这种输入/输出设备的改变对

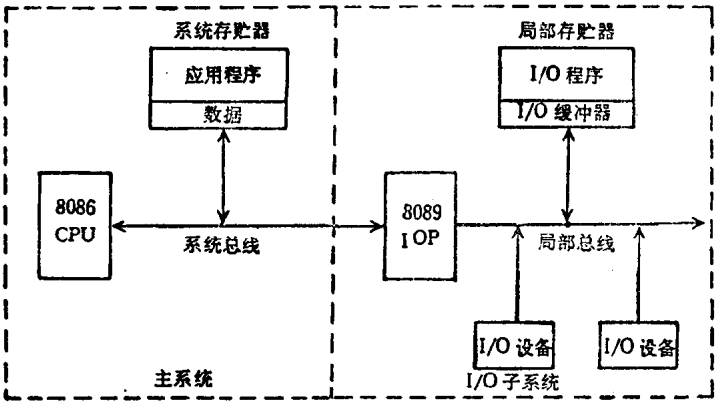


图 4.25 多处理器系统

于 8086 和软件来说完全是透明的.

我们已经知道 8086 是为多处理器环境而设计的,它有一些内部机构能帮助解决处理器间相互协调的问题,而相互协调的问题正是过去研制多处理器系统中遇到的棘手问题.

一、总线封锁

当 8086 构成最大模式系统时,它将提供 $\overline{\text{LOCK}}$(总线封锁)信号. 当 EU (8086 中的执行部件)执行一字节 LOCK 前缀指令时, BIU (8086 中的总线接口部件)就发出 $\overline{\text{LOCK}}$ 信号. 在跟在 LOCK 前缀后面的指令的整个执行过程中, $\overline{\text{LOCK}}$ 信号始终保持有效, 但中断不受 LOCK 前缀的影响. 如果在 $\overline{\text{LOCK}}$ 信号有效期间,有另一处理器请求使用总线(通过请求/应答线),则 CPU 记录该请求,但要在该封锁的指令执行完以后才予以应答.

$\overline{\text{LOCK}}$ 信号在单条指令执行期间保持有效. 如果两条相邻指令各有 LOCK 前缀领先,则在这两条指令中间仍会有未封锁周期. 在重复执行封锁的串操作指令情况下, $\overline{\text{LOCK}}$ 信号在块操作期间保持有效.

当 8086 构成最小模式系统时,不能使用 $\overline{\text{LOCK}}$ 信号,但可以用 LOCK 前缀去延迟响应 HOLD 请求的 HLDA 信号的产生,一直到封锁指令执行完毕后,才产生 HLDA.

$\overline{\text{LOCK}}$ 信号只提供信息,当 $\overline{\text{LOCK}}$ 有效时,共享总线上的其他处理器就不能获得总线. 如果系统利用 8289 总线仲裁器去控制访问共享总线时, 8289 接受 $\overline{\text{LOCK}}$ 信号作为输入,在该信号有效时,它就不放弃总线.

二、等待和检测

8086 (在最大或最小模式中) 能用 WAIT (等待 $\overline{\text{TEST}}$ 信号)指令和 $\overline{\text{TEST}}$ 输入信号与外部事件同步. 当 EU 执行 WAIT 指令时,结果取决于 $\overline{\text{TEST}}$ 输入引脚的状态. 若 $\overline{\text{TEST}}$ 无效, 处理器就进入一个空闲状态, 并在 5 个时钟间隔里反复测试 $\overline{\text{TEST}}$ 引脚的状态. 若 $\overline{\text{TEST}}$ 有效,则继续执行 WAIT 后面的指令.

三、交权

ESC (交权) 指令提供一种方式,使另一个协处理器能从 8086 程序中获得指令或存贮器操作数. 当它同 WAIT 和 TEST 一起应用时, ESC 启动一个"子程序",并使其在协处理器

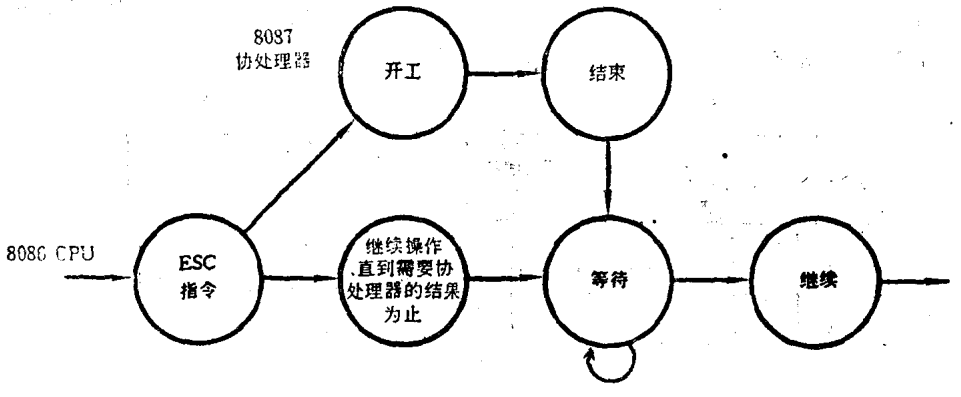


图 4.26 ESC 指令同 WAIT 指令和 TEST 信号的联合使用

中同 8086 并行执行。上述情况可用图 4.26 来说明。

ESC 指令中的 6 位，可以由程序员在写指令时规定。协处理器通过监视 8086 的总线和控制线的方式，在 BIU 取指令时获得 ESC 指令。这样，指令中的 6 位就能指引协处理器去执行某些预定的活动。

如果 8086 构成最大模式系统，则协处理器就通过监视 QS_0 和 QS_1（指令排队状态）来获得 ESC 指令和决定何时执行该 ESC 指令。如该 ESC 指令访问存贮器，则协处理器就通过监视总线来获得操作数的物理地址和操作数。

要注意的是，取得 ESC 指令，并不等于执行它。ESC 指令的前面可以是一条转移指令，该转移指令将使排队机构重新初始化，因而应该由排队机构的状态线决定这种活动。

四、请求/应答信号

当 8086 构成最大模式系统时，HOLD 和 HLDA 引脚的信号演变成更完善的两个信号，称作 $\overline{RQ}/GT_0$ 和 $\overline{RQ}/GT_1$。这些双向信号传输引脚可以使 8086 和另外两个处理器共享局部总线。

请求应答序列是三节拍周期：请求、应答和释放。首先，请求使用总线的处理器向 CPU 的请求/应答引脚发送一个脉冲。然后，CPU 在同一条引脚上返回一个脉冲，说明它正在进入“保持响应状态，并且正在释放总线”。8086 在保持响应期间，BIU 在逻辑上同总线脱开，但 EU 却继续执行指令，直到有一条指令请求总线访问或者排队机构空了，才不再继续执行。当占用总线的处理器用完总线后，它将把一个脉冲发送给 8086，说明请求已经结束，CPU 可以收回总线。

$\overline{RQ}/GT_0$ 比 $\overline{RQ}/GT_1$ 优先权要高。这就是说，若两个请求同时到达 CPU 时，8086 将先应答通过 $\overline{RQ}/GT_0$ 引脚提出请求的处理器，待总线返回给 CPU 后，再响应通过 $\overline{RQ}/GT_1$ 提出请求的处理器。但是，当 CPU 正在处理 RQ/GT_1 上早先的请求时，来自 $\overline{RQ}/GT_0$ 上的请求要等到 $\overline{RQ}/GT_1$ 上的处理器释放总线后，才能获得 CPU 的响应。

五、多总线体系结构

Intel 公司设计了一种通用多处理器总线，称作多总线 (Multibus)，它就是用在 iSBC 单板微型计算机中的标准设计产品。其他许多制造厂家也提供了与多总线结构兼容的产品。当 8086 构成最大模式系统时，8288 总线控制器输出的信号在电特性上与多总线规约兼容。多处理器系统的设计者在设计产品时，可以用多总线结构，以减少研制时间和成本，并可获得与各种现有的 iSBC 产品系列印制板的兼容性。

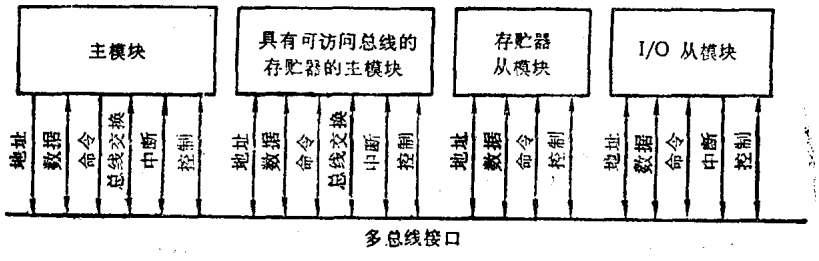


图 4.27 多总线系统

多总线体系结构提供了通用的通讯通道。这些通道能协调各种计算模块(见图 4.27)。在多总线系统中的模块分为主和从两类。主模块获得总线的使用权，并在总线上进行数据传送；从模块只能是数据传送的目的模块。多总线结构允许在系统中混合使用 8 位和 16 位的主模块。除 16 条数据线外，多总线还提供 20 条地址线，8 条多级中断以及控制线、仲裁线。多总线还提供了辅助电源线，以便在正常电源出故障时，将备用电源接到存贮器上。

多总线结构含有自己的时钟，它与连接在它上面的各模块的时钟是无关的。这样，就能使不同速度的主模块共享总线，并可使主模块彼此异步工作。总线的仲裁逻辑，使低速主模块同等地使用总线。然而，一旦模块获得总线，传送速度就取决于模块发送和接收的能力。最后，多总线标准确定在总线上进行通讯的模块的形式参数和物理要求.关于多总线的完整的说明，可参见第二篇中的有关内容。

六、8289 总线裁决器

多处理器系统，需要一种协调各处理器使用共享总线的方法。8289 总线裁决器和 8288 总线控制器一起，为 8086 系统提供这种控制。它同多总线结构是兼容的，而且还可以用在其他共享总线的设计中。

8289 消除了竞争条件，解决了总线争用问题和处理器之间彼此异步操作的匹配问题。在总线上的每个处理器，都分配有不同的优先权.当总线请求同时到来时，8289就解决争用问题，并把总线使用权转让给具有最高优先权级的处理器。在解决总线争用的问题上，8289 可以采用三种不同的优先权处理技术 (详见第二篇)。图 4.28 表示了一个使用 8289 的 8086/8089多处理器系统。

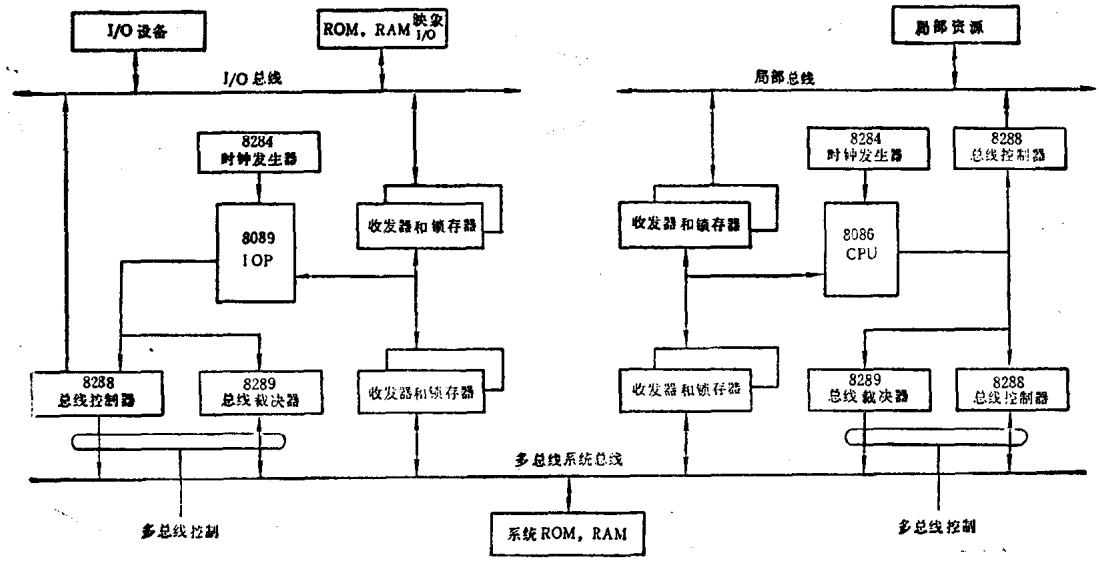


图 4.28　8086/8089 多处理器系统

习　　题

1. 为什么8086要采用地址/数据分时共享，它有什么好处？
2. 8086为什么要地址锁存器和数据放大器，它们的主要功能是什么？
3. 描述一下8086处理器是怎样把一个字节送到奇地址及偶地址的存贮单元中去的，当要送一个字时又是怎样进行的呢？
4. 图4.16是用存贮器映象I/O的方法构成的，试改成用I/O指令的结构。
5. 一个8086系统具有由ROM 32KB（用2716片子）、RAM 64KB（用2142片子）构成的存贮器，试画出恰当的存贮器的地址和数据通路及有关的控制信号（ROM处于低地址，RAM处于高地址）。
6. 在多处理器环境下，8086有哪些内部机构能帮助解决相互协调的问题？试简述它们的操作过程。

第五章　8086 汇编语言

通过前面几章的学习，我们已经对 8086 的体系结构及指令系统有了一个基本的了解，知道如何将 8086 与其他部件组合起来以形成一个完整的系统。也就是说，我们现在所面临的问题就是为这样一个系统编制程序。本章和后面两章将着重讨论 8086 的程序设计问题。

§5.1　目标代码和源代码

我们先来看一个极简单的程序，该程序完成这样一些工作：从 5 号输入转接口读字，再给读进来的每个值一个增量，最后将结果写到 2 号输出转接口。具体程序如下：

存贮器地址 (16 进制)	存贮器内容 (2 进制)	注　　释
00000	11100101	把字读到 AX …
00001	00000101	…从 5 号输入转接口读入
00002	01000000	AX⇐(AX) +1
00003	11100111	把 AX 中的内容…
00004	00000010	…从 2 号输出转接口输出
00005	11101011	跳转重复该过程…
00006	11111001	回跳七个字节
00007		

头两列分别确定了存贮器的地址及其中的内容，实际上处理机所能理解的也仅仅是这种类型的程序。我们通常把这种形式称之为目标代码，把这种由 0 和 1 组成的语言叫做机器语言。只要将目标代码程序放到存贮器当中，就可命令 8086 执行这段程序了。

在第二、第三两章当中，我们已经给出了编写程序的目标代码所需要的全部信息，包括每条指令的格式及指令各字段的编码。因此，至少在理论上讲，我们可以结束对程序设计的讨论。

但实际上，利用 0 和 1 编制程序是一项乏味的、容易出错的、效率极低的任务，虽然计算机能够很好地执行用机器码所编制的程序。因此我们仍不准备用机器语言编写程序，而利用一种我们较为熟悉的语言编制程序，然后再使用计算机将它翻成 8086 的机器语言。利用这种较为熟悉的语言编写出的程序就叫做源代码，把源代码翻译成目标代码的那段程序被称为翻译程序(见图 5.1)。

图 5.1　翻译过程

我们能够使用两种不同的语言来编制源程序代码，它们分别叫做汇编语言和高级语言，与

之相对应的翻译程序分别叫做汇编程序和编译程序。

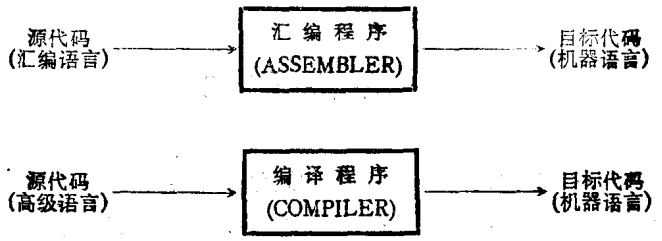


图 5.2 汇编程序和编译程序

汇编语言程序是机器语言程序的一个符号表示，汇编语言程序的语句与其产生的目标代码之间的关系通常是比较明显的。另一方面，高级语言则是一种形式化的、类似于自然语言(特别是英语)的一种方言，高级语言的语句与其目标代码之间的关系通常不是很明显。可见汇编语言给程序员提供了完全控制目标代码的手段，对于一个优秀的程序员来讲，利用汇编语言就能够产生高效的目标代码；而利用高级语言时，就无需考虑产生目标代码，从而可以把注意力集中在分析任务上，至于目标代码的有效程度则完全取决于编译程序。一个很好的编译程序有时能够产生比汇编语言程序更为高效的目标代码，倘若你不是一个很熟练的高效汇编程序员的话，其差别可能会更大。当然你不必为此而感到羞愧，因为我们当中的绝大多数程序员都存在着这样一个问题。

本章的其余部分将讨论 8086 的一种汇编语言 ASM-86，第七章讨论一种可用于 8086 的高级语言 (PL/M-86)，第六章比较详细地介绍了 MCS-86 汇编语言，这几章的内容是相互独立的，可以根据需要先读其中的任意一章。

§5.2 符 号 名

使用汇编语言来代替机器语言的主要优点，就是可以使用符号名，我们可以通过利用汇编语言重新编写上节中的程序实例来说明其优越性。

```
CYCLE:  IN    AX,5      ;read word from port 5 into AX
                        (从转接口 5 中读进字并放入 AX 中)
        INC   AX        ;increment AX (AX+1)

        OUT   2,AX      ;write result to port 2
                        (把结果自转接口 2 中输出)
        JMP   CYCLE     ;keep repeating
                        (跳转到标号为 CYCLE 的那条命令处，重复执行)
```

由于上面这段程序尽可能多地使用了符号名而不是数字，因而阅读、理解起来就比较简单了。例如，目标代码中 4 条指令的操作码分别为 1110010-、01000---、1110011- 和 11101011，而在汇编语言的源代码中它们变成了 IN、INC、OUT 和 JMP。这些操作码的符号名叫做指令助记符，实际上，第三章中引进的那些符号操作码名都是 ASM-86 的指令助记符。ASM-86 汇编程序能够识别这些指令助记符并产生相应的目标代码。

除了操作码字段之外，目标码中还有其他一些字段，这些字段的内容也必须在汇编语言源代码中规定好，以使汇编程序能够产生正确的目标代码．例如，INC 指令具有一个 3 位的寄存器 (reg) 字段，它指出当执行该指令时是哪个寄存器中的内容加 1，在源代码中，寄存器字段的内容是通过给出寄存器的符号名来规定的，就象在 "INC AX" 指令当中这样。ASM-86 中使用的寄存器符号名为 AX、BX、CX、DX、AL、BL、CL、DL、AH、BH、CH、DH、BP、SP、SI、DI、CS、DS、ES 和 SS．

IN 指令和 OUT 指令当中都包含一位的 W 字段和一个 8 位的转接口号码字段，在源代码中转接口号码的确定方法是非常直接的，如 "IN AX, 5" 和 "OUT 2, AX"，而 W 字段的确定却是比较微妙的，利用指令 "IN AX, 5" 和 "OUT 2, AX" 中的 AX 获得 W 字段的信息，输入输出指令总是使用累加器，特别是进行字传送时使用 AX，而进行字节传送时则使用 AL．所以，在 IN 和 OUT 指令当中出现 AX 就表明 W 字段为 1 (ASM-86 协议规定目在源之前，因此 IN 指令中 AX 位于转接口号码之前，OUT 指令中 AX 位于转接口号码之后)．

上面这段程序当中的另一个符号名就是 IN 指令当中的标号 CYCLE，它使得指令 "JMP CYCLE" 能够使控制转移到 IN 指令，汇编程序利用这些信息就能够确定这是一条后跳 7 个字节的指令，并在 JMP 指令的特定字段中填上 a-7．

§5.3 一个完整的程序

在上一节当中，我们给出了一个 ASM-86 程序段，为了使之成为一个完整的程序，还得另加一些语句：

```
1. IN__AND__OUT:  SEGMENT                  (段起始)
2.                ASSUME    CS: IN__AND__OUT
3. CYCLE:         IN        AX,5
4.                INC       AX
5.                OUT       2, AX
6.                JMP       CYCLE
7. IN__AND__OUT:  ENDS                     (段结束)
8.                END       CYCLE          (汇编结束)
```

整个程序都放在 8086 存贮器的某个单独的段中，在汇编过程中，我们并不知道 (当然也不关心) 这样一个存贮段的位置，利用符号名 IN__AND__OUT 我们就可以很满意地指出该段的起始地址．第 1 行和第 7 行规定了段的范围，其中第 1 行引进了名为 IN__AND__OUT 的段，第 7 行则标志该段的结束 (ENDS)．

第 8 行是源程序的结束标志，它告诉汇编程序后面没有指令行需要汇编了，此外它还指出程序应该从标号为 CYCLE 的指令 (第 3 行) 开始执行．由汇编程序产生的目标代码除了含有有关的存贮器中的内容外，还包含该起始地址．

第 2 行中的 ASSUME 语句在此很难解释清楚，而只想给出这样一条规则：在任何包含有代码的段的头上，都必须告诉汇编程序待运行的代码信息放在 CS 寄存器中，至于我们所关心的总是该段的起始地址，因此必须包括这样一个语句：

```
ASSUME    CS: 段名
```

§5.4 ASM-86 程序的结构

现在，先来分析一个更为详细的ASM-86程序，然后再试着导出ASM-86程序的一般结构。在整个这一章中，都将把这段程序当作样本。

```
1. MY_DATA    SEGMENT                  (数据段)
2. SUM        DB      ?                (为 SUM 保留一个字节空间)
3. MY_DATA    ENDS
4. MY_CODE    SEGMENT                  (代码段)
5.            ASSUME  CS: MY_CODE, DS:MY_DATA
6. PORT_VAL   EQU     3                (将符号名定义成 3 号转接口)
7. GO:        MOV     AX, MY_DATA
8.            MOV     DS, AX           (让 DS 指向 MY_DATA)
9.            MOV     SUM, 0,          (把 SUM 清 0)
10. CYCLE:    CMP     SUM, 100         (把 SUM 和 100 比较)
11.           JNA     NOT_DONE         (若 SUM 不超过 100，则跳转到标
                                        号为NOT_DONE 的那条去，否则
                                        做下去)
12.           MOV     AL, SUM          ;… then output SUM to port 3
                                        ((SUM)⇒AL)
13.           OUT     PORT_VAL, AL     (自 3 号输出转接口输出AL中
                                        的内容)
14.           HLT                      ;…and stop execution (暂停)
15. NOT_DONE: IN      AL, PORT_VAL (从 3 号转接口读进数)
16.           ADD     SUM, AL          ; (AL) + (SUM) ⇒SUM
17.           JMP     CYCLE            ;and repeat the test (跳转到标号
                                        为CYCLE 的那条指令去继续检查)
18. MY_CODE   ENDS
19.           END     GO               ;this is the end of the assembly
                                        (汇编结束，并指明程序从标号为GO
                                        的那一条指令开始执行)
```

第 1 行使 8086 在其存贮器的某个地方(不需关心它的物理位置)建立一个段，对应的段名是 MY_DATA，第 3 行结束这个段，段中仅有的东西就是一个被定义成数据字节 (DB) 的 SUM，其中第 2 行的那个问号表示产生的目标代码需要为 SUM 保留一个存贮单元，但无需规定该单元中的初值，MY_DATA 将用作数据段。

第 4 行又引进了一个名为 MY_CODE 的新段，直到第 18 行都属于该段的范围。第 7 行到第 17 行是这个段中包含的指令，显然这一段将用作代码段。第 19 行标志源程序的结束，同时它还指出在执行该程序时，应该从标号为 GO (第 7 行)的那条指令开始运行。

ASSUME 语句(第 5 行)给汇编程序这样一些信息：当执行代码段时，汇编程序应该把假

定的那些信息放在寄存器 CS 和 DS 当中。前面已经讨论过了对 CS 作假设的必要性，至于对 DS 作设定的必要性的理解也还是比较容易的，由于代码段中的某些汇编语言指令要求直接存取数据(例如字节 SUM)，那末汇编程序就必须产生利用直接寻址方式 (见第二章中介绍的操作数寻址方式)访问 SUM 的机器语言指令，其目标指令必须确定：(1) SUM 的偏移量；(2) 某个段寄存器(一般情况下为 DS)，其中包含了 SUM 所在段(名为 MY_DATA) 的始址。汇编程序需要知道在执行指令时哪些段寄存器中将放着 MY_DATA 的起始地址。借助于这样一些信息，汇编程序就能够确定这样一些指令是否需要一个段跨越前缀，如果真需要这种前缀，汇编程序还要在此前缀中规定作为替换段基的段寄存器，比如在上例中，如果只有 ES 含有 MY_DATA 的始地址，那就得在某些指令前加上段跨越前缀，把 ES 用作 DS 的替换段基。进一步来说，倘若在执行指令的时候没有一个寄存器包含 MY_DATA 的起始地址，那么汇编程序就意识到它产生不了能够访问 SUM 的任何指令，因此在指令汇编过程中就会指出这个错误。

至此我们应该认识到，为了在执行指令时能够访问 SUM，必须设置某个段寄存器使之包含 MY_DATA 的起始地址，同时也应该意识到把 DS 作此用途是较为明智的，因为这时不再需要加上段跨越前缀了。现在所需要的就是要确保此假设得到了满足，实际上在访问 SUM 之前执行的几条指令(第 7、第 8 行)就起这样一个作用。

第 6 行规定PORT_VAL与常量 3 等效，这就使得在后面这些行中可把 PORT_VAL 用在出现 3 的任何位置上，其目的就是把 PORT_VAL 用作 3 号转接口的一个符号名，这样当要访问 3 号转接口时，只要引用 PORT_VAL 即可。假如我们决定重新编写该程序，使之利用 4 号转接口，那么只需要在第 6 行做一次改动，即改为

```
PORT_VAL    EQU    4
```

就可以了。

第 7 行到第 17 行的这段指令所做的工作就是：不断地从 3 号转接口读进数值，并将它们累加起来直到总和超过 100，然后将此和数从 3 号转接口输出，最后停机。整个程序的执行过程是这样的：第 7 行中的指令将 MY_DATA 段的起始地址的高 16 位送到寄存器 AX，第 8 行又把该值从 AX 送到 DS，这就使得后继指令可以访问 SUM 了。第 9 行的指令把 SUM 的初值设置为 0，观察一下第 7、8、9 三行可以看出目标操作数 (例如第 9 行中的 SUM) 总是写在源操作数(如第 9 行中的 0) 的前面，第 10 行将 SUM 中的值与 100 相比较，同时设置处理机的诸标志位以指示比较的结果；第 11 行对这些标志位进行测试，如果 SUM 不大于 100 就决定转移 (JNA)，跳转的目标是标号为 NOT_DONE 的那条指令(第 15 行)，如果第 11 行的指令转移不成功的话 (SUM 超过 100 了)，就把 SUM 送到 AL 当中(第 12 行)，然后又把 AL 的内容送到 3 号输出转接口(第 13 行)，接着处理机停止(第 14 行)。如果第 11 行的指令转移成功了的话 (SUM 不超过 100)，就将 3 号输入转接口的值送到 AL (第 15 行)，再将它加到 SUM 当中去(第 16 行)，第 17 行中的跳转指令又将控制转移到第 10 行。

现在让我们从上面的例子出发，导出 ASM-86 程序的一般结构。概括起来讲，程序的结构是一个或多个段块后接一个 END 语句，每个段块都以 SEGMENT 语句作为开始，且以 ENDS 语句作为结束 (End-of-segment)，在 SEGMENT 和 ENDS 语句之间是一系列的其他语句，每个语句常常都占据一行(如果需要后继行的话，它们就以 "&" 打头)。ASM-86 程序的结构形式如下所示：

```
名1        SEGMENT        (段"名1"开始)
             语句
             ⋮
             语句

名1        ENDS           (段"名1"的终点)
名2        SEGMENT        (段"名2"的始点)
             语句
             ⋮
             语句

名2        ENDS           (段"名2"的终点)
             ⋮
           END            (源程序结束)
```

这里给出的程序都与表格相似,很整齐。当然,这种表格化方法并不是程序结构的组成部分,它对于汇编程序毫无影响,但是这种程序格式是我们要大力提倡的,因为它使我们更容易阅读和理解程序。作为比较,请看下面的未经表格化处理的 IN_AND_OUT 程序文本,尽管它没有增加汇编程序的麻烦(事实上,汇编可能会更快),但我们理解起来就困难多了。

```
IN_AND_OUT SEGMENT ;start of segment          (段开始)
ASSUME CS: IN_AND_OUT ;that's what's in CS    (设定CS的内容)
CYCLE: IN AX, 5                               (把5号输入转接口的内容送到AX中)
INC AX                                        (把AX的内容加1)
OUT 2, AX                                     (把AX的内容从2号输出转接口输出)
JMP CYCLE                                     (跳转到CYCLE)
IN_AND_OUT ENDS ;end of segment               (段结束)
END CYCLE ;end of assembly                    (汇编结束)
```

§5.5 标　记

在讨论组成 ASM-86 程序的各种语句之前,我们先得熟悉一下语句的各个组成部分。语句是由标识符 (identifiers)、保留字 (reserved words)、界符 (delimiters)、常量 (constants) 和注释 (comments) 这样一些东西组成的,有时我们也把这样一些组成块称做标记 (tokens)。

一、标识符

标识符是由程序员规定的一些名字,在前面的样本程序中,有这样一些标识符如 SUM、CYCLE 和 PORT_VAL。标识符是不以数字开头的一个字母、数字、下划线(_)的序列,一个标识符允许的最大长度为 31 个字符,从实用角度可把它的长度看作是无限制的,因为除非程序员的故意刁难,人们是没有什么理由取一个长度超过 31 个字符的字符串作为标识符的。下面是几个标识符的例子:

```
X
GAMMA
```

JACK5
THIS_DONE
THISDONE

最后两个标识符是不相同的，因为前者中间有一个短下划线"_"。

二、保留字

保留字看上去很象标识符，但是它们在语言中具有特定的意义，因此不可以在程序中把它们用作一般的标识符。在前面的样本程序中，可以看到这样一些保留字，即 SEGMENT、MOV、EQU 和 AL 等等。根据规则，我们完全可以设置象 EQUAL 这样的一个名字，如

```
EQUAL    DB    ?
```

但写下面这个语句却是不妥当的

```
EQU      DB    ?
```

这是因为 EQU 是保留字。表 5.1 中列出了 ASM-86 的全部保留字。

三、界符

界符是一些非字母数字字符，它们在 8086 的汇编语言中具有特殊的意义。在我们的样本程序中，已经看到过象：和；这样一些界符，本章还要接触到其他一些界符。表 5.2 给出了 ASM-86 中的所有界符。

表 5.1 ASM-86 中的保留字

A. 指令助记符

AAA	CLD	ESC	JAE	JNA	JNP	LDS	MOV	POPF	RET	STC
AAD	CLI	HLT	JB	JNAE	JNS	LEA	MOVS	PUSH	ROL	STD
AAM	CMC	IDIV	JBE	JNB	JNZ	LES	MUL	PUSHF	ROR	STI
AAS	CMP	IMUL	JCXZ	JNBE	JO	LOCK	NEG	RCL	SAHF	STOS
ADC	CPMS	IN	JE	JNE	JP	LODS	NIL	RCR	SAL	SUB
ADD	CWD	INC	JG	JNG	JPE	LOOP	NOP	REP	SAR	TEST
AND	DAA	INT	JGE	JNGE	JPO	LOOPE	NOT	REPE	SBB	WAIT
CALL	DAS	INTO	JL	JNL	JS	LOOPNE	OR	REPNE	SCAS	XCHG
CBW	DEC	IRET	JLE	JNLE	JZ	LOOPNZ	OUT	REPNZ	SHL	XLAT
CLC	DIV	JA	JMP	JNO	LAHF	LOOPZ	POP	REPZ	SHR	XOR

B. 寄存器名称

AH	BL	CL	DI	ES
AL	BP	CS	DL	SI
AX	BX	CX	DS	SP
BH	CH	DH	DX	SS

C. 指示符(伪指令名)

ASSUME	END	EXTRN	NOSEGFIX	PUBLIC
CODEMACRO	ENDM	GROUP	ORG	PURGE
DB	ENDP	LABEL	PROC	RECORD
DD	ENDS	MODRM	RELB	SEGFIX
DW	EQU	NAME	RELW	SEGMENT

D. 其他保留字

ABS	EQ	INPAGE	MASK	NOTHING	PROCLEN	STACK
AT	FAR	LE	MEMORY	OFFSET	PTR	THIS
BYTE	GE	LENGTH	MOD	PAGE	SEG	TYPE
COMMON	GT	LOW	NE	PARA	SHORT	WIDTH
DUP	HIGH	LT	NEAR	PREFIX	SIZE	

四、常量

表 5.2 ASM-86 的界符表

'	;	>	$	*
,	:	>	=	/
(	.	<	-	?
)	&	>	+	

常量是在 ASM-86 的程序中出现的一些固定值。在样本程序中我们看到了象 0、3、100 这样一些常量，它们都是数字常量，实际上汇编程序也允许串常量的出现。

数字常量可以是 0 到 65535 (即 $2^{16}-1$) 中的任何整数，数字常量通常都是以 10 进制数表示的，当然也可以把它们写作 2 进制(以 B 结尾)、8 进制 (以 Q 结尾) 或 16 进制 (以 H 结尾)。对 16 进制常量而言，为了避免与标识符相混淆，规定其开头必须为 1 数字，这一问题只要在常量的头上加个 0 就足以解决了。15、1010B、27Q、3A0H，0BFA3H 这些表示都代表数字常量。

所谓串常量，就是指包在撇号内部的一个或两个字符所组成的序列(在某些特殊情况下，也允许多于两个字符的字符串出现)。撇号本身也可以包含在某个串常量当中，只要连续写两个撇号即可。'A'、'AB' 和 '''' 都是串常量，最后一个串常量中包含了一个撇号。串常量的值就是串中字符的 ASCII 码值。例如，'A' 的值相同于 41H (两者的值都为 10 进制的 65)，'AB' 的值相同于 4142H。可见串常量与数字常量可以交换使用。

五、注释

注释是跟在分号(;)之后直到行末的字符序列。它们对汇编程序来讲是没有什么意义的，但在程序中使用这样一些注释能够使我们较清楚地了解程序正在做的是些什么工作。尽管象这种注释：

```
INC     CX      ;increment      CX      (CX 的内容加 1)
```

仅仅传递了很少一点信息，好象作用不大，但是象下面这种注释：

```
INC     AX      ;prepare count for next iteration      (为下次迭代预先计数)
```

将使程序容易读得多了。

§5.6 表达式

在开始介绍语句之前，还要先引进一个新的组成块——表达式的概念。表达式是由前面介绍的一些标记组成的。

我们也可以把表达式看作是操作数和运算符组成的一个字符序列，在汇编某个程序时，可以将这些操作数和运算符组合起来从而得到一个“值”。下面先得介绍一下操作数和运算符，然后还得指出它们的组合方式以及表达式的值的计算手段。

一、操作数

操作数就是具有值的某种实体，操作数可能具有两种类型的值——一为数字值，一为存贮器的地址值。

所谓某个操作数具有数字值是指它要么是常量，要么是代表常量的一个标识符。在样本程序中，100 以及 PORT_VAL 都是具有数字值的操作数。这种类型的操作数的取值范围是从 −65,535 到 +65,535。

这里需要注意的是：操作数的值可以是负的，而常量决不可以取负值。当然，在常量之前可以加上一个减号，但不可以把它看作是常量的一个部分，它是一个数值运算符。

存贮器地址操作数通常是一些标识符，就象样本程序中的SUM和CYCLE一样。存贮器地址的值并不简简单单地是一个数，它是一组分量的集合，其中的每个分量一般都是一个数，分量之一是包含该存贮器地址的那个段的起始地址的高16位(段起始地址的低4位总是0)，另一个分量被称为存贮器地址的段内偏移量，这两个分量分别称为存贮器地址操作数的段(segment)和偏移量(offset)。

还有一种操作数就是括号内的表达式，这种操作数一般都是用在某些较大的表达式当中，例如3*(PORT_VAL+5)。

二、运算符

一个运算符可以对一个或多个操作数实施运算，从而得到一个新值。在ASM-86当中有5种类型的运算符，它们是：数值运算符(arithmetic operators)、逻辑运算符(logical operators)、关系运算符(relational operators)、解析运算符(analytic operators)和综合运算符(synthetic operators)。

数值运算符就是我们所熟悉的加(+)、减(-)、乘(*)、除(/)，在ASM-86中还加了一个数值运算符MOD，它得到的是除法的余数，也就是说19/7等于2，而19 mod 7等于5。

数值运算符总可以用于一对数值操作数，且其结果也将是一个数值，而将数值运算符用于存贮器操作数就要受到限制了，其应用规则可以归纳为：仅当其结果具有一个有意义的物理解释时，我们才说操作是有效的。例如，两个存贮器地址的乘积没有有意义的物理解释(它位于哪个段？它的偏移量为多少？)，因此这种操作是被禁止的；可是，位于同一个段中的两个存贮器地址的差却是有意义的，可以把它解释成两者之间的距离(偏移量之差)，其结果也是一个数字值；此外还有一种(也只有这一种了)包括一个存贮器地址的有意义的运算，就是加一个数值到某存贮器地址上去，或者从某个存贮器地址中减去一个数值，这样得到的结果也是一个存贮器地址，其段值保持不变，但其偏移量分别在原来的基础上加上或减去了一个数值。因此，在样本程序中，表达式SUM+2、CYCLE-5及NOT_DONE-GO都是有效的，但是应该注意，SUM-CYCLE不再是一个有效的表达式，因为它们位于不同的段当中。应该着重指出的是SUM+2是MY_DATA段中的一个存贮器地址(超过SUM两个字节)，而不是2加到SUM的内容所得到的一个数字值(因为存贮单元的内容只有在执行程序时才能得到，而表达式的赋值却是在程序的汇编过程中实现的)。

逻辑运算符就是通常的按位与(AND)、或(OR)、异或(XOR)、非(NOT)。逻辑操作数必须是数值类型的，即不允许把存贮器地址用作逻辑操作数，经运算得到的结果也为数值。例如：

```
1010101010101010 B AND 1100110011001100 B = 1000100010001000 B
1100110011001100 B OR  1111000011110000 B = 1111110011111100 B
NOT 1111111111111111 B = 0000000000000000 B
1111000011110000 B XOR SUM 是一无效操作。
```

作为逻辑运算的一个例子，请看下面这两个语句：

```
IN    AL, PORT_VAL
```

OUT PORT_VAL AND 0FEH, AL

IN 指令从转接口 PORT_VAL 取进输入数据，而 OUT 指令则执行把数据输出到转接口 PORT_VAL AND 0FEH 这样一个逻辑操作，如果转接口号码 PORT_VAL 为偶数，那么它就是输出到 PORT_VAL 转接口，但当 PORT_VAL 为奇数时，它实际上是输出到比 PORT_VAL 低一的那个转接口。OUT 指令的转接口实际值，是在指令的汇编期间确定的，而不是在该指令的执行过程中赋予的。

由前面的讨论可以看出，AND、OR、XOR 和 NOT 既是指令助记符，又是 ASM-86 的运算符。但当作为运算符使用时，其值的计算是在程序的汇编期间完成的；而在当作指令助记符使用时，其作用是在程序执行过程中实现的。例如：

```
AND    DX, PORT_VAL AND 0FEH
```

它将使得汇编程序去计算 PORT_VAL AND 0FEH 的值，然后产生一条“与”(AND)立即数指令，在该指令的操作数字段中包括了刚刚计算出的 PORT_VAL AND 0FEH 的值。在以后执行此段程序时，该指令就把上面得到的那个值和 DX 中的内容相“与”，再把结果存入 DX 寄存器当中。

关系运算符有这样几个：相等 (EQ)、不等 (NE)、小于 (LT)、大于 (GT)、小于等于 (LE)、大于等于 (GE)。PORT_VAL LT 5 就是关系运算符应用的一个例子。关系运算中的两个操作数要么都是数值，要么是位于同一段中的两个存贮器地址；但得到的结果总是数值，若关系不成立结果就为 0，若关系成立(为真)结果就是 0FFFFH(16 位全 1)。

下面是一个使用了关系运算符的例子：

```
MOV    BX, PORT_VAL LT 5
```

如果 PORT_VAL 的值小于 5 的话，汇编程序就将该指令汇编成为：

```
MOV    BX, 0FFFFH
```

否则，即当 PORT_VAL 大于等于 5 时，汇编程序就将该指令汇编成：

```
MOV    BX, 0
```

由于我们也许并不希望指令中的某个字段只能取 0 值或 0FFFFH 为值，而没有任何其他选择，所以初看起来关系运算符似乎没有多大用处，但是通过将关系运算符与逻辑运算符结合起来使用，就可以得到所需要的任何数值了。例如：

```
MOV    BX, ((PORT_VAL LT 5) AND 20) OR ((PORT_VAL GE 5)
                AND 30)
```

在 PORT_VAL 小于 5 时，就把它汇编成：

```
MOV    BX, 20
```

否则汇编成

```
MOV    BX, 30
```

值得注意的是，这里使用了大量的括号以使运算能够按序进行。倘若你总是使用括号来明确规定运算次序，那你就无需再去死记那一套无聊的算符优先级别了。

解析运算符的作用，就是把存贮器地址操作数分解成为组成它的各个分量。而综合运算符的作用正好与此相反，它是将这些存贮器地址的各组分量组合起来，从而形成存贮器地址。有关这两种运算符的讨论将在存贮器地址操作数之后再度进行。

§5.7 语　句

在一段 ASM-86 程序中有两种语句能够出现，它们就是所谓的指令性语句 (MOV, ADD, JMP 等等)和指示性语句(即伪指令，如 DB, SEGMENT, EQU 等等)。对于每条指令性语句，汇编程序都要为之产生目标代码，而指示性语句仅仅告诉汇编程序对于后面的指令性语句应该产生哪种代码。例如，下面这个指示性语句

```
MY_PLACE    DB    ?
```

告诉汇编程序 MY_PLACE 被定义成一个字节，汇编程序为单元 MY_PLACE 分配一个存贮器字节，当汇编程序在此之后碰到指令性语句

```
INC    MY_PLACE
```

时，它就产生一条把 MY_PLACE 单元的内容加 1 的目标代码，由于前面已经遇到过那条指示性语句，所以汇编程序就知道在 INC 指令的 W 字段中应该放上一个 "0" (表示它是一个字节加 1 指令)。

两种类型的语句格式很相似。指令性语句的格式为：

标号：指令助记符　变元,…,变元　;注释

而指示性语句的格式为

名字　指示符　　变元,…,变元　;注释

由此可见，指令性语句中的标号后面有一个冒号，而指示性语句的名字后面就没有这个冒号，这是两种语句的突出区别。标号使得某个符号名与一个指令的位置相联系，在跳转或调用子程序指令中，可以把标号用作一个操作数。指令性语句中的标号是可有可无的，但指示性语句中的名字，却要根据指示性语句的形式确定，它可以是非出现不可的、可选择的或是不允许出现的。

指令性语句中的指令助记符和指示性语句中的指示符，规定了语句的功用。指令助记符对应于 8086 上的近 100 个操作码的集合，指示符则对应于由 ASM-86 汇编程序所提供的大约 20 种功能(见表 5.1)。特定的指令助记符或指示符，可能还需要一些附加信息以完整地确定其功能，这样一些信息是通过一组变元 (argument) 提供的。

语句中的注释使得程序更容易读，它们总是可以选择的，但当注释确实存在时，为了便于识别，必须在这些注释之前加上分号(;)。

§5.8　指示性语句(伪指令)

ASM-86 中有多种指示性语句，它们是符号定义语句 (symbol-definition statements)、数据定义语句 (data-definition statements)、段定义语句 (segmentation-definition statements)、过程定义语句 (procedure-definition statements) 和终止语句 (termination statements)。本节将对这些语句进行讨论。

一、符号定义语句

EQU 语句为定义的符号名提供了一种代表某个值或另一个符号名的手段。EQU 语句的

两种形式如下所示：

```
名字        EQU     表达式
新名字      EQU     老名字
```

举例如下：

```
BOILING_POINT     EQU     212
BUFFER_SIZE       EQU      32
NEW_PORT          EQU     PORT_VAL+1
COUNT             EQU     CX
```

最后一个例子与另外三个例子不同，因为在这个例子中，COUNT 并不代表一个值，而是与 CX 具有相同的意义。

利用 PURGE 语句，能够解除对某个符号名的定义，以使得在后面可把该符号名用来表示某个完全不同的东西。例如：

```
PURGE     BUFFER_SIZE
```

二、数据定义语句

一个数据定义语句为某个数据项分配存贮器，并且把一个符号名与该存贮器地址联系起来，还可以(任选)给此数据项赋一个初始值。与数据项相联系的那些符号名被叫做变量。下面是几个数据定义语句的例子：

```
THING             DB    ?     ;定义一个字节
BIGGER_THING      DW    ?     ;定义一个字 (2 个字节)
BIGGEST_THING     DD    ?     ;定义一个双字 (4 个字节)
```

在上面的例子当中，THING 是一个与存贮器中的一个字节相关联的符号名，BIGGER_THING 与存贮器中的两个连续字节相关联，BIGGEST_THING 与存贮器中的 4 个连续字节相关联。

在对问号(?)进行讨论之前，我们需要引进数据项的初值的概念。汇编程序所产生的目标代码就是由 0、1 所组成的每条指令，以及存放这些指令的存贮器地址，在目标码产生之后，就把指令送到存贮器的指定地址中，然后执行。在装入指令的过程中，也有可能需要把数据项的初值同时装入存贮器，这些初值就是在数据定义语句中被规定的。下面这个语句使汇编程序在装入其目标代码的同时，在单元 THING 中放上一个初值 25：

```
THING     DB     25
```

处于初值那个位置上的问号意味着我们没有为对应的数据项规定初值，也就是说在对应的存贮单元中无论出现什么初值都将满足要求。当汇编程序看到这个问号时，它仍然为那个数据项分配存贮器，但此时不再需要为它产生目标代码来初始化该存贮单元了(尽管有时初始化一下也许很好)。

在一般情况下，初值能用一个表达式来确定，这是因为在程序汇编的时候，就对表达式赋值了。因此，可以写这样一些语句：

```
IN_PORT       DB     PORT_VAL
OUT_PORT      DB     PORT_VAL+1
```

前面已经说过，表达式可能有两种类型的变量——数字和存贮器地址，给某个字节、字或

双字一个数字值作为初值当然总是有意义的，但给它一个存贮器地址作为初值情形又会怎样呢？很显然肯定不可能将一存贮器地址装入一个字节当中，所以不考虑这种情况；但是，偏移量正好能够装入一个字；偏移量和段分量也可以装入一个双字当中。因而我们也能写出这样一些语句：

```
LITTLE_CYCLE    DW    CYCLE
BIG_CYCLE       DD    CYCLE
                :
CYCLE:          MOV   BX, AX
```

上面对 LITTLE_CYCLE 的这种初始化将允许段内间接跳转或调用指令的执行，为了把控制转移到标号为 CYCLE 的那条指令，只要使用一条目标名为 LITTLE_CYCLE 的间接段内跳转或调用指令即可；类似地，一条段间跳转或调用指令通过使用名为 BIG_CYCLE 的数据项就能把控制转移到 CYCLE。

至此我们已经使用了数据定义语句来每次定义单个字节(字或双字)，实际上我们还会常常与表格或许多字节 (字或双字) 打交道。举个例子，8086 的 XLAT 指令利用一张由字节组成的表格，将某个已经编码的值，翻译成在不同的编码系统下的同一个值；此外，8086 的中断机构，利用一张起始地址为 0 的双字表格，来指向各中断服务子程序的开始地址；8086 的串指令，也是对包含这些串元素的字节或字表格实施其操作功能的。

将几个初值放在一个数据定义语句中就定义了一个表格。下面这个语句就定义了一张由 2 的幂组成的字节表：

```
POWER_2    DB    1, 2, 4, 8, 16
```

对应于 POWER_2 的那个字节中的初始值为 1 (其初始化过程是在把目标代码装入存贮器时实现的)，接下来的 4 个字节将分别赋以初值 2、4、8 和 16。若要将一个字节表中的所有字节都初始化为 0，则可以这样定义：

```
ALL_ZERO    DB    0, 0, 0, 0, 0, 0
```

或简写成

```
ALL_ZERO    DB    6 DUP (0)
```

一张不赋初值的表格可用下面两个等价语句中的任一语句来定义：

```
DONT_CARE    DB    ?, ?, ?, ?, ?, ?, ?, ?
DONT_CARE    DB    8 DUP (?)
```

三、存贮单元的类型

ASM-86 为程序中引用的每个存贮单元，均规定一个类型，汇编程序一直知道各存贮单元的类型，因而能够在它碰到一条访问存贮器的指令时，生成正确的代码。例如数据定义语句

```
SUM    DB    ?
```

就告诉汇编程序存贮单元 SUM 的类型为 BYTE (字节)，在后面碰到象

```
INC    SUM
```

这种指令时，汇编程序就知道此时应该产生一个字节加 1 指令，而不是产生字加 1 指令。

一个存贮单元的类型，可以是下列类型中的任意一种：

1. 数据字节 (BYTE)，例如

SUM　　DB　?　(定义一个字节)

2. 数据字　(WORD，两个连续字节)，如
   BIGGER_SUM　DW　?　(定义一个字)
3. 数据双节 (DWORD，4个连续字节)，例
   BIGGEST_SUM　DD　?　(定义一个双字)
4. NEAR (靠近)指令位置，例如下面这个语句中的CYCLE：
   CYCLE：　　CMP　SUM，100
5. FAR (远离)指令位置。

指令位置能够出现在跳转或调用指令语句当中，如果指令的位置类型为NEAR (靠近)型，那末汇编程序就产生一条段内跳转或调用指令；倘若该单元的类型为FAR (远离)型的，汇编程序就产生一条段间跳转或调用指令。例如带标号的指令性语句

CYCLE：　CMP　SUM，100

它通知汇编程序存贮单元CYCLE的类型为NEAR型。此后，当汇编程序遇到这样的一条指令

JMP　CYCLE

时，汇编程序就能够知道产生一个段内跳转指令，而不是一个段间跳转指令。

一个存贮器地址加上或减去某个数字值，所得到的存贮器地址与原存贮器地址具有相同的类型，例如 SUM+2 是另一个字节 (BYTE)，BIGGER_SUM-3 是一个字 (WORD)，CYCLE+1 等均是NEAR型的指令单元。

四、解析和综合运算符

有关存贮器地址方面的内容我们已经知道得够多了，现在该对其运算符进行讨论了。解析运算符被用来将存贮器地址操作数分解成诸分量，起这种作用的运算符有SEG、OFFSET、TYPE、SIZE和LENGTH。

SEG运算符返回的是存贮器地址操作数的段分量，OFFSET运算符返回的是偏移量，这两种分量一般都是数值。

TYPE运算符也返回一个数字值，只是返回值代表的是存贮器地址操作数的类型分量，各种存贮器地址操作数的类型分量的值如下表所示：

存贮器地址操作数	类型分量
数据字节	1
数据字	2
数据双字	4
NEAR 指令单元	−1
FAR 指令单元	−2

注意字节、字、双字的类型分量分别对应于各自所占有的字节数目，但最后两种指令单元的类型分量的值没有物理意义。

运算符 LENGTH 和 SIZE 只可用于数据存贮器地址操作数 (BYTE，WORD 或 DWORD)。运算符 LENGTH 返回一个数字值，该数值为与存贮器地址操作数相关的单元(字节、字或双字)数目；运算符SIZE也返回一个数值，它等于分配给指定的存贮器地址操作数

的字节总数。例如，如果 MULTI_WORDS 是由语句

```
MULTI_WORDS    DW    50 DUP (0)
```

定义的，那末 LENGTH MULTI_WORDS 等于 50，而 SIZE MULTI_WORDS 等于 100。实际上，稍微思考一下就能导出下列关系：

```
SIZE X = (LENGTH X) * (TYPE X)
```

综合运算符则被用来从各分量出发构造存贮器地址操作数，这种类型的运算符是 PTR 和 THIS。

运算符 PTR，产生一个与另一存贮器地址操作数段和偏移量相同但类型不同的存贮器地址操作数，它与数据定义语句不同，PTR 运算符不分配任何存贮器，它只是给已分配了的存贮器单元一个新的意义。例如，倘若我们这样定义 TWO_BYTE：

```
TWO_BYTE    DW    ?
```

然后我们再给这个字的头一字节一个新的名字

```
ONE_BYTE    EQU    BYTE PTR TWO_BYTE
```

在这个例子当中，运算符 PTR 产生一个新的存贮器地址操作数，它具有与 TWO_BYTE 相同的段分量和偏移量，但其类型分量为 BYTE，当然也可以这样来命名 TWO_BYTE 的第二个字节

```
OTHER_BYTE    EQU    BYTE PTR (TWO_BYTE+1)
```

或者更简单地利用语句

```
OTHER_BYTE    EQU    ONE_BYTE+1
```

还可以把运算符 PTR 用来产生字或双字，例如

```
MANY_BYTES        DB    100 DUP (?)    ;an array of 100 bytes (100个
                                          字节的数组)
FIRST_WORD        EQU   WORD PTR MANY_BYTES
SECOND_DOUBLE     EQU   DWORD PTR (MANY_BYTES+4)
```

此外，还可以把运算符 PTR 用来产生指令的位置，例如：

```
INCHES:    CMP    SUM, 100              ;type of INCHES is NEAR
           :
           JMP    INCHES                ;intrasegment jump(段内跳转)
           :
MILES      EQU    FAR PTR INCHES        ;type of MILES is FAR
           JMP    MILES                 ;intersegment jump(段间跳转)
```

通过上面这些讨论可以看出，由原来的存贮器地址操作数形成新的存贮器地址操作数的方法有下面几种：(1) 使用 PTR 运算符，就象在 BYTE PTR TWO_BYTE 中那样；(2) 利用表达式，如在 ONE_BYTE+1 中那样；(3) 利用 PTR 和表达式，就象在 BYTE PTR (TWO_BYTE+1) 中一样。当希望改变偏移量而保持原来的类型分量不变时，表达式就起作用了。与此相反，即当我们希望保持偏移量不变而改变类型分量时，就该使用 PTR 运算符。表达式和 PTR 都不会改变段分量，由它们所产生的新的存贮器地址操作数的长度分量总为 1 (只要它不是一个指令单元)。

与 PTR 类似，综合运算符 THIS 也产生一个规定类型的存贮器地址操作数，也不给它分

配任何存贮器。这样一个新的存贮器地址操作数的段分量和偏移量就是下一个可分配单元的段分量和偏移量，例如：

```
        ⋮
MY_BYTE    EQU    THIS    BYTE
MY_WORD    DW     ?
        ⋮
```

就将产生一个类型分量为 BYTE 的存贮器地址操作数 MY_BYTE，其段分量与偏移量跟 MY_WORD 的对应分量相同，在这个例子当中，我们也可以用运算符 PTR 来构造 MY_BYTE，如下所示：

```
MY_BYTE    EQU    BYTE PTR MY_WORD
```

利用 THIS 运算符能够非常方便地定义 FAR 指令单元，看下面这个例子：

```
MILES    EQU    THIS FAR
         CMP    SUM, 100
           ⋮
         JMP    MILES
```

值得注意的是，上例中在使用了运算符 THIS 之后，就不再需要一个与 MILES 具有相同段分量和偏移量的 NEAR 指令单元了，假如我们希望利用 PTR 算符来代替 THIS，那么这样一个 NEAR 指令单元是完全必要的。

五、段定义语句

段定义语句为我们提供了构造程序的手段，它使程序能够使用 8086 的存贮器段。这些语句就是 SEGMENT，ENDS，ASSUME 和 ORG。

SEGMENT 和 ENDS 语句把汇编语言源程序分成一些段，这样的一些段与待装入目的码的那些存贮器段相对应。汇编程序之所以关心程序的分段，是因为考虑到下面两种情况：

1. 段内跳转和调用指令仅仅包含新位置的偏移量 (16 位)；而对段间跳转和调用指令而言，除了需要知道偏移量之外，它还必须包含段分量(又是 16 位)；
2. 使用当前数据段和栈段的数据存取指令时 (8086 结构的最优存取方法)，只需包含数据单元的偏移量 (16 位)；而对于其他类型的数据存取指令(访问当前 4 个可寻址段中的某个单元)，还要包含一个段跨越前缀(另加 8 位)。所谓“当前”是指指令执行期间，而不是指汇编期间。

因此，为了汇编出正确的目标代码，汇编程序所必须知道的不仅仅是程序的段结构，还要知道在执行各种指令时，哪些段是可寻址的(由段寄存器指向)，这方面的信息是由 ASSUME 语句提供的。

下面这个例子，说明了如何把语句 SEGMENT、ENDS、ASSUME 用来定义一个代码段、数据段、附加段和堆栈段：

```
MY_DATA    SEGMENT
X          DB      ?
Y          DW      ?
Z          DD      ?
```

```
MY_DATA     ENDS
MY_EXTRA    SEGMENT
ALPHA       DB          ?
BETA        DW          ?
GAMMA       DD          ?
MY_EXTRA    ENDS
MY_STACK    SEGMENT
            DW          100 DUP (?)
TOP         EQU         THIS WORD
MY_STACK    ENDS
MY_CODE     SEGMENT
            ASSUME      CS: MY_CODE, DS: MY_DATA
            ASSUME      ES: MY_EXTRA, SS: MY_STACK
START:      MOV         AX, MY_DATA     ;initializes DS
            MOV         DS, AX          (初始化 DS)
            MOV         AX, MY_EXTRA    ;initializes ES
            MOV         ES, AX          (初始化 ES)
            MOV         AX, MY_STACK    ;initializes SS
            MOV         SS, AX          (初始化 SS)
            MOV         SP, OFFSET TOP  ;initializes SP
                                        (初始化SP)
              ⋮
MY_CODE     ENDS
            END         START
```

MY_CODE 段头上的码字将在执行程序时，初始化各个段寄存器，以让它们指向适当的段，同时也初始化了栈指针以让其指向栈段的末尾，ASSUME 语句使得汇编程序在执行程序码字的时候，知道将存放在段寄存器当中的值。

为了说明 ASSUME 语句的作用，先来分析一下把字节 X 中的内容送到 ALPHA 中这个码字(在段 MY_CODE 里面)，为了完成该动作，需要一条把 X 的内容送到某个寄存器(例如 BX) 的指令，还要一条把此寄存器的内容送到 ALPHA 的指令，所以很自然地使用了下面两个语句：

```
MOV     BX, X           ;from X to BX ((X)→BX)
MOV     ALPHA, BX       ;from BX to ALPHA ((BX)→ALPHA)
```

执行情况究竟怎样呢？在这种 MOV 指令的执行过程中，8086 处理机通常总是考察 DS 寄存器，以求得存放某数据项 (X 或 ALPHA) 的段的始址。因为 DS 包含了段 MY_DATA 的始址，X 位于该段当中，所以当访问 X 时，指令的执行确实是正确的；但若要访问 ALPHA (第二条指令)，指令的执行就不正确了，这是由于 ALPHA 位于段 MY_EXTRA 当中，而该段的始址不在 DS 中的缘故。ASSUME 语句使得汇编程序知道第一条指令将正确执行，同时汇编程序也知道了尽管 MY_EXTRA 的始地址不在 DS 中，但它确实在某个别的段寄存器 当 中

(这里是ES)，这也要归功于我们的ASSUME语句．这样，在碰到第二条指令时，汇编程序就将产生一个段跨越前缀，使其亦能正确地得到执行．

当然，我们也并不是总能知道在执行到某一点时的各个段寄存器中的内容，看下面这个例子：

```
OLD_DATA    SEGMENT
OLD_BYTE    DB          ?
OLD_DATA    ENDS
NEW_DATA    SEGMENT
NEW_BYTE    DB          ?
NEW_DATA    ENDS
MORE_CODE   SEGMENT
            ASSUME      CS: MORE_CODE
            MOV         AX, OLD_DATA    ;put OLD_DATA into
            MOV         DS, AX          ;…DS and
            MOV         ES, AX          ;…ES
                                (把OLD_DATA放到DS和ES当中)
            ASSUME      DS: OLD_DATA, ES: OLD_DATA
              ⋮
CYCLE:      INC         OLD_BYTE
                                (这里DS的内容是什么呢?)
              ⋮
            MOV         AX, NEW_DATA    ;put NEW_DATA
            MOV         DS, AX          ;…into DS
                                (把NEW_DATA放到DS中去)
            JMP         CYCLE
              ⋮
MORE_CODE   ENDS
```

在第一次执行INC指令时，DS将含有OLD_DATA，故对DS的假设是正确的；然而后面将把DS变成NEW_DATA且接下来还要执行INC指令，因此在执行INC指令时，汇编程序对DS的任何假定都是错误的，这就要求汇编程序为INC指令产生一个段跨越前缀(定义附加段)，尽管第一次执行该INC指令时，此前缀是不必要的，但也还是加上．为了通知汇编程序不要对DS作任何假设，我们必须把下面这个设定正好放在INC指令之前：

```
              ⋮
            ASSUME      DS: NOTHING
CYCLE:      INC         OLD_BYTE
              ⋮
```

在包含码字的任何段的开头或前面，我们都必须告诉汇编程序(通过一个ASSUME语句)，在执行该码段的时候，汇编程序所设定的信息将存放在CS寄存器当中．

与此不同，我们并不一定需要使用ASSUME语句来告诉汇编程序DS、ES和SS中放着什么东西，这是因为我们可以告诉汇编程序执行某条指令应该使用哪个段寄存器．还是前面

那个例子，把 X 送到 ALPHA，可用下面两条指令：

```
MOV    BX, DS:X
MOV    ES: ALPHA,BX
```

这就是说，存取 X 时，应使用 DS；存取 ALPHA 时，应使用 ES。由于在执行这些指令时，处理机通常总是使用 DS，所以汇编程序在为第二条指令生成目标代码时，应该产生一个段跨越前缀，而对第一条指令来讲不产生此类前缀。

现在来分析一下存贮器分段的一个缺点，同时看看我们是如何克服它的。存贮器段的段长最短为 16 个字节，最长为 2^{16} 个字节（记住：段起始地址的低 4 位总为 0）。倘若某个段并没有使用完它所有的存贮空间，那么，其他的段，就从上一段之后开始分配，但是这个段也要求以 16 个字节为边界开始分配，因而它也许不能紧接着上一段的最后一个已占用的字节之后开始，也就是说在段与段之间最多可能浪费 15 个字节。

作为例子，假设第一段的始址为 10000 H，它只占用 6DH 个字节，可见它占用的最后一个字节的地址为 1006CH，故第二段的始址最早也只能是 10070 H，这样就浪费掉 3 个字节（1006DH、1006EH 和 1006FH）。

假如，我们不是从由第一段使用过的最后一个字节之后的最低 16 字节界地址开始第二段，而是在上一段的最高 16 字节界地址处开始第二段，那就不会再浪费掉任何字节了。在前面这个例子当中，我们能够在地址 10060 H 处开始第二段，这就导致了这样一个结果：由第一段使用的最后几个字节（精确地讲，本例中是 13 个字节）同时又在第二段中，只要规定第二段不要使用这几个字节，那就公平合理，皆大欢喜！因此，如果第二段始址为 10060 H，那它就不使用位于第二段中而偏移量不到 000DH 的那些字节，这样对第一段没有影响，又没有浪费存贮空间。

一般说来，我们并不关心我们的一些段在存贮器中的实际位置，这要有翻译程序去做抉择。但是我们有时也许要给翻译程序一些限制，例如："不要把这个段与其他任何段重迭"，"保证该段使用的第一个字节的地址为偶数（以便能在一个存贮周期内存取字）"或"在下面这个地址开始该段"。我们可以把这样一些限制写成源程序如下：

1. 不要重迭。段中的第一个可用字节位于某个 16 字节的边界上，且偏移量为 0000。

```
MY_SEG    SEGMENT     ;this is the normal case      (一般情况就是如此)
                      :
MY_SEG    ENDS
```

2. 若需要就重迭，但第一个可用字节的地址为偶数。

```
MY_SEG    SEGMENT     WORD; word aligned      (以字为边界)
                      :
MY_SEG    ENDS
```

3. 若需要就重迭，第一个可用字节的位置任意规定。

```
MY_SEG    SEGMENT     BYTE    ;byte aligned     (字节定位)
```

4. 在指定的 16 字节边界处开始段，第一个可用的字节，也由特定的偏移量规定。

```
MY_SEG    SEGMENT     AT 1A2BH     ;address 1A2B0H
          ORG         0003 H       (地址为 1A2B3H)
          :
```

```
MY_SEG    ENDS
```

最后一个例子中出现了一个新语句即ORG，它规定在该段中待用的下一个字节的偏移量。

六、过程定义语句

过程是在程序中的任何地方都可以调用和执行的一段代码，每调用一次过程，就执行一遍组成该过程的那些指令，然后返回到调用点。

调用过程的8086指令是CALL，从过程返回的指令为RET。它们都有段内和段间的指令形式，用于段间的调用和返回指令，要压入(CALL)或弹出(RET)返回地址的段分量和偏移量；而段内调用和返回指令，只要压入和弹出偏移量。

利用段内CALL调用的过程，必须用段内RET返回，这种过程被称为NEAR过程；与此类似，利用段间CALL调用的过程，也必须用段间RET返回，并且把这些过程称为FAR过程。

过程定义语句PROC和ENDP，定义了一个过程的界，同时也指出了该过程是一个NEAR过程还是一个FAR过程，这就从两个方面支持了汇编程序：

1. 当汇编到调用哪个过程的CALL指令时，汇编程序就知道将其汇编成哪种形式的CALL。

2. 当汇编到哪个过程的RET指令时，汇编程序也可以知道将其汇编成哪种RET。

下面这段程序说明了这一点：

```
MY_CODE    SEGMENT
UP_COUNT   PROC        NEAR
           ADD         CX, 1
           RET
UP_COUNT   ENDP
START:     ⋮
           CALL        UP_COUNT
           ⋮
           CALL        UP_COUNT
           ⋮
           HLT
MY_CODE    ENDS
           END         START
```

由于把UP_COUNT说明成一个NEAR过程，故调用它的所有CALL都汇编成段内调用；所有返回语句也都汇编成段内返回。

从上面这个例子，也可以看出RET指令与HLT指令具有不少相似之处，一个过程当中可能有多个RET，就象一个程序中可能有多个HLT出现那样。有时也不一定需要过程（程序）的最后一个语句为RET(HLT)，但如果最后一句不是这种语句，那么最后这个语句就必须是一个跳回到该过程(或程序)内部的跳转语句。ENDP(或END)语句告诉汇编程序过程(或程序)在哪里结束，但它不会使汇编程序产生一个RET(或HLT)指令。

七、结束语句

除了一种例外情况之外，每个结束语句，都与某个开始语句配对，例如SEGMENT与ENDS、PROC和ENDP。这些结束语句的解释是与相应的开始语句一道进行的。

唯一的例外情况就是END，它是源程序的结束标志，END通知汇编程序不要再汇编其他指令了。END语句的形式是：

```
END    表达式
```

这里的表达式，必须产生一个存贮器地址值，该地址就是当要执行该程序时待执行的第一条指令的地址。

下面这个例子说明了END语句的使用情况：

```
        ⋮
START:
        ⋮
        END    START
```

§5.9 指令性语句

指令性语句中的绝大部分对应于8086处理机的指令，每条指令性语句都将引起汇编程序去产生一个8086指令。8086的指令由一个操作码字段以及确定操作数寻址方式的一些字段组成，因此，ASM-86中的指令性语句必须包含一个指令助记符，以及足够的寻址信息，从而使得汇编程序能够产生这条指令的目标代码。

一、指令助记符

大多数指令助记符，就是第三章中为8086指令所引进的那些操作码的符号名。为使汇编语言更为通用，又加进了两个指令助记符NIL和NOP。

指令助记符NOP（空操作），引起汇编程序产生一条单字节指令，这条指令所要做的动作，就是把AX的内容送到AX（操作码为90H）。可见这条指令不起什么作用，又因为它不需要访问存贮器，所以它花的时间也不多。尽管初看起来设置这条指令有些不可思议，但有时它确实有用。例如在程序编好之后的某个时候，甚至在执行程序时，我们可能需要加进一些指令，这时就可把NOP用作待填入指令的一些保留空间。又例如，当精确的时间关系变得重要时，还可以把NOP用来"减慢"程序的某个部分的执行速度。

NIL仅仅是个指令助记符，汇编程序碰到它时不产生任何指令，这一点是与NOP不同的。在汇编语言的程序中可把NIL用作具有标号的一个空语句，这一点在下面的例子中可以得到说明：

```
CYCLE:  NIL
        INC    AX
```

尽管它等价于

```
CYCLE:  INC    AX
```

但如果我们需要在今后某个时候要在INC指令之前插进一条指令的话，那么使用这个NIL就使插入工作容易多了。

二、指令前缀

8086的指令集还允许在指令之前，加上一个或多个前缀字节，在我们的系统中，总共有三

种前缀——段跨越、重复及锁。

ASM-86 允许指令助记符与下列前缀一道出现：

REP　　(重复)
REPE　　(相等时重复)
REPNE　　(不等时重复)
REPZ　　(为 0 时重复)
REPNZ　　(非 0 时重复)
LOCK

下面这个例子就是使用一个前缀的一条指令性语句：

```
CYCLE:    LOCK    DEC    COUNT
```

段跨越前缀的用法有些特殊，它是在汇编程序认识到某个存贮器存取需要这样一个前缀时，由汇编程序自动产生的。这个过程可以分成两步：首先汇编程序要选择一个段寄存器，利用这个段寄存器将使得指令能够正确地得到执行，这一步的基础就是从前面的 ASSUME 语句中获得的一些信息。当然，我们也能够迫使汇编程序去选择某个特定的寄存器，这只要在指令当中包括寄存器，例如：

```
MOV    BX, ES: SUM
```

汇编程序要做的第二步，就是根据 8086 处理机的需要，为了确保指令正确执行选择段寄存器，以此确定是否需要一个段跨越前缀。

三、操作数寻址方式

8086 处理机提供了多种操作数寻址方式，因此在编写指令性语句时，ASM-86 也必须提供各种寻址方式的表示手段，这里我们准备用例子来说明：

1. 立即数寻址：

```
MOV    AX, 15    (15 是一个立即数)
```

2. 寄存器寻址：

```
MOV    AX, 15    (AX 是一个寄存器操作数)
```

3. 直接寻址：

```
SUM    DB    ?
       :
       MOV    SUM, 15    (SUM 是一个直接寻址的存贮器操作数)
```

4. 通过基址寄存器间接寻址：

```
MOV    AX, [BX]
MOV    AX, [BP]
```

5. 通过变址寄存器间接寻址：

```
MOV    AX, [SI]
MOV    AX, [DI]
```

6. 通过基址寄存器加变址寄存器的间接寻址：

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

```
MOV     AX, [BP][SI]
MOV     AX, [BP][DI]
```

7. 通过基址或变址寄存器加偏移量间接寻址：

```
MANY_BYTES    DB    100 DUP(?)
                    ⋮
                    MOV  AX, MANY_BYTES[BX]
                    MOV  AX, MANY_BYTES[BP]
                    MOV  AX, MANY_BYTES[SI]
                    MOV  AX, MANY_BYTES[DI]
```

8. 通过基址寄存器加变址寄存器加偏移量间接寻址：

```
MANY_BYTES    DB      100 DUP (?)
                    ⋮
                    MOV     AX, MANY_BYTES[BX][SI]
                    MOV     AX, MANY_BYTES[BX][DI]
                    MOV     AX, MANY_BYTES[BP][SI]
                    MOV     AX, MANY_BYTES[SP][DI]
```

前面已经说过，当汇编程序产生一条访问存贮器单元的指令时，它要用到有关存贮器单元的类型方面的信息。例如，在下面这段程序中，汇编程序就知道该产生一条字节加 1 指令：

```
SUM     DB      ?       (字节型)
          ⋮
        INC     SUM             (字节加 1)
```

然而，对于间接的操作数寻址方式而言，汇编程序并不能全都知道存贮器单元的类型，例如

```
MOV     AL, [BX]
```

尽管汇编程序不知道上面这条指令中的源操作数的类型，但是它到是确实知道目标操作数 AL 的类型(是字节)，因此汇编程序就假定[BX]的类型也为 BYTE (字节)，从而产生一条字节传送指令。但对于下面这个语句：

```
INC     [BX]
```

这里不存在第二个存贮器单元来帮助汇编程序确定[BX]的类型，所以汇编程序也就无法决定是该产生一个字节加 1 指令，还是该产生一个字加 1 指令，因此为了使汇编程序能够确定操作的类型，必须把上面这个语句写成：

```
INC     BYTE PTR [BX]     (字节加 1)
```

或

```
INC     WORD PTR [BX]     (字加 1)
```

四、串指令

汇编程序通常能由对操作数的说明来确定其操作数的类型，从而知道为了访问该操作数应产生哪种码字。当然我们刚才也已看到在使用间接寻址方式时，汇编程序也许要求我们给它一些附加信息，以便它能够确定操作数的类型。

串指令是需要这种附加信息的又一个例子。现在来分析一下串指令 MOVS，这条指令把

偏移量在SI中的存贮器里的内容送到偏移量在DI中的存贮器里去。由于在这条指令当中，我们无权指定传送哪些项，也无权选择源地址或目标地址，因此照理说不应该要我们确定任何操作数；但是，因为这条指令既能传送一个字节也能传送一个字，这就要求汇编程序清楚当时所要传送的是什么，以产生正确的指令。为此，用于表达指令MOVS的ASM-86语句必须指定送到SI和DI的是哪些项。

例如下面这段程序：

```
ALPHA   DB      ?
BETA    DB      ?
        :
        MOV     SI, OFFSET ALPHA
        MOV     DI, OFFSET BETA
        MOVS    BETA, ALPHA
```

MOVS语句中BETA和ALPHA的存在，就使汇编程序知道产生一个字节传送的MOVS指令（因为BETA和ALPHA的类型分量都是字节）。更进一步分析，汇编程序还能从BETA和ALPHA的段分量出发，确定MOVS指令的操作数是否在可存取的段中，这里忽略掉了ALPHA和BETA的偏移量分量。

与MOVS类似，还有其他4条包含操作数的串指令，其中MOVS和CMPS具有两个操作数，而SCAS、LODS和STOS都只有一个操作数。举例如下：

```
CMPS    BETA, ALPHA
SCAS    ALPHA
LODS    ALPHA
STOS    BETA
```

XLAT也需要一个操作数——送到BX中用作翻译表的项。这个操作数的段分量，使汇编程序能够确定翻译表是否在当前可存取段中，其偏移量被忽略了。下面是XLAT语句的一个例子：

```
MOV     BX, OFFSET TABLE
XLAT    TABLE
```

习　题

5.1 将下列10进制数转换成与其等值的2进制数、8进制数和16进制数：

139D，　2337D，　16383D，　7345D

5.2 设A=1011101001100011B，B=1100100011011101B

求A OR B，　A XOR B，　A AND B，　NOT A

5.3 阅读下面的程序段，并对之进行分析，指出它完成什么任务。

```
MY_DATA     SEGMENT
GRAY        DB          18H, 34H, 05H, 06H, 09H, 0AH, 0CH, 11H, 12H,
                        14H
MY_DATA     ENDS
MY_CODE     SEGMENT
            ASSUME      CS: MY_CODE, DS: MY_DATA
```

```
GO:         MOV     AX, MY_DATA
            MOV     DS, AX
            MOV     BX, OFFSET GRAY
CYCLE:      IN      AL, 1
            XLAT    GRAY
            OUT     1, AL
            JMP     CYCLE
MY_CODE     ENDS
            END     GO
```

用 ASM-86 语言设计如下程序：

5.4 找出两个数中的较大者.假定有 3 个字节变量 (x、y 和 z),试找出 x、y 中的较大者，并将其送到 z 中.

5.5 设有一字数组，它包含 100 个元素,试求出该数组各元素之和,并将和送至 SUM 当中(设 SUM 为一双字变量).

5.6 设有一字数组,它包含 100 个元素,试从该数组中找出最大者,并将它送至 MAX 单元当中.

5.7 编写一个过程,使它能够查出 AX 中 1 的个数,并将这个数目放到 CX 中.

5.8 假定有一个由 100个元素组成的字数组(这些数可以相同),它们已经按递增次序存放在始址为TABLE的数组中. 要求用 ASM-86，编写一段程序，把出现次数最多的数放到 BX 中,而把它出现的次数送至 CX 当中.

第六章　MCS-86 汇编语言程序设计

通过前一章的学习，我们已经对 8086 的汇编语言程序设计概念、思想及其基本方法有了一个基本的了解，对于只想了解一下 MCS-86 汇编语言的读者来讲，前一章的内容已经是够丰富的了，因此也就不一定有细读本章的必要。但是对大多数读者来讲，往往需要用汇编语言来编制程序，这样，就需要比较完整地掌握汇编语言的一些规定和细节，本章就是专为这些读者准备的。

§6.1　8086 汇编语言程序的基本组成部分

§6.1.1　引　　言

在 8086 汇编语言与 ASM86 之间有一个区别，后者(即 ASM86)实质上是一个将汇编语言翻译成目标代码(机器指令)的程序，但在后面的讨论当中，并不想将它们区别得很清楚。

用 8086 汇编语言所编写的程序格式是很随便的，也就是说，输入原程序行的各列之间，具有相当程度的独立性，在程序的各个分量之间，可以任意插入空格字符。但有一个例外，即对连续行而言，必须在前一行的终止符的后面，紧跟上一个(&)符号。

§6.1.2　ASM 86 的字符集

ASM86 中所使用的字符集，仅是 ASCII 和 EBCDIC 字符集的一个子集。ASM86 的有效字符集，由这样几个部分组成：

字母和数字：

ABCDEFGHIJKLMNOPQRSTUVWXYZ

abcdefghijklmnopqrstuvwxyz

0123456789

特殊字符：

+ - * / = ( ) [ ] < > ; ' . " , _ : ? @ $ &

及非打印字符：空格、制表(TAB)、回车、换行。

假如 ASM86 程序中，包含任何不属于上列字符集的字符，那末，汇编程序就将这些字符作为空格处理。紧跟在回车——换行后面的符号"&"代表一个连续行，但汇编程序也把它当空格处理，当然，在字符串或注释当中的情况除外。

除了在字符串当中之外，大写字母与小写字母之间是没有分别的，例如，xyz 与 XYz 之间是没有区别的，在程序中可以互换。但在下面的讨论中，为了便于将程序码部分与说明部分区别开来，故用大写字母来表示ASM86 的代码。

空格之间也是没有区别的(字符串常量当中除外)，任何空格都被认为是与别的空格相同，

进一步讲，一个不断的空格序列与单个空格等价。

特殊字符及其组合在 ASM86 程序当中具有特殊的意义，这在后面将做说明。

§6.1.3 ASM86 的语法元素

ASM86 源程序，可看作是由标记(Token)及分隔符按照一定的规则组织起来的。标记是 ASM86 源程序的最小的、有意义的单位，它非常象文章当中的字或词组。分隔符所起的作用就是将两个相邻的标记分开，以达到不会将它们错当作一个较长的标记之目的。最常用的分隔符是空格( )，此外，也可以将水平方向的制表键(tab)用作分隔符，这时，除了在文件列表时把它当作多个空格以外，汇编程序就把它 (tab) 看作是与空格相同的。任何非法字符，也都被当作空格处理。

空格可以任意插到标记之间，而不会改变 ASM86 语句的意义，因此，下面两个语句的意思在汇编程序看来是相同的：

```
MOV ITEM, [BX+3]
MOV       ITEM, [BX+  3]
```

一、界符

界符是一些特殊字符，利用它们可以表明某个标记的结束，它们本身也有一定的意义，这一点与分隔符不同，因为分隔符只表示标记的结束。在上面的例子当中，逗号、加号、方括号都是界符。若已有了界符，那就不一定需要分隔符。但是适当地使用一些分隔符，可使程序更易读、更易理解。

下表给出了 ASM86 中界符与分隔符的一些典型应用：

字 符	名 称	使 用 说 明	字 符	名 称	使 用 说 明
20H	空格	分隔或结束标记，提高程序的可读性	‹…›	尖括号对	在宏代码的内部，利用括号中的数给记录置初值
09H	水平 tab	分隔或结束标记，提高程序的可读性			
,	逗号	在多个操作数出现时，将后一操作数与前一操作数分开	$	美元符号	"位置计数器的当前值"的简写符号
			[...]	方括号对	变址或指针(下标)表达式
'...'	单引号对	字符串定界	=	等号	将字段宽度说明符与缺省的初值分开
(…)	圆括号对	表达式或子表达式定界，常用来提高程序的可读性或改变运算符的优先级别	−	减号	在两个数中间时，表示做减法；在一个操作数的左面时，指出它为负值。
0DH(CR)	回车	语句结束符(除非后面紧跟一个'&')			
0AH(LF)	换行		+	加号	在两个操作数中间表示做加法；在一个操作数的左面表示该操作数为正数。
CR-LF	回车换行				
;	分号	注释部分的界符			
:	冒号	用于标号、段跨越前缀、宏指令代码中的说明符、extrn 类型、设定(assume)部分，记录字段定义中	*	星号，乘号	表示乘法运算
			/	斜线，除号	表示除法运算
			?	问号	给某存贮单元一个不确定的初始值；与其他字符一起使用，形成标识符
.	句号或点	从记录当中取一字段的选择符；仅在宏代码中允许出现			
&		连续行指示符(在紧跟语句结束符之后时)	@		用来形成标识符
			_	下划线	用于组成标识符

上表中引进了一些新的概念，这些将在后面几节介绍。

二、常量

所谓常量，就是在汇编时已经确定的值，且它在程序的运行期间不会变化。常量分数值常量(整数)和字符串常量两类。其中的整数常量又可以用2进制、8进制、10进制或16进制数来表示。

2进制数由一串数字(0,1)组成，以"B"结束。例：

```
0B
1B
011000110101B
111111111111B
-0001000b
```

8进制数由一个(0, 1, 2, 3, 4, 5, 6, 7)数字序列组成，并以"O"或"Q"结束。例：

```
1234567O
-3Q
377O
0q
```

10进制数是一个由(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)所组成的数字序列，且以"D"结束(也可不写)。例：

```
10000
-6d
65535
```

16进制数是一个由 (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F) 所组成的数字序列，它以"H"作为终止符。为了使 ASM86 能够将它与标识符区别开来，所以规定这种常量必须以数字开头，而不允许开头为字母。显然，若常量的开头是一个字母，只要在其头上添上一个"0"，就可满足要求。例：

```
1H
-600H
0FFH
-0AAAh
```

所有这些常数都必须可用17位表示，其中包括一个符号位，即

```
-1111111111111111B<=2进制常量    <=1111111111111111B
        -177777Q<=8进制常量      <=177777Q
          -65535<=10进制常量     <=65535
        -0FFFFH<=16进制常量      <=0FFFFH
```

字符串常量，是由包含在单引号内部的一些可打印 ASCII 字符组成的。空格符和水平制表符 (tab)，都可以出现在这种字符串内，但回车和换行不可以在串中出现。汇编程序把这种字符串，表示成一个字节序列，这些字节包含了串内每个字符的 ASCII 码。例如：

```
'ABCDEFG'
```

'This is a string'

'Let"s include a "" character'

'123456¡"#$%&( )__—'

这些都为字符串。注意，在第3个字符串当中，包含了两个连续的撇号(')，以此来表示串中存在一个撇号。

长度为1的字符串，翻译成一个单字节数值，长度为2的字符串的值为一个字。例：

'A' 等价于 41H

'Ag' 等价于 4167 H

'#' 等价于 23H

长于两个字符的字符串，只可以用来初始化存贮器(见下一节)。可以使用单字节立即数的地方，就可以用单个字符组成的字符串；而可用字立即数的地方，也可用两个字符组成的字符串。

三、标识符

所谓标识符，就是对汇编程序具有特定的符号意义的字符序列。ASM86 中的标识符必须遵守下列规则：

1. 第一个字符必须是字母(A, …, Z, a, …, z)或为某个特殊符号(@, _, 或 ?)，但问号本身不能单独作为标识符。

2. 标识符首字符后面的字符，可以是规则1中给出的任何字符，也可以是数字(0, 1, …, 9)。

3. 标识符的有效长度为31个字符，若字符数超过31，就忽略后面的那些字符。

可见，下面这些例子都为有效的标识符：

A

WORD

FFFFH

Third_Street_and_Main

Should_We_Jump?

@variable_number_1234567890123456

@variable_number_12345678901234567

必须注意，汇编程序把最后的两个标识符当成同一个标识符了，这是因为它们的头31个字符是相同的。

下面这些是无效标识符：

First$ld (不允许 $ 作为标识符的组成部分)

0FFFFH (这是一个数，因为它是以数字开头的)

'Memphis' (包含在撇号内的字符组成一个字符串，而不是一个标识符)。

四、关键字

上面给出的有效标识符中有一个为 WORD，它就是一个关键字。关键字就是对汇编程序具有预定意义的标识符。一般来讲，关键字只可以用在其特定的上下文中，除非它们首先被

清除了(purge),但并不是所有的关键字都可被清除的。下面这些标识符都是关键字:

AX

Segment (记住:大写字母与小写字母是等价的)

END

MOV

表 5.1 列出了所有的关键字。

五、符号及其属性

用户为了表示某个存贮器单元、数据、表达式或代码(或数据)结构所定义的标识符,就叫做符号 (Symbol)。实际上,汇编程序已预先对寄存器集及段定义过了,当然,根据需要,用户也可以重新定义寄存器符号。可见,这里使用的"符号"具有一定的特殊意义,它不同于常用的标记名称。

我们可以把符号分成五类,即:

寄存器

变量

标号

数

其他(段、组、记录、记录字段、宏指令代码、形式参数)。

每个符号都具有一定的属性,以允许汇编程序使用该符号来代表所需的信息。

1. 寄存器

每个寄存器都属于某种类型,由这些类型可以确定出特定的寄存器是一个字节寄存器还是字寄存器(两个字节),8086的标志位被看作是一位寄存器。下表是 8086 所有预定义寄存器的一个划分。

类型	包含的寄存器	大 小	其他已知信息	类型	包含的寄存器	大 小	其他已知信息
H	AH	一个字节	累加器的高字节	I	SI	两个字节	源变址
	BH	〃	基址寄存器的高字节		DI	〃	目变址
	CH	〃	计数寄存器的高字节	S	CS	〃	代码段寄存器
	DH	〃	数据寄存器的高字节		SS	〃	栈段寄存器
L	AL	〃	累加器的低字节		DS	〃	数据段寄存器
	BL	〃	基址寄存器的低字节		ES	〃	附加段寄存器
	CL	〃	计数寄存器的低字节	F	AF	一位	辅助进位标志
	DL	〃	数据寄存器的低字节		CF	〃	进位标志
X	AX	两个字节	累加器(整字)		DF	〃	方向标志
	BX	〃	基址寄存器(整字)		IF	〃	中断允许标志
	CX	〃	计数寄存器(整字)		OF	〃	溢出标志
	DX	〃	数据寄存器(整字)		PF	〃	奇偶位
P	BP	〃	基址指针		SF	〃	符号位
	SP	〃	栈指针		TF	〃	自陷位
					ZF	〃	0位标志

通用寄存器(类型为 X、H 或 L)可以用作大多数指令的源操作数或目标操作数,某些寄存器还有些专用性能,例如:BX 用于计算地址,CL 可用在某些指令中作为计数寄存器等

等。至于标志位，它们是不能单独寻址的，但其内容却可以由一些算术逻辑指令或关系指令所改变，有些数据传送指令，也可能改变这些标志位。

2. 变量

利用变量来确定放在存贮器中的数据。所有的变量都具有三重属性：

1) 段分量(在定义该变量时，正在汇编的是哪个段)

2) 偏移量(从该段的起点到此符号单元之间的字节数)

3) 类型(在引用该变量时，要对多少数据字节实施操作)

变量的类型必为下列三种类型之一：

BYTE (1 字节长)

WORD (2 字节长)

DWORD (4 字节长)

变量通常由下述方式定义：

1) 作为存贮器初始化伪指令中的名字。例：

Variable_1 DB 0 (见 §6.2)

2) 作为 LABLE 伪指令的名字。例：

Variable_2 LABLE BYTE (见 §6.3)

3) 作为 EQU 伪指令的名字。例：

Variable_3 EQU Another_variable (见§6.3)

§6.2 对变量进行了较为详细的解释。

3. 标号

标号代表存贮器单元，这些单元中包含了跳转或调用(call)指令需要引用的指令代码。所有的标号都具有 4 种属性：

1) 段(类似于变量中的段)

2) 偏移量(类似于变量中的偏移量)

3) 距离(类似于变量的类型分量，它指明了为了到达该标号，是需要 2 个字节的偏移量呢？还是需要一个段——偏移量对(4 个字节))

4) CS 设定(指出在定义此标号时，CS 中被假定的东西)。

对汇编程序来讲，标号和变量的意义是类似的，只是前者所引用的是存贮器中的指令代码，而后者引用的是存贮器中的数据。这也正是两者的属性相似的原因。标号的段分量和偏移量分量的定义，是与变量类似的。其距离分量只可能取两个值：

NEAR：所有对该标号的引用，仅使用两个字节，它是"自相关的"，也就是说，为了到达此位置，必须改变的只是 IP 的内容，而不需改变 CS 寄存器的内容。

FAR：所有对该标号的引用，都需要改变 IP 和 CS 寄存器的内容，跳转到该标号的跳转指令及调用该标号的 CALL 指令，都必须赋与 IP 及 CS 以新值。

NEAR 或 FAR 的选择，依赖于此标号是否会被 Jump 或 Call 所引用，以及 Jump 或 Call 的 CS 设定，是否不同于此标号的 CS 设定。这通常意味着从定义该标号的代码段的外面，来引用这个标号。

倘若所有的引用所使用的是同一个 CS 设定，那末就应该把标号的距离属性定为 NEAR；否则说明是 FAR。若没有对标号进行距离属性规定，就假定它为 NEAR。

NEAR 就意味着告诉汇编程序，跳转到该标号的任何跳转，都可以通过一个 16 位的自相关偏移量实现；而 FAR 总是需要 2 个字，其中之一放偏移量，另一个存放该标号所在段的段基址.

假如汇编程序发现要到达某标号，需要一个段间跳转或调用("长跳")才能实现，而此标号又被说明是 NEAR 的，那就置上出错标志.

由此可见，在汇编程序确定为了转到某个标号需要哪种跳转或调用时，CS 的设定是很关键的. §6.3 将对这个概念给出较为完整的解释.

标号一般是由下面几种方式定义的：

1) 加在指令的前面，在标号与指令中间用冒号(:)隔开. 例如：

```
Lable_1: ADD AX, BX
```

2) 作为 LABLE 伪指令的名字出现. 例：

```
Lable_2 LABLE FAR
```

3) 作为 EQU 伪指令的名字出现. 例：

```
Lable_3 EQU THIS NEAR
```

4) 作为 PROC/ENDP 对的名字. 例：

```
Lable_4 PROC
    ⋮
Lable_4 ENDP
```

详细介绍请见 §6.3.

4. 数

符号也可以被定义来表示一个纯粹的数，而不是代表某个寄存器或存贮器单元 (地址). 在使用这种符号时，它所代表的数好象已被显式编过码了. 例：

```
Number_5 EQU 5
MOV AL, Number_5
```

就等价于语句

```
MOV AL, 5
```

可见，符号 Number_5 不是代表某个特定的存贮器地址，而是代表数值 5.

如果把某个符号定义为一个数，那末它就代表一个 16 位的值(在汇编期间还包括一个符号位——17 位)，对这些数的处理是相当直观的，详细解释请见 §6.4. 应该记住，一个或两个字节的字符串也可用作数值，例：

```
Initials EQU 'AA'
MOV AX, Initials
```

5. 其他符号

除了上面所讲的 4 种符号之外，还定义了一些别的符号，把它们用作汇编程序伪指令当中的名字. 需要这些符号的伪指令是：

SEGMENT/ENDS	(定义一个段名)
GROUP	(定义一个组名)
RECORD	(定义一个记录名以及记录内的任何记录字段的名字)
CODEMACRO	(定义一个宏指令代码，以及其中用到的形参名，但应该注意，一

旦用宏代码定义了某个符号，就不再把它看成是一般的符号，而是把它看作是一个指令助记符）

EQU　　（除了定义变量和标号之外，还可把 EQU 用来命名表达式）

SEGMENT/ENDS，GROUP，RECORD 及 EQU 的详细解释将在 §6.3 中给出，CODEMACRO 的详细介绍请见 §6.5。

利用 EXTRN 伪指令，就可以引用非局部定义的符号，也就是说允许在当前模块中，引用其他模式中产生的变量表，但需要保持类型（即 BYTE，WORD，DWORD，FAR，NEAR 或 ABS—— 用于数）的一致性。例：

```
EXTRN little:BYTE, medium: WORD, bit:DWORD, close:NEAR,
  distance:FAR, x:ABS
```

伪指令 EXTRN 的进一步解释，及其在连接程序中的应用说明，请见 §6.3。

§6.1.4 语　句

就象可把标记看作是字或词组一样，我们也可把这里的语句看成是平常的句子。对8086的汇编程序来讲，语句就是一个完成什么动作的说明。事实上，可把计算机的程序看成是一个语句序列，它负责完成某个特殊功能。语句可分成两类：

指令（指令性语句）：汇编程序要把它们翻成机器指令代码，这些代码命令 8086 完成某些动作。

伪指令（指示性语句）：汇编程序并不把它们翻成机器指令代码，它们只是引导汇编程序，去完成某些事务性的功能（注意：在装入程序时，存贮器初始化伪指令确实将信息送到 8086 的存贮器当中。但是，应该看到，这通常都是用作数据，如为变量提供初值，而不是表示“执行”）。

指令助记符可以由汇编程序预先定义，也可以由用户通过宏指令代码伪指令来规定（见 §6.5）。实际上，汇编程序所能识别的，也仅仅是那些这样定义过的指令。在任何时候，都可以改变、加入、重新定义或注销指令助记符。

另一方面，伪指令却是固定的，它们是汇编语言的固有特征，它们在很大程度上规定了 ASM86 的性质。当汇编程序碰到某个伪指令时，它总是完成相同的、特定的动作，伪指令既不能消灭也不能产生。

在源程序文件当中，一个语句通常占据一“行”的空间，所谓一行就是以终止符（回车、换行或回车/换行组）结束的一个字符序列。应该看到，由于 ASM86 提供了“连续行”特性，这样，一个语句就有可能在源程序文件当中，占据几个物理行。如果在终止符后面的第一个字符是“&”，那就表示这是一个连续行。但是，一个“符号”不可以分到几个连续行中，同样字符串也不可以简单地分开来，它必须在一行中用一个撇号括起来，然后再在下一个连续行中，用一撇号把上行尾的字符串重新打开。对注释语句来讲，碰到终止符也认为它结束了，假如希望注释继续下去，那末跟在“&”后面的第一个非空字符必须是“;”。

还存在这样的伪指令，它们的编码必须多于一行，这将在后面介绍。

一个语句的编码格式是相当灵活的，这主要表现在标记可以在源程序行的任何位置出现，唯一的限制只是用于代表连续行的“&”。然而，对应于语句的各个组成部分的位置，只允许某些特定类型的符号出现。更进一步讲，由指令及伪指令的格式就可以将两者区别开来，并对它

们的各个字段作出不同的解释。

指令性语句的格式为：

标号　前缀　指令助记符(操作码)　操作数；注释

各部分的定义是这样的：

标号：它是一个后跟":"的符号，它是以当前段的当前位置计数值来定义的。标号总是可供选择的。

前缀：有几条机器指令只可以用作其他一些指令的前缀，如 LOCK、REP。这个部分也总是可供选择的。

指令助记符：由汇编程序或用户通过 Code Macro (宏指令代码) 伪指令所定义过的符号，就称为指令助记符。它也是可供选择的，但若被省去，就不能出现操作数，而其他部分可以不受影响。在任何时刻，汇编程序所能识别的所有指令助记符，组成了该汇编程序的"指令集"。

操作数：一个指令助记符，可能需要其他符号跟随其后，这些符号将作为该指令的作用目标操作数就是指这些符号，ASM86 提供的指令，可能不需要操作数，也有可能需要一个或两个操作数。利用 Code Macros，用户还可以定义一些其他的需要多于两个操作数的指令。操作数之间是用逗号分开的。

;注释：出现在字符串外面的任何分号(;)，都意味着一个注释的开始，至于注释的结束则是一个行终止符。注释一般用于文本化程序，它可提高程序的可读性。注释在程序中总是可供选择的。

例：

```
LAB1: MOV AX, BX
        :
LAB2: MOV CX, ALF [SI]
```

指示性语句(伪指令)的格式为：

名字　指示符(伪指令)　操作数　;注释

其中，各字段的定义是这样的：

名字：这里的名字绝对不会与指令性语句中的标号相混淆，因为它不可能以一个":"结束。对于有些指示符来讲，这样的名字是需要的 (这种类型的指示符是 SEGMENT, ENDS, GROUP, RECORD, LABLE, EQU, PROC,ENDP)；而其他的一些指示符，不允许使用这种名字(如PURGE, NAME, ASSUME, ORG,PUBLIC,EXTRN, END)；存贮器初始化指示符(DB, DW, DD)允许该名字可选择地出现；Codemacro 指示符对上面的指示性语句的描述来讲，是一个例外，后面我们将对它作专门描述。

指示符：它是汇编程序所定义的 20 个关键字中的某一个，它完成各种汇编时的功能，为程序员在进行存贮器分配、模块间通讯，符号的处理时提供支持。

操作数：类似于指令助记符的操作数。有些指示符允许操作数的任一表格(如DB, DW, DD, PUBLIC, EXTRN, PURGE) 作为它的操作数；其他一些指示符，允许这些特殊的关键字将某些属性透露给正被定义的实体 (如：SEGMENT, PROC)；根据指示符的需要，确定是否要这些操作数。

;注释：与指令性语句中的定义完全相同。

例：

```
RRR1 EQU AX
```

前面已经提到，CODEMACRO 是上面公式化描述的一个例外，它的形式是：

```
CODEMACRO  名字  操作数  ;注释
```

即名字出现在关键字 CODEMACRO 之后。

有些指示符是成对出现的，它们是：

SEGMENT/ENDS

PROC/ENDP

CODEMACRO/ENDM

对 SEGMENT/ENDS 和 PROC/ENDP 对来讲，出现在头一个指示符中的名字，也必须在其对应的第二个指示符中出现。例：

```
Seg-1 SEGMENT
需要配对的
Seg-1 ENDS
```

指示符 ENDM 可以包含相应的 CODEMACRO 中的名字，但这不是必需的。

§6.1.5 模块(MODULES)

模块是一个汇编单位，也就是说，在汇编某个源程序文件时，其目标文件就定义了一个模块。一个程序可以由几个模块组成(若使用模块程序设计方法)，这些模块常常具有不同的功能，利用 MCS-86 软件包中的重定位和连接程序，就可把它们组织起来，形成所需要的程序。

§6.2 变　量

在设计一个程序或系统时，总必须先给出控制流程，即给出处理输入命令或数据时，计算机所遵守的执行步序。这样做的目的，就是以对下述人员方便的方式，完成各自的任务，有关人员是设计者，程序员以及将来某个时候可能需要修改代码或加入代码的工作人员。

程序设计的一个重要特色，就是其中具有引用多次的单元、过程及数据，这一点至少对存贮器容量受到限制的机器来讲是有意义的。在程序当中，往往不是使用数值地址，而是利用一些名字来寻址某些项目，这样做要方便得多。可用名字来寻址的项目有

a. 标号，用于引用所选择的指令(码字)，见 §6.3

b. 变量，用于访问数据

c. 数，立即数的引用

d. 表达式，带变址的量或更为复杂的一些表达式的引用。

本书当中，标号几乎总是表示代码，它仅与 CS (代码寄存器)相关联，唯一的例外情况只会在指示性语句 LABLE 中出现(见 §6.3)；变量总是意味着数据，尽管 DS 是通常所使用的段寄存器，但也并不是说，对变量的段寄存器有什么特别的限制。

标号与数据有这样一些区别：标号必须有一个取值为 NEAR 或 FAR 的距离属性，并且能在其名字之后使用一个冒号来定义；而变量具有如 BYTE 这样的一种类型属性，不会有

NEAR 或 FAR 这样的距离属性，变量也不可以用冒号来定义。

名字的方便性还表现在，它至少可用五种方式扩充一些更大的代码块：

1．命名象过程这样一些代码序列，它们仅需定义一次，然后可以在不同地点调用执行它们；

2．命名象宏指令代码这样的代码序列，根据几个可变参数定义它们，然后只要利用其名字以及所需的实参就可以使用它们；

3．用于代码块的结合；

4．检查名字与寄存器使用的一致性；

5．在调试过程中实现错误定位。

为了获得最后的三个命名块特性，都要使用段寄存器。

后面将讨论这样一些内容：

1．如何建立这些名字？

2．在随后的表达式中，这些名字的缺省值是什么？

3．为了改变或加入这些名字的自动结果，应作何种选择？

4．如何使用名字。

标号在 §6.1，§6.3 和 §6.4 中讨论，宏指令代码在 §6.5 中讨论，过程在§6.3 中讨论，段也在§6.3中讨论，下面主要讨论数据命名、存贮器分配及其初始化几个问题。

§6.2.1 变量说明及其初始化

由于数据能够根据位、字节、字、双字或其他一些分组来定义，所以有必要告诉汇编程序，它需要多少存贮单元。

三种存贮器分配单位是：

1．字节——使用 DB 定义；

2．字——使用 DW 定义；

3．双字——使用 DD 定义。

在为数据分配存贮器时，必须确定其初始内容，倘若不知道或不需关心该单元的初始值的话，那末就应使用一个问号(?)，来表示不要给此单元某个特定的初始值。

一、DB、DW 和 DD 指示符

使用 DB、DW 和 DD 有两个目的：(1)初始化存贮器；(2)定义变量。它们的意思分别是“定义字节”、“定义字”及“定义双字”。

二、变量的定义

一个变量可以用 DB、DW 或 DD 来定义，所需要的变量名出现在指示符(DB、DW 或 DD) 的左面。变量具有三个寻址分量：SEGMENT、OFFSET 和 TYPE (段、偏移量和类型)。指示符给出了变量的类型(即 DB 为字节，DW 为字，DD 为双字)，变量汇编时的偏移量等于段中到此变量的字节数，段分量就是当前段的始地址。

若某个变量所命名的是一个数组(向量)，那末它的类型，就是变量的单个元素(分量)所占

用的字节数目。

例：

```
TABLE_DATA  SEGMENT
    TABLE   DW    12
            DW    34
    NUM1    DB    5
  TABLE_TWO DW    67
            DW    89
            DW    1011
    NUM2    DB    12
    RATES   DW    1314
OTHER_RATES DD    1718
TABLE_DATA  ENDS
```

上面所有变量的段分量都为 TABLE_DATA，DW 定义了一个字变量（两个字节），DB 定义了一个字节变量，TABLE_TWO 的偏移量是5，这也是从段始点到该变量之间的字节数目。RATES 的偏移量是12。

变量 NUM1 和 NUM2 的类型为1，表示字节。OTHER_RATES 的类型为4，表示双字。TABLE_DATA 中的所有其他变量的类型分量都为2，表示字。

标识符	段分量	属性	
		偏移量	类型分量
TABLE	TABLE-DATA	0	2
NUM1	〃	4	1
TABLE-TWO	〃	5	2
NUM2	〃	11	1
RATES	〃	12	2
OTHER-RATES	〃	14	4

三、存贮器的初始化

DB、DW 和 DD 这三个指示符，还可以用于初始化存贮器。由上例可见，在指示符的右面可以出现一个表达式，其作用就是将该表达式的值，作为某个存贮器"单位"的初值，一个存贮器"单位"，可以是一个字节(DB)、字(用于 DW)或双字(用于 DD)，即它们分别代表了1个字节、2个字节或4个字节。

1. 表达式

利用表达式可以初始化存贮器，第四节将专门对表达式进行讨论，这里我们只要知道表达式有数值表达式和寻址表达式两类就够了。5是一个数值表达式，同样 4*50 也是数值表达式，而标号就可看作是地址表达式。在使用地址表达式来初始化存贮器时，这样的表达式只可以在指示符 DW 或 DD 中出现，绝不允许它出现在 DB 当中。"DW 变量"表示利用变量的偏移量来初始化一个存贮器字，而"DD 变量"将用该变量的段分量和偏移量，来初始化存贮器的两个字。

例：

```
FOO SEGMENT AT 55H
ZERO   DB  0              ;ONE BYTE OF0
ONE    DW  ONE            ;ONE WORD OF  1(0001H)
TWO    DD  TWO            ;LOW WORD OF  3(0003H),HIGH WORD O
                              55H  (0055H)
FOUR   DW  FOUR+5         ;ONE WORD OF  12 (000CH)
  SIX  DW  ZERO-TWO       ;ONE WORD OF -3 (0FFFDH)
 ATE   DB  5*6            ;ONE BYTE OF 30
 FOO ENDS
```

其中，"TWO DD TWO"提供了这样一些信息，首先，指示符 DD 分配 4 个存贮器字节；其次，它还为这 4 个字节设置了初值，头 2 个字节所组成的字，装入的是变量 TWO 的偏移量(偏移量等于3)，第 2 个字 (后 2 个字节) 中的初始内容是 55H，它是段 FOO 的开始地址。因此，DD 中的 2 个字表示符号 TWO 的绝对地址，这也是指示符 DD 的主要功用。

2. 问号：无特定初值

在汇编语言中，单个问号也是一个关键字，它只可以用在存贮器初始化的过程中，并且表示随便汇编程序给此存贮器单元一个什么样的初值，即该单元内的内容是不确定的。

例：

```
DB  ?
DW  ?
DD  ?
```

它们分别保留 1 个字节、2 个字节和 4 个字节，但都未经初始化设定初值。

3. DUP

DUP 利用给出的一个初值(或一组初值)以及这些值应该重复的次数，实现存贮器的初始化功能。利用一条 DUP 命令就可以初始化一个较大的存贮区域。DUP 的格式为：

表达式 DUP (项)

这里的表达式，是一个其值大于 0 的数值表达式，项可以是一个表达式(地址或数值表达式)、问号、由多个项组成的表或 DUP 重复，项必须用圆括号括起来。

例：

```
     DB  100  DUP  (0)          ;100个字节，其内容都为 0
     DW  10   DUP  (?)          ;10个字，其值不知道
FOO  DD  50   DUP  (FOO)        ;FOO 的地址(段和偏移量)的50个拷贝
     DB  10   DUP  (10 DUP(0))  ;0 的 10 次重复的 10 次重复 (100 个 0)
     DW  35   DUP  (FOO, 0, 1)  ;FOO 的偏移量、0 和 1 这三个字的35次重复
```

4. 表

从上面的例子可以看出，在初始化存贮器时，DUP 可以使用一个用圆括号括起来的项目表，表 (FOO, 0, 1) 代表一个待重复的实体。任何能够单独出现的客体，包括表和字符串，都可以作为一个表元。举例来说：

```
DB  5  DUP  (1, 2, 4 DUP(3), 2 DUP (1, 0))
```

这个 DB 指示符初始化 50 个字节，它们是下面这些字节值的 5 份拷贝：

1, 2, 3, 3, 3, 3, 1, 0, 1, 0

ALPHA DW 2 DUP (3 DUP (1, 2 dup (4, 8), 6), 0)

这个 DW 指示符初始化 38 个字，它是下面这些字值的 2 份拷贝：

1, 4, 8, 4, 8, 6, 1, 4, 8, 4, 8, 6, 1, 4, 8, 4, 8, 6, 0

没使用 DUP 的单个表格不可以加圆括号，下面这些表格所获得的存贮器定义及其初值，是与上面的 DW 语句相同的：

DW 1, 4, 8, 4, 8, 6
DW 1, 4, 8, 4, 8, 6
DW 1, 4, 8, 4, 8, 6, 0
DW 1, 4, 8, 4, 8, 6
DW 1, 4, 8, 4, 8, 6
DW 1, 4, 8, 4, 8, 6, 0

5. 字符串

一个字节可以保存某个字符的 ASCII 代码。'ABCDE' 是由 5 个字符组成的一个字符串，保存该字符串需要 5 个存贮器字节。

假如希望用一些字符来初始化存贮器，那末只要用单引号将这些字符括起来就可以达到目的。但在进行存贮器的分配和初始化时，长于 2 个字符的字符串，仅仅在 DB 命令当中才是合法的，而在 DW 和 DD 命令当中，却不允许出现多于 2 个字符的字符串。

例：

```
PART1   DB  'THANKS'
PART2   DB  'LOT'
LINE1   DB  'THANKS A LOT'
BUFFER  DB  128 DUP ('')    ;置 128 个字节的初值为空格
LINE    DB  80 DUP (72 DUP '', 0DH, 0AH)
                    ;初始化 80 行，其中每一行都具有 72 个空格，且以回车、换行
                    ;作为结束
```

因为每个字符都占用一个存贮器字节，所以上面的每条 DB 命令，实际上都保留或初始化了多个存贮器字节：PART1 得到 6 个字节，PART2 得到 3 个字节，LINE1 获得了 12 个字节。如果接下来执行下面两个语句：

```
MOV  PART1, ' '
MOV  PART1+1, 'B'
```

那末开始于 PART1 的 6 个字节的字符串就成为' BANKS'。存贮器单元 PART1 仅为一个字节，其中所保存的是刚送进去的一个空格字符，但是利用它可以访问整个字符串' BANKS'，PART1 是该字符串的始地址。利用下面这个语句可以得到相同的结果。

```
MOV  WORD PTR PART1, 'B '   ;之所以用'B '，是因为装入存贮器字节
                            ;的次序是反向的，即先装' '，后装'B'
```

例：

```
INVENTORY_ACCESS  SEGMENT
```

```
FILTER_1   EQU 4              ;FILTER_1 is now a
                              ;name for the absolute number 4.
SWITCH_1   DB  ?
LEVELS_1   DB  4 DUP(?)
ACCESS_1   DW  FILTER_1
STORES_1   DW  FILTER_1 DUP(0, 1)
SWITCH_2   DB  ?
LEVELS_2   DB  0, 1, 2, 3
ACCESS_2   DW  FILTER_1+2
STORES_2   DW  (FILTER_1+2) DUP(1, 2)
INVENTORY_ACCESS    ENDS
```

在上面这个例子当中，EQU 后面的4条命令，一共分配了23个字节的存贮器空间。SWITCH_1 分配到一个字节，它未经特定的初始化处理；LEVELS_1 分配4个字节，也没给它们初始值；ACCESS_1 分配1个字，其初值为 FILTER_1 即4；STORES_1 分配8个字，它把 FILTER_1 用作 DUP 的控制表达式，这8个字的初值分别为 0, 1, 0, 1, 0, 1, 0, 1。

第二个命令组，共分配及初始化了31个存贮器字节。其中，SWITCH_2 分配到一个未给初值的字节；LEVELS_2 给它本身的字节值为0，而给接下来的3个字节的初值分别为1, 2, 3；ACCESS_2 分配一个字，其初值为 FILTER_1+2，即为6；STORES_2 分配12个字或曰24个字节，这12个字的初值分别为 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2。

变　　量	TYPE	LENGTH	SIZE
SWITCH—1	1	1	1
LEVELS—1	1	4	4
ACCESS—1	2	1	2
STORES—1	2	8	16
SWITCH—2	1	1	1
LEVELS—2	1	4	4
ACCESS—2	2	1	2
STORES—2	2	12	24

6. 字和双字

DW 产生16位值，而 DD 产生32位的值。

```
EARLY     EQU     3
MIDDLE    EQU     1041          ;=0411H
FINAL     EQU     28672         ;=7000H
BLAST     EQU EARLY *MIDDLE
HEAT1  DW EARLY *MIDDLE         ;=0C33H
HEAT2  DW (FINAL+BLAST) *EARLY  ;=95385=17499H (ERROR)
HEAT3  DW (FINAL+BLAST) *EARLY/4 ;=95385=17499H (ERROR)
HEAT4  DW (EARLY/4) *FINAL+BLAST ;NOT AN ERROR
```

为了初始化 HEAT1 所计算出来的值，将装在一个字 (16 位) 当中，它必须比 0FFFFH 起码小 2，因为 HEAT2 和 HEAT3 都超过这个数，所以它们是不正确的，这两个变量也就保持为未经定义状态。但是如果在执行期间计算出这样的结果，就会自动地将结果截取为 16 位，例如 17499H 就成为 7499H。

必须强调的是，在存贮字时，总是将低字节放在地址编号较低的存贮单元，而将高字节放到下一个较高的地址单元中。存贮双字的情况也与此相似。因此，HEAT1 的初值，将这样存放 33H 0CH，而 HEAT2 干脆就相当于没经定义。

更一般地讲，存贮器的内容是按照地址的递增次序从左到右、或从上到下表达的。因此

```
DW  1234H
DW  5678H
```

在存贮器中就变成 34H 12H 78H 56H，或从上到下把这些字节表示成：

```
34H
12H
78H
56H
```

所以，上面那个语句中，HEAT1 的初值为 33H 0CH，存在内存中的格式为

```
33H
0CH
```

命令 DD 分配两个字，设置这个命令的一个目的，就是为今后保存某个标号或变量的段分量和偏移量做准备，其中的标号或变量的定义不在当前段中。

2 个字就是 4 个字节，由于每个字节可以存放 2 个 16 进制位，所以一个双字就可以存放 8 个 16 进制位。但是，该汇编程序所允许的最大常量是一个 17 位数 (一个符号位加上 16 个有效的数据位)，即一个字，这也是与 8086 的结构及其机器指令一致的，因为指令 (乘法与除法除外) 允许的操作数，不能大于 16 位所能表示的数值，所以可能取的最大数值是 0FFFFH。

因此，若给出语句

```
DD  1234H
```

那末就把双字的高位部分假设为 0000H，分析一下就可以看出，这也是与上面的字节存放规则一致的。所以

```
DD  1234H
```

意味着数值为 0000H 1234H，它是与下面的两个 DW 的作用相同的：

```
DW  1234H
DW  0000H
```

这是因为两者都使存贮器的内容变为

```
34H
12H
00H
00H
```

假设需要存放两个非 0 的字，比如说 8765H、9423H 这样的一个常量或段/偏移量时，就需要两个 DW 命令：

```
DW  8765H
DW  9423H
```

在存贮器当中，它是这样存放的：65H、87H、23H、94H，或记为：

```
65H
87H
23H
94H
```

注意：DD 命令，常被用来建立地址空间，用它来表示某个 8086 地址的偏移量和段分量。根据 8086 的规定，表示地址的这一对字，总是先为偏移量、后为段分量。因此，在上面那对字中，8765H 是偏移量，9423H 可被看作是段分量。

存贮器中的这种字节反向贮存技术，通常只有在进行内存转贮(如调试时)时，才显示出其重要性，而在大多数情况下，硬件会自动地遵守、利用这一协议，不需程序员考虑这一问题。例如：

```
LOC1:  MOV AX, 'NO'
LOC2:  MOV MEMWORD, AX
LOC3:  MOV BX, MEMWORD
LOC4:  MOV MORWORDS, BX
```

执行过程如下：

在 LOC1，AH 中装入的是表示 N 的 4E，AL 中是表示 O 的 4F：

```
4E  4F
AH  AL
```

在LOC2，MEMWORD 的低位字节装入 4F，高位字节装入 4E：

```
4FH
4EH
```

或表示为

```
4FH    4EH
低字节  高字节
 MEMWORD
```

在 LOC3，BH 得到 4E，BL 获得 4FH:

```
4E  4F
BH  BL
```

在 LOC4，MORWORDS 的低字节得到 4F，高字节得到 4E：

```
4FH
4EH
4F      4E
低字节  高字节
 MORWORDS
```

存在这样一种情况，即有时需要把一个字中的两个字节，当作单个字节处理，这时就非常有必要了解各字节的存放顺序，例如：

```
VECTORB  LABEL  BYTE   ;给下面这个单元另一个名字,且其类型为"BYTE"
                       ;而不是"WORD",从而允许用字节访问
VECTOR   DW   1234H
          ⋮
         MOV AL, VECTORB
```

结果是将 34H 送到 AL，而不是把 12H 送到 AL。

借助于 DW 或 DD，能够使用两个字节的字符串(但不能再长)，不过必须记住反向存贮的规定。例如：

```
SIGNAL1  DW  'GO'
SIGNAL2  DW  'NO'
```

汇编程序把它们都解释为两字节的数，即字符的 ASCH 码值，G 的 ASCII 码为 47H，O 为 4FH，N 为 4EH，因此上面的 DW 命令等价于：

```
SIGNAL1  DW  474FH
SIGNAL2  DW  4E4FH
```

按地址的递增方向，得到这些存贮单元的内容为 4FH 47H 4FH 4EH 或为 'OGON'。

在把单个字符的串用来初始化字或双字变量时，由于它们仅仅占用了 2 个字节 (DW)或 4 个字节(DD)中间的一个字节，所以需要遵守补 0 规则。例如：

```
SIGNAL3  DW  'K'
SIGNAL4  DD  'P'
```

它们与下面的命令对完全相同：

```
SIGNAL3  DW  4BH
SIGNAL4  DD  50H
```

或

```
SIGNAL3  DW  004BH
SIGNAL4  DD  0050H
```

根据前述的存贮规则有：4BH 变成了 4BH 00H，50H 却成为 50H 00H 00H 00H。

§6.2.2 几个属性运算符 (Length, Size, Type)

回忆一下 LINE1 的定义：

```
LINE1  DB  'THANKS A LOT'
```

假如在程序的后面某个地方，希望组成一些信息，那末能够了解由 LINE1 指向的字符串的长度，也许是相当重要的，运算符 LENGTH 提供了这一功能。指令

```
MOV  AX, LENGTH LINE1
```

将把 12 或 000CH，送到累加器 AX 当中。在早期的汇编程序当中，要获得上面这一条指令的功能，需要使用两个标号和一次减法。

类似地，在用 DB、DW 和 DD 定义了变量之后，当然也就可以在指令当中使用它们。在许多场合，为了能够指向取数据或控制转移的正确位置，有必要根据定义的单位及其数目，了解它们起先是如何被定义的。为了达到这些要求，各种表格、循环及向量都需要精确的指针。

利用一个"主表"、或在运行过程中，利用这种列表技术来查找所需的信息，你就可以很自然地确定某个名字，是代表字节还是代表字。但是，汇编程序能够自动地完成该任务。

为了产生正确的机器指令，就用到变量所隐含的类型，倘若代码指令与所用的数据定义不相符合，汇编程序就指出这是一个错误。

此外，汇编程序还提供了一些特殊的运算符，这些算符可用在需要类型信息的那些表达式当中，在建立(或调用)通用过程时，这些算符是特别有用的，这里的通用过程，是指无论给出何种参数，它都提供相同的处理过程。

在许多的文本当中，利用名字或表达式，总比用一个显式的数要好。这是因为在读列表文件时，名字比较容易理解，假如需要的话，利用名字也使修改方便多了，因为名字的单次定义(或改变)，就可应用于程序中名字的每次引用。

汇编程序的运算符 SIZE、LENGTH、TYPE，提供了上述功能，TYPE给出所定义的基本单位中的字节数，例 TYPE LINE1为 1，因为它的基本单位是一个字节；TYPE SIGNAL 3为 2，因为 SIGNAL 3 的基本单位是一个字，即 2 个字节；TYPE SIGNAL 4 为 4，这是因为SIGNAL4 的基本单位是一个双字，4 个字节。

SIZE 得到的是由说明该名字的整个行所定义的字节数目，而 LENGTH 得到的是所使用的基本单位数目。例如，在下面这个说明中：

```
PATH1    DW    1234H,    5678H,    0ABCDH
```

它初始化 3 个字，PATH1 指向其中的第一个。

TYPE PATH1 等于 2，表示其基本单位是一个字；LENGTH PATH1 等于 3，因为该 DW 分配及初始化了 3 个单位；SIZE PATH1 等于 6，因为存贮 3 个字需要 6 个字节。但对 LINE1而言，由于它是用字节定义的，所以有 LENGTH LINE1＝SIZE LINE1。

它们之间的关系可用下面这个公式表示：

SIZE 名字＝LENGTH 名字*TYPE 名字

SIZE 的一种用途，是检查某个索引或下标是否越出了对应的表格或向量的范围。在希望指向某表的下一项时，利用 TYPE，可以把循环计数器或索引指针的内容增/减一个正确的字节数目。例如，在以字节定义的表中

```
LIST_EXMPL  DB  500 DUP (13, 21, 34)
```

正确的增量为 1，对字而言其增量为 2，若为双字就应取 4。上面这个语句一共定义了 1500 个字节，其中第一个字节的名字是 LIST_ EXMPL。假设将这 1500 个字节看作一张表或一个向量，那末利用一个索引或下标，来存取这些表元就很有意义。例如，在下面这个变量名中，利用 SI(或DI)作为索引，是相当有用的。

```
LIST_EXMPL [SI]
```

当 SI 为 0 时，它就寻址第一个字节；而 SI 等于 5 时，它就访问第 6 个字节。

下面这些例子，说明了以上运算符以及 DUP 的应用情况：

例 1：ZERO_ARRAY DW 1000 DUP(0)

它把 1000 个字的存贮块初始化为 0，即 2000 个 0。

```
TYPE ZERO_ARRAY=WORD=2
LENGTH ZERO_ARRAY=1000
SIZE ZERO_ARRAY=2000
```

例 2： BUFFER DB 256 DUP(' ')

BUFFER 为一个 256 字节的向量，各字节的内容都为空格。

例 3： FIB DW 1,1,100 DUP(?)

FIB 为一个字向量，其初值为 1,1,?,?,…，且 LENGTH FIB=102，SIZE FIB=204，当作字节时，其值为 1,0,1,0,?,?,…

例 4： ALT DB 50 DUP (0,1)

ALT 是 0,1 的 50 次重复，即 0,1,0,1,……，LENGTH ALT=100。

例 5： 对于字符串来讲，SIZE 运算符特别有用

```
S1  DB  size S1-1, 'this string has 29 characters'
S2  DB  size (S2)-1, '12345678901234567890123456789O', '32'
```

可见，利用 SIZE 运算符，可以在字符串之前，自动地对串中的各个字节实现初始化处理。Size S1 等于 30，故 Size S1-1 为 29；Size S2 等于 33，故 Size S2-1 等于 32。

总之，“LENGTH 名字”是按照名字的组织单位计算该块的长度，它在诸如循环控制之类的应用中是最为有用的。然而，有时我们希望根据某些标准单位(常用字节)来计算块长，这时就以用 SIZE 为好。

§6.2.3 记录定义

记录仅仅可以用在宏指令代码当中，而宏指令代码是 §6.5 讨论的内容，我们之所以在这里讨论记录，是因为它们在宏指令代码中要分配存贮器。一个记录就是一个经定义的映象或样板，在经过定义之后，就可以以规定的格式很方便地分配和初始化存贮器。样板本身并不占用存贮器，而只有在用其名字作为指令的操作内容时，才根据样板的定义，为它分配存贮器。

指示符 RECORD 的格式为：

```
名字 RECORD  字段 1, 字段 2,…
```

其中，字段名的格式为：

```
字段名:长度表达式[=其他表达式]
```

这样的一个说明，就将“名字”定义为一个记录，根据整个定义中的位数，决定将这些字段压缩成一个字节或一个字。

每个字段的定义是这样的：给出该字段名字的编码、一个冒号、以及该字段所占位数的一个表达式。仅有的一个可供选择的参数是“=其他表达式”，它可以接在字段长度的后面说明，以提供一个缺省的初始值。倘若给出的初始值太大，就报告出错。

一个记录中所包含的最大位数是 16，最小的记录为一位。记录运算符 WIDTH 以 2 进制位为单位给出该记录的宽度，即各字段长度表达式值的总和。记录的 SIZE 被定义为为了存放该记录所需要的字节数目，即

```
SIZE EXAMPLE_REC=
    1; 如果 WIDTH EXAMPLE_REC 为 1 到 8 位
    2; 如果 WIDTH EXAMPLE_REC 为 9 到 16 位
```

一旦经过定义，就可以利用记录“名字”去分配和初始化存贮器，其方法类似于 DB、DW 或 DD 所使用的方法，下面专作讨论。

在表达式或指令当中，每个“字段名”都可以用作向右调整所需要的移位计数，“MASK 字段名”得到的是为了访问该字段（按其原位置）所需要的分离字，为了便于理解，请看下例：

```
HASH_ENT  RECORD  FREE:1,  EMPTY:1,  INDEX:14
```

上面这个 HASH_ENT 的记录说明将产生如下的符号值：

```
FREE=15      MASK  FREE=8000H
EMPTY=14     MASK  EMPTY=4000H
INDEX=0      MASK  INDEX=3FFFH
```

假设在后面的指令当中用到这些字段，为了向右调整这些字段，就得利用左边的这些值作为移位计数值。右边的值是分离字，它们是抽取或测试这些字段所需要的，在逻辑与 (AND) 或 TEST 指令中尤为有用。注意 WIDTH HASH_ENT=16，SIZE HASH_ENT=2。

为了把 VAR_ONE 的 EMPTY 字段，送到 VAR_TWO 的 EMPTY 字段中去，可以使用下面的指令序列：

```
MOV AX, VAR_ONE              ;把位于 AR_ONE 的那个字送到累加器
AND AX, MASK EMPTY           ;(AX) ∧ 0100000000000000⇒AX
MOV BX, VAR_TWO
AND BX, NOT MASK EMPTY       ;清除 VAR_TWO 的 EMPTY 字段;
OR AX, BX                    ;获得新的 EMPTY 字段值
MOV VAR_TWO, AX              ;VAR_TWO⇐(AX)
```

利用

```
TEST VAR_ONE, MASK EMPTY
```

可以检查 EMPTY 字段的内容，它根据 VAR_ONE 的 EMPTY 字段的内容设置 0 标志位 (ZF)，即若 EMPTY 字段的内容为 0，则把 ZF 置为 1。

运算符 SIZE、WIDTH、MASK 加上字段各移位计数的自动定义提供了一些有力的工具，当然这些还不是一下子就看得出来的。在没有显式给出各种特定值（如存放该记录的字节数、各字段的位置或宽度以及整个记录的宽度）的情况下，利用这些工具，就可以对记录实施操作。

正象在 FREE、MASK EMPTY 及 SIZE HASH_ENT 中那样，我们并未用数，而是用名字或“运算符　名字”，这样，就不需要程序员去计算所用记录各字段的宽度、移位或分离字。此外，由于这些宽度 (sizes)、移位数 (shifts) 或分离字 (masks)，不是作为定值出现的，而是根据记录定义的操作得到的，所以，对于记录内容及结构需要经常改变的那些应用来讲，使用这些工具就更有效了。

所有这些，都使得通用序列或过程的构造容易多了，对于那些处理方法类似而处理的数据差别很大的过程来讲，构造通用过程就变得相当重要，因为这时我们可以不加修改地将这些过程运用到不同的应用当中去。

§6.3 汇编命令

本节讨论汇编命令，汇编命令是在产生目标代码时，用来控制 8086 汇编程序的指示符。

一般来说，汇编命令具有与指令相同的格式，可把它们分类如下：

*地址计数和分段控制

SEGMENT/ ENDS

ORG

GROUP

ASSUME

PROC/ENDP

LABLE

*符号定义

EQU

PURGE

*程序连接

NAME

PUBLIC

EXTRN

END

*存贮器保留与数据定义

DB (已在前一节当中讨论过)

DW 〃

DD 〃

RECORD 〃

§6.3.1 段的定义：Segment 和 Ends 命令

任一指令或变量，都包含在某个称为段的单元块中。利用命令 Segment，就可以建立一个段及段名，即

name 1 SEGMENT [定位类型][联系类型] [分类名]

在这条 SEGMENT命令之后，所有指令或数据(嵌套段除外)，都被认为位于“name 1”段当中，直到遇到伪指令

name 1 ENDS

它表示至此结束段“name 1”的定义，需要注意的是这两个指示符中的名字(name 1) 必须相同，跟在 SEGMENT 后面的那些参数必须按照给定的顺序排定。嵌套在该段中的段与此段是不相干的，也就是说，它们的指令及数据只在其局部段中，而不包含在外层段中。

由上可见，该命令中有三个可供选择的参数，利用它们来确定该段的属性：

1. 定位类型，它有 5 种不同的选择

2. 联系类型，它也有 5 种选择

3. 分类名(classname) 是任意选择的一个名字，最长为 40 个字符，它必须用单引号括起来。除非对 LOC86 (或 QRL86) 施加更为严格的控制，否则，将把同名的那些段分在一块存贮器区域当中。

如果给出的参数不止一个，那末它们就应该按照上面给出的先后次序出现。

一、定位选择：PARA, BYTE, WORD, PAGE, INPAGE

定位选择，实质上是一个定位命令。为了按照规定方式定位段，连接程序及定位程序就需要一定的信息，而汇编程序正是利用定位命令来提供这类信息的。

5 种定位方式，为我们提供了规定段的界地址的手段，象所有的地址那样，这里的界地址也具有两个部分——一个段号及一个偏移量，段号就是段名的值。

认识到每个地址由一对数值确定是很有用处的，如这对数为 1234H，0056H，它就意味着地址 12396H，即先把段号左移 4 位（一个 16 进制位），然后加上偏移量 0056 H，有

12340H＋0056H＝12396H

对汇编程序来讲，段名就代表段号，它是段的起点，即它的偏移量为0，从这个意义上讲，段名有时还意味着段的范围或内容，它可以向高地址延伸任意字节（0 到 64K-1 个字节中的任一数值）。例如，我们可以讲把某个段送到一个段寄存器当中去，其意义就是传送该段始址（段号）；我们也可以问某个变量是否在该段中，以此表示它是否为该段内容的某个部分。

在汇编语言源程序中，某段中的所有地址都是对应段的始址的相对地址。各段起点处的相对地址为 0，从而使得段内标号或变量的定义的相对地址为 0，1，2，…

在源程序代码中，必须先把段的始点送至某个段寄存器，例：

```
MOV AX, SEGNAM44
MOV ES, AX
```

然后，8086 硬件，就能够自动地为程序中使用的变量或标号，形成完整的正确地址。

然而，当 LOCATE 将程序模块装入存贮器的时候，就要从绝对地址中，减去某个数值以得到相对地址，由于段长可变，故段的终点的位置也是不确定的。

LOCATE 程序建立段号及段起点的位移量（为了充分利用存贮空间，有时得到的段始址可能不是16的倍数，这时就需要此位移量来对地址作修正），该段中的所有地址引用都要将段的位移加到各单元的偏移量上去，从而形成相对于段号的相对地址。

缺省的定位类型是 PARA，它表示段的始址能被 16 除尽，即其 16 进制数的最后一位为 0，这时，段的初始偏移量为 0。

PAGE 意味着界地址的最后两个 16 进制位为 0(00)，例 76500H，表示段号本身以 0 结尾，与 PARA 一样，偏移量的最后一个 16 进制位也为 0。

BYTE 表示任何偏移量都是可接收的，WORD 意味着一个偶数的偏移量（段号的最低位＝0），从而使得段的起始地址为偶数。从偶数地址访问字，仅需要一个存贮器周期，但若为奇数地址，这样一次访问就需要两个存贮器周期。假如段中的字变量都具有偶数的界地址，即 16 进制地址的末位为 0，2，4，8，A，C，E，那末每访问这样一个变量，只需花费一个存贮周期。

INPAGE 表示必须把整个段定位在一页的界地址与下一页的界地址之间，如 56700H 和 56800H 之间，而绝不允许它覆盖一页界地址，因此其规模不能超过 256 个字节。在通常情况下，这种说明仅仅与转换某些类型的程序有关，这些程序是用 8080/8085 汇编语言，为早期的 INTEL 微机所设计的。

若省略定位类型，就表示它为 PARA，但不能用其他任何名字来规定定位类型，否则出错。

二、联系类型：(NONE), PUBLIC, COMMON, AT 表达式, STACK, MEMORY

SEGMENT 命令中的联系类型给出这种信息：当将该段连到及定位到绝对地址时，如何

把它与其他段联系起来。

若没有指出联系类型，那末就不执行联系动作，表示该段是局部于此模块或程序的。如果规定一个联系类型，它就应该出现在该段的首次定义当中，而在此段的后面一些 SEGMENT 命令当中就可以省略不写它，但不可以与第一次说明相矛盾。比如说，假设在该段的第一个 SEGMENT 命令当中没给出联系类型，那就自动作缺省处理，这样后面的对于该段的 SEGMENT 命令，也只可以规定缺省的联系类型，而不允许给出任何其他的显式联系类型。

对于联系类型，可作5种选择，讨论如下：

1. PUBLIC

若规定联系类型为 PUBLIC，那末，就把该段和在与其他模块连接期间所碰到的其他一些同名段连结起来，即最终所有这些段都将是连续的，其顺序不受该命令的影响，但要受到重定位及连接(R&L)命令或R&L缺省的制约，R&L 命令要用到正在连接中的文件的顺序。

2. COMMON

规定 COMMON 会使该段与别的模块中的所有其他同名段共享相同的存贮空间，这就意味着不同的标号或变量名(来自不同的模块之中)能够作用于相同的地址。

例如，在程序的一个模块(MODULE1)中，说明了这样一个段：

```
GLOBAL__DATA  SEGMENT  COMMON
PARAM1  DB  34H
ASSOC1  DB  82H
PARAM2  DB  61H
ASSOC2  DB  75H
GLOBAL__DATA  ENDS
```

然后，在另一个模块(MODULE)当中，又定义 GLOBAL__DATA 如下：

```
GLOBAL__DATA  SEGMENT  COMMON
ITEM1  DW        ?
ITEM2  DW        ?
GLOBAL__DATA  ENDS
```

它们所属的两个模块是分开汇编的。分配给 GLOBAL__DATA 的存贮空间正好是4个字节，但如何访问它们就要依赖于引用的是哪个名字，除非使用 PUBLIC 和 EXTRN 命令，否则名字只在其所属的模块当中才有可能被识别。

正象所期望的那样，MODULE1 中的 PARAM1 或 ASSOC2 的传送(MOV) 得到的是所需的字节，分别为 34H 或 75H。然而，MODULE2 中对 ITEM2 的引用，将取得一个整字 7561H。

3. AT 表达式

利用“AT 表达式”，汇编程序就把表达式的值当做段号，表示该段的绝对地址的数对将成为〈段号,0〉。例如，若写出 AT 1234H，则该段将被定位在绝对地址 12340H 处。若需要段从绝对地址 12345H 处开始，那末此 SEGMENT 命令后面一行必须是 ORG 5。

4. STACK

这种联系类型是与 8080 的 STKLN 命令相关联的，它利用覆盖方式而不是连结方式，将该段与在别的模块中的其他同名段联系起来，也就是说，不是一段接在上一段的后面，而是所

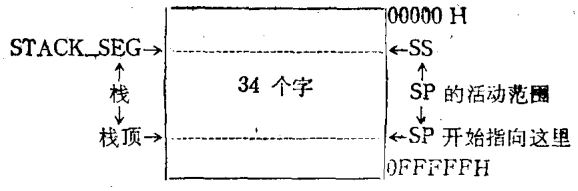
有这些同名段都从同一基址开始、栈段的覆盖是沿着存贮器的高地址进行的。例如，假设在一个模块中定义：

```
STACK_SEG  SEGMENT  STACK
           DW  20  DUP (?)
STACK_TOP  LABEL  WORD
STACK_SEG  ENDS
```

而在另一模块中又有：

```
STACK_SEG  SEGMENT  STACK
           DW  14  DUP (?)
TOP_STACK  LABLE  WORD
STACK_SEG  ENDS
```

定位程序将把这两个 STACK_SEG 段结合成为：



注意：结合起来的段长为 34 个字，即各部分的长度之和，但是栈顶变量 "STACK_TOP" 和 "TOP_STACK"（其名字局部于各模块）被赋与相同的偏移量——存贮器高端 (68D=44H 个字节)。利用下述指令，就可以设置栈段基址和栈指针(模块 1 中)：

```
MOV  AX, STACK_SEG
MOV  SS, AX
MOV  SP, STACK_TOP
```

栈是一种特殊的段，它主要用作暂时存贮器，利用 PUSH、POP、CALL 及 RET 来访问它，多用于过程等的参数传递及返回地址的保护与恢复，其工作方式是后进先出(LIFO)。

在执行过程中，栈是随着栈的增大向下增长的，即从存贮器的高地址到低地址。其用法类似于一迭碟子，PUSH SI 将 SI 当中的字送至栈顶，POP DI 则弹出栈顶的内容且送到 DI 当中去。

为连结栈段保留的存贮器，是各段所需单元的和，这是由于在某个模块装满了其栈空间而没释放时，有可能另一模块又希望使用栈。

5. MEMORY

MEMORY 的作用与 COMMON 类似，但经这样定义的段，被定位在存贮器中的所有段的上面。

在每个被连到一起的模块组中，只应该有一个段具有这种联系类型，因为仅可能把 R&L 遇到的第一个段解释成 MEMORY。因此，假如 MEMSEG 是 R&L 碰到的第一个具有联系类型 MEMORY 的段，那末它以及其他模块中名为 MEMSEG 的任何段，都将被定位在所有其它段之上。而把以 MEMORY 作为其联系类型的其他段当作 COMMON 处理，也不要将它们定位在所有其他段之上，这时 LOCATE 程序将发出警告。

应该注意到，尽管上面把 STACK 和 MEMORY 用作关键字，但仍然可以把它们当作用户

变量名字。在用早期的 Intel 汇编语言编写的程序中，STACK 表示栈的第一个单元的地址，MEMORY 表示程序末端之后的第一个存贮器字节地址。

在用 ENDS 命令封闭了段名为“名字1”的段之后，假如再次使用段名“名字 1”，那末写出的代码或数据，将自动地接在前面在段“名字 1”中写出的那些行之后。例如：

```
ROUTINE_FIRST    SEGMENT
                   .
                   .                ;这里的码子在段 ROUTINE_FIRST中
                   .
            A:     .
ROUTINE_FIRST    ENDS
  OTHER_ROUTINES    SEGMENT
                      .
                      .
                      .             ;这里的码子位于段OTHER_ROUTINES
                      .             ;当中
  OTHER_ROUTINES    ENDS
  ROUTINE_FIRST     SEGMENT
                      .
                      .             ;这里的码子位于段 ROUTINE_FIRST中
                      .             ;且跟在标号A的后面。
                      .
  ROUTINE_FIRST     ENDS
```

还可以这样来实现段的连结：

```
PROCESS_1    SEGMENT
               .
               .                    ;这里的代码位于段 PROCESS_1 当中
               .
  PROC1DATA  SEGMENT
               .
               .                    ;这里的代码位于段 PROC1DATA 当中
               .
               .
  PROC1DATA  ENDS
               .
               .                    ;这里的代码位于段 PROCESS_1 当中,且
               .                    ;直接跟在由 PROC1DATA 的定义打断的
               .                    ;PROCESS_1 块的最后一行的后面
               .
  PROCESS_1  ENDS
```

嵌套段必须在外层段结束之前结束，因此，如果语句

```
PROC1DATA  ENDS
```

在命令

```
PROCESS_1  ENDS
```

之后出现，那末就指出这次段的覆盖出现错误。

对于那些出现在 SEGMENT/ENDS 对之外的代码，汇编程序将自动地把它们送到一个名为“??SEG”的缺省段中。

当把段名用在其他汇编语言指令当中时，如：

```
MOV  AX,  Segnam
MOV  ES, AX
```

这个名字将自动地获得该段始点的段号值，如果一个同名段出现在待连接的另一模块中，后面就将把这些段连结起来。

§6.3.2 ORG 指示符

汇编程序的位置指针，在汇编期间所起的作用类似于执行期间的指令指针，具体讲，它告诉汇编程序下一个可分配给指令或数据的存贮器单元。

第一次出现伪指令

```
名字1  SEGMENT
```

时，它定义段“名字 1”的起点，建立一个新的位置指针且把该指针清 0。以后每分配一个字节，一般都将这个位置指针自动加 1。此段的一个 ENDS 指示符将冻结该指针，直到再次“打开”此段，再次“打开”表示后面又出现了伪指令“名字 1 SEGMENT”，这时就需要把位置指针用来继续汇编过程，位置指针又开始对分配的字节进行计数，不过这时的计数将从上次获得的数目开始。

利用指示符 ORG (origin)，能够改变当前的活动位置指针的内容。

前已提及，在所有用户定义段之外产生的码子，都放到一个名为 ??SEG 的特殊的汇编语言定义段当中，??SEG 的联系类型为 PUBLIC，用户定义段之外的 ORG 语句，就作用于该段内的偏移量。

ORG 命令把当前被定义段的位置指针设置成由操作数表达式所确定的值，其格式为：

```
操作码     操作数
ORG        表达式
```

必须注意，该命令不可以带标号，例

```
SWITCH: ORG 14
```

是无效的。

操作数表达式的值不可以取负数，它可以在当前段内取一个绝对编号或可重定位编号。汇编期间 ORG 表达式的赋值，总要进行一个模除 64K 地址的操作，从而保证位置指针在 0 到 65,535 的范围内取值。表达式中的所有符号，都必须是前面已经定义过的。ORG 的下一条指令或数据项就将在指定地址处进行汇编。

在大多数模块当中，都不需要伪指令 ORG。如果在某段中的第一个指令或数据字节之前

不包含 ORG 指示符，那末汇编就从相对于该段始址的 0 位置处开始。

程序中可包含的 ORG 命令的数目是不受限制的，多个 ORG 也不需要以递增的序列来确定地址，但应该注意，倘若不这样，就要冒这样的风险：汇编程序已在某些地址处，为前面的某个程序部分汇编过了，而现在又要为同样的地址产生第二块代码，因此，在把它们装入时，其中一块将复写到另一块的上面。

例：设位置计数器的当前值是 0FH (10进制的 15)，这时碰到一个

ORG 指示符：

```
ORG   0FFH        ;令汇编程序从 0FFH (255D) 位置处开始汇编
```

那末就将在位置 0FFH 处，汇编下一条指令或数据字节。

§6.3.3 Group 定义(成组定义)

这条伪指令，通知汇编程序，把在操作数表中列出的那些段，装至存贮器的同一个 64K 空间中，同时还给该组一个名字，组名的使用方式与段名的用法相类似。分组的一大优点，就是可以得到比较紧凑的代码，这样，组内的跳转也仅仅需要 16 位的跳转地址，哪怕这种跳转是段间跳转。由于待成组的各段之间的关系不一(不相关的、有联系的等)，故汇编程序就无法检查这些段是否能够装到 64K当中，但它却让重新定位与连接 (R&L) 程序来完成这一检查。若组长超过 64K，则表示出错。本命令不影响 LOC86 (或 QRL86) 对这些段的定位。

伪指令 GROUP 的格式是：

```
名字   GROUP     段 1,段 2,…
```

其中，段 1、段 2 等操作数既可以是 SEGMENT 命令的名字字段，也可以是形如 "SEG 变量名"或"SEG 标号名"的表达式，这种表达式返回的是定义上面的变量名或标号名的段号，它对于向前引用或外部名字是特别有用的。

例：

```
CODE_SET   GROUP I_O_ROUTINES, INIT_PROC, SEG FIRST_RECS
DATA_SET   GROUP INVOICE_REC, ACCT_REC, GEN_LDGR_DATA
```

在把某个组名装到一个段寄存器当中，且在邻近的 ASSUME 语句中给出此信息之后，就能够利用这一个段寄存器访问组中所有段的符号，使用的偏移量为自组基地址开始到各符号的距离，它一般不同于该符号在其本身段中的原始偏移量，这是由于在它们中间有可能存在一些其他段，这就意味着调试过程中所使用的地址，将来自 LOCATE 的输出，而不是汇编的输出。

相应的 ASSUME 语句为：

```
ASSUME CS: CODE_SET, DS: DATA_SET
```

成组命令中的段的次序，不一定就是它们在存贮器中的次序。某个给定的段名，还可以出现在多个 GROUP 命令当中，不存在这样一种自动的方法，它能保证满足由各种 ASSUME 和 GROUP 命令所给出的所有限制条件。但此时要注意多条命令的一致性。还需注意，这些组是不可以向前引用的。

§6.3.4 Assume 命令

Assume (设定) 命令的格式是：

ASSUME segreg：segnam [，segreg：segnam，…]

(ASSUME 段寄存器：段名[，段寄存器：段名，…])

或

ASSUME NOTHING

其中，段寄存器 (segreg) 为 4 个段寄存器 (DS，ES，CS，SS) 中的任意一个。段名(segnam) 可以是任何段名、任何前已定义过的组名、表达式“SEG变量名”或“SEG 标号名”、以及关键字 NOTHING。

例：

```
ASSUME  DS:DATAWORDS_SEGMENT_NAME,
&       ES:STRING_SEGMENT_NAME,
&       SS:NOTHING, CS:CODE_SEG_NAME
```

必须注意，本命令不可以带标号，故

```
CASEI:  ASSUME CS:S4
```

是非法的。

通知汇编程序执行时的环境，是相当重要的，所谓执行时的环境，是指所产生的指令将在此环境中运行，这个环境由 4 个段寄存器的内容组成，ASSUME 指示符将告诉汇编程序应在这些寄存器中装入的地址，汇编程序也正是利用这种信息，来检查所引用的变量或标号，是否可以通过这些段寄存器来寻址，假如需要的话，汇编程序还要由此确定为某些变量产生段跨越前缀字节。

汇编程序产生的指令依赖于各段寄存器中的内容，我们知道，每个存贮器地址实际上是一对 16 位的数，即偏移量和段基址，几乎所有访问存贮器的指令，都仅使用偏移量，而期望段基址来自某个段寄存器。假如在汇编时没有对那个段寄存器的运行时内容做设定的话，那末就不可能用给定的偏移量及该段寄存器计算出正确的运行时存贮器地址。因此，在使用这些段寄存器之前，以及在程序中对某个段寄存器进行修改之前，需要一条 ASSUME 命令。

对存贮器或栈的任何访问，都将使用寄存器 DS、ES 或 SS，这些寄存器必须在访问存贮器的代码之前，首先在 ASSUME 中出现。类似地，指令标号 (包括过程名) 隐含地用到 CS 寄存器，所以，CS 必须在任何引用标号的代码之前，先在一条 ASSUME 语句当中出现。倘若在某个模块或段当中，没有用到某个段寄存器，那末就应该为该寄存器设置一条 ASSUME NOTHING 命令。

这条命令对 CS 尤其关键，因为包含指令的那些单元的偏移量总是与 CS 相关的。指令偏移量包含在指令指针 IP 当中，用它来确定待执行的下面一条指令。在大多数情况下，它只有在 CS 包含汇编期间为那条指令设定的段基址时，才是有效的。如果 CS 含有一个不同的值，那末不合要求的结果就很有可能发生，这样就要经过很大的努力才能跟踪指令的执行过程。

因为这个原因，汇编程序要为每个标号及指令跟踪设置好 CS 值，它禁止段内/组内 (NEAR) 跳转的目标为 CS 设定值不同的标号。

但是，这种限制一般不适用于CS内容要变化的跳转，如段间（"long"或FAR）跳转或调用，在CS的内容改变了之后，后面的所有CS的使用，当然也都只用这个新值，在目标标号处，就需要一个新的ASSUME命令，以此来通知汇编程序在执行到该点时，要用一个新值装到段寄存器中去。

伪指令ASSUME，除了告诉汇编程序各段寄存器的运行时内容之外，还可以省掉很多段跨越前缀代码。

在不使用ASSUME的情况下，对存贮器中数据的所有访问，都必须显式给出为了访问数据所需要的段名（作为基址），汇编程序为了能够为机器指令产生前缀字节需要这一信息，举例来讲，如果SOURCE和DEST是在段GLOBAL中定义的，而FILL的定义位于段PARAMS当中，那末为了把SOURCE－DEST＋1送到FILL，可以写下面这段程序：

```
GLOBAL SEGMENT
  SOURCE  DW ?
  DEST    DW ?
GLOBAL ENDS
PARAMS SEGMENT
  FILL    DW ?
PARAMS ENDS
CODE SEGMENT
  ASSUME CS:CODE
MOV  AX, PARAMS
MOV  DS, AX
MOV  AX, GLOBAL
MOV  ES, AX ;following code assumes values were assigned to source and dest
MOV  AX, ES:DEST
SUB  AX, ES:SOURCE
INC  AX
MOV  DS:FILL, AX
 ⋮
 ⋮
CODE ENDS
```

可以看到，利用显式的段前缀，ES指向GLOBAL，DS指向PARAMS。

为了访问这些段中的任何变量，总需要写出"ES:"和"DS:"，同样，在源程序当中，为了访问栈中的数据，要加前缀"SS:"，若要访问代码段中的内容，也要加前缀"CS:"。

然而，利用ASSUME命令，就可以只告诉汇编程序一次为了访问这些变量需要使用哪些段寄存器，而不需要告诉每条指令有关这方面的信息。下面这个程序序列与上面那段程序是等价的

```
ASSUME CS:CODE, DS:PARAMS, ES:GLOBAL
MOV  AX, PARAMS
MOV  DS, AX
```

```
MOV  AX, GLOBAL
MOV  ES, AX
MOV  AX, DEST
SUB  AX, SOURCE
INC  AX
MOV  FILL,AX
```

正如前面指出的那样，在每个改变段寄存器的指令块前面，都需要一个 ASSUME 命令，其原因就在于执行期间的控制流程，可能与汇编程序在汇编时所见到的指令顺序很不相同。

利用适当的 ASSUME 命令之后，程序员就不必考虑段跨越前缀字节的需要与否，因为这些工作将由汇编程序自动完成。

假设需要访问这种变量，既没有用 ASSUME 将所需的段基址送到某个段寄存器，又没有显式地给出前缀，那末汇编程序就给出一个错误标志。

每次访问数据项(变量)或标号(指令代码)时，都要检查其初始的段说明以及当时设定的段寄存器中的内容。假如包含该访问的那个段被设定(ASSUMED)到任何段寄存器当中的话，汇编程序就将为该访问产生正确的指令，根据需要，它还可以包含一个段跨越前缀字节。汇编程序究竟选择哪个段寄存器呢？这要依赖于其地址表达式的类型，即可以通过它所使用的变量及下标(即指针与/或索引寄存器) 来确定某个段寄存器。倘若某个变量出现在地址表达式当中，就一定存在一个包含该变量的段基址的段寄存器，如果需要的话，它就是用作前缀的那个段寄存器；假如地址表达式当中只用到寄存器，那末由 MODRM 表就可知道应用哪个段寄存器(大多数情况用 DS，对于地址表达式中包含 BP 者应用 SS)。

我们可以这样来理解整个检查过程：

a) 哪个段包含该引用？

b) 该段(或包含该段的某个组)是否已被设定到某个段寄存器当中，即送到 SS、CS、DS 或 ES 中？

c) 若回答为“否”(没被设定)，那末除非该数据引用带有一个显式编码的段跨越前缀字节，否则报告出一个错误。

d) 若回答为“是”(设定过了)，问该段寄存器是否为相应的硬件指令所常用的那一个？

e) 若所设定的寄存器是通常的那个可省去者，则产生一般形式的代码 (无段跨越前缀字节)。

f) 若不是这样，则看是否可加前缀或是否要加前缀？

g) 若可加前缀，则先产生一个前缀字节，以便使用正确的段寄存器，然后产生一般形式的代码。

h) 否则，报告出错。

下面来看看 ASSUME NOTHING，它将消去前面对段寄存器的所有设定，这就需要程序员为每个操作数都应显式给出段跨越前缀字节。倘若不提供这种前缀字节，汇编程序就无法确定可从哪个段寄存器出发来寻址该变量，因而给出一个出错信息。因为若汇编程序不适当地使用了某些段寄存器来访问数据，那就有可能造成混乱，得出错误的结果。

例如，大多数访问变量的硬件指令，都隐含地期待着将 DS 寄存器用作基址寄存器：

```
MOV  AX, DATAWORD
```

```
MOV  CX, ARRAY [SI]
MOV  DX, MATRIX [BX+7] [SI]
```

除非程序员给出一个段跨越前缀字节，否则8086的硬件，都将期望这些伪指令使用DS，而汇编程序却不能保证可从DS寄存器出发访问当前的所有数据。在未加段跨越前缀字节的情况下，汇编程序必须从ASSUME出发，来检查和利用所设定的段寄存器。如果不存在ASSUME语句，或者在利用了ASSUME NOTHING的情况下，就必须为它们显式编码出段跨越前缀字节，甚至对于DS来讲也是如此。

因此，在ASSUME NOTHING的条件下，上面这些指令的正确代码就变为：

```
ASSUME NOTHING
MOV    AX, DS:DATAWORD
MOV    CX, DS:ARRAY [SI]
MOV    DX, DS:MATRIX [BX+7] [SI]
```

当然，首先应把BX和SI设置成正确的值，以使它们指向数据段中所需要的元素。

再举个例子，设已知DATAWORD位于基址在SS的段中，其他变元位于基址在ES中的段里面，这就要适当地利用一个ASSUME语句，告诉汇编程序这些已知信息。否则，若利用了ASSUME NOTHING，指令就变为：

```
MOV  AX, SS:DATAWORD
MOV  CX, ES:ARRAY [SI]
MOV  DX, ES:MATRIX [BX+7] [SI]
```

对于串指令，其解释又有所不同。若指令为MOVS，则源操作数一般在DS段中，而目标操作数则一定在ES段中。因此，指令序列：

```
ASSUME DS:SOURCE_STRING_SEGMENT,
       ES:DEST_STRING_SEGMENT
MOV DI, DEST_STRING_INDEX
MOV SI, SOURCE_STRING_INDEX
STD      ;它设置方向标志
MOVS  DEST_ STRING, SOURCE_STRING
```

将把一个字节(或字)从DS送到ES中，且把索引指针DI和SI加1(或加2)。

假设源串实际是在栈段当中，这就需要一个段跨越前缀字节：

```
MOVS  DEST_STRING, SS:SOURCE_STRING
```

在ASSUME NOTHING的情况下，不能省略该SS前缀字节，否则，汇编程序就将发出一个出错信息。但假如SS是包含源串的那个段，且前面有ASSUME：

```
ASSUME SS:SOURCE_STRING_SEG
```

那末SS段跨越前缀，则由汇编程序自动产生。

在一条指令当中，至多只能允许出现一个段跨越前缀字节。

如果显式给出段跨越前缀字节，如：

```
MOV AL, ES:DATABYTE
```

那末汇编程序就照此办理，而不检查它是否有意义，也就是说，汇编程序将产生该指令以及给出的前缀字节，而不检查从段寄存器ES出发是否真的可访问该数据。

例：

硬件期望 BP 为栈段中的一个指针，即希望把 BP 用作栈段 SS 中的内容的偏移量。然而，也允许把 BP 用作数据段中的一个指针，只不过这时需要一个前缀字节(DS:)。

下面这段代码，将使汇编程序产生所需要的前缀：

```
ASSUME  DS:SEG ARRAY
    .
    .
MOV AX, ARRAY[BP]
    .
    .
```

为了清楚地反应这种用法，便于今后偶然的阅读与理解，最好在所有这种指令当中，显式地加上前缀字节，即写成：

```
MOV AX, DS:ARRAY[BP]
```

§6.3.5 标号的定义

在较为一般的意义上讲，标号就是存贮器中某些特定单元的名字，这些单元的内容可以是指令，也可是数据。但在本书当中，变量和标号之间是有区别的，即变量代表数据，标号代表指令。NEAR 和 FAR 是标号的两种类型，而 BYTE、WORD、DWORD 是变量的类型。

指示符 LABEL 能为任何单元建立一个名字，而不管该单元的内容或特定用途如何，并且还给该名字一个类型。类型决定了该名字的合法用途。

如果类型为 NEAR 或FAR，那末该名字就代表一个标号，它可以用在跳转或调用指令中，但却不可用于 MOV 或其他数据操作指令当中。而如果类型为 BYTE、WORD、DWORD 或某些其他变量，那就表示该名字是一个变量，在 MOV 这类指令当中，它是有效的，但是绝不可把它直接用在跳转或调用指令当中。

不管它是一个数据还是一个指令，LABEL 命令都将为汇编的当前单元建立一个名字。命令格式为：

```
名字 LABEL 类型
```

其中，“类型”有 5 种选择：BYTE，WORD，DWORD，NEAR，FAR。

例：

```
PUFF LABEL BYTE
     DB 21
```

它等价于

```
PUFF DB 21
```

它们都把当前的汇编地址命名为 PUFF 且把它规定为一个字节变量，初始值为 21。

在同一行命名指令需要一个冒号，如：

```
TRANS: MOV AX, CX
```

它等价于

```
TRANS LABEL NEAR
```

```
MOV AX, CX
```

只有当正在汇编的段，被设定在 CS 寄存器可达的范围之内时，才允许定义标号（只用于指令码，不表示变量），也就是说，必须提供一个 "ASSUME CS：名字"语句，这里的"名字"为该段本身的名字，或是包含该段的某个组名。如果设定的是一个组名的话，那末标号的偏移量将从该组的基址算起，而标号的段号就是该组的始址。"名字：" 与 "名字 LABEL NEAR"具有相同的意义。

但是，这种"名字："结构，不可以与存贮器初始化出现在同一行中，即 "ITEM：DW 0" 是不合法的。去掉其中的冒号，就表示把 ITEM 定义为一个初值为 0 的字变量。假如需要把 ITEM 作为标号而不是变量，就可以改写为：

```
ITEM：
    DW 0
```

或者写成：

```
ITEM LABEL NEAR
    DW 0
```

LABEL 命令中"名字"的值，包括当前的段分量及偏移量，即定义该地址的一对数值，此外，"名字"还带有类型属性，这里属性为"NEAR"。

这条命令常用来给某个单元加上第二个名字，使之可在不使用 PTR 运算符的情况下，以不同的方式访问这个单元。例如，假设希望把一个相同的存贮器区域作为一个字节向量和一个字向量，就可这样来定义向量：

```
VECTORB LABEL BYTE
VECTOR DW 1000 DUP (0)
```

将来要想把它当作字节向量使用时，可这样

```
ADD AL, VECTORB
```

当作字向量使用时，用这样的指令：

```
ADD AX, VECTOR
```

利用 PTR 运算符，同样可以达到这一目的：

```
ADD AL, BYTE PTR VECTOR
```

下面这个问题只与代码有关：问题是这样的，我们有可能希望改变标号的距离属性，希望某个标号既可以在其段内被引用，也可以从该段的外面（还可能从它所属组的外面）来引用。所有的外部引用，都要求把其属性规定为 FAR，而对于局部引用来说，因为只需要偏移量的值，而不需要段值，为了节省这一个字，就在该段内定义了一个距离属性 NEAR 的同义词。假设有这样一个指令序列：

```
PROCESS_ITEMS LABEL FAR
LOCAL_NAME：MOV AX, FIRST_ITEM  ；任何指令
                .
                .
                .
```

利用 EQU，也能够达到这个目的，因为在汇编当中，下面两个语句是相同的：

```
名字  LABEL X
```

名字 EQU THIS X

运算符THIS，把当前的段分量——偏移量对以及由"X"确定的属性分量送到"名字"当中，"X"必须是上述的某种类型。

因此，可把上面的例子写成：

```
PROCESS_ITEMS EQU THIS FAR
LOCAL_NAME: MOV AX, FIRST_ITEM      ;任何指令
```

§6.3.6 过程的定义

所谓过程，就是一段 ASM86 代码，它可以在程序的其他部分被激活，就好象它们在这些地方顺序地出现一样。CALL 语句激活过程，使机器转去执行过程代码，即程序的控制将从激活点转到过程码的起点，且从这一点开始执行过程的代码。当遇到一条返回 (RET) 指令时，就从该过程返回，程序的控制又返回到激活点的下一条指令。在需要的地方都要加上 RET，在一个过程当中，可能有多个返回点，所以返回不一定在过程的终点。

使用过程具有下列优点：

1. 它是模块程序设计的基础；
2. 利用过程可以建立和使用程序库；
3. 为程序设计和文本化提供了方便；
4. 减少了程序产生的目标代码的数量。

下面将讨论过程说明及激活所采用的方法。

```
名字    PROC    NEAR | FAR
          .
          .
          .
        RET
          .
          .
          .
名字    ENDP
```

这组命令给代码序列的入口加上了一个标号，并且指明该过程是 NEAR 或 FAR 型，如果不写出类型信息，就把它当 NEAR 使用。在这对命令当中，采用的名字必须相同，在汇编语言的程序当中，这两个命令必须是成对出现的。

PROC 说明几乎与 LABEL 相同，但是，若要使用调用或返回指令，就只能使用 PROC/ENDP 这对命令。当调用一个过程时，在控制转移到过程之前，先要把下一条顺序指令的地址送到栈中保护起来，以便过程返回。

如果过程的类型为 NEAR，返回只要将栈顶的一个字弹出，并送至指令指示器 (IP)。如果过程的类型为 FAR，则从该过程返回时，就要先从栈顶弹出一个字，送到 IP 中，然后再弹出一个字，并送至 CS 中，完成返回动作。

如果某个过程需要有多个入口，就可以利用 LABEL 命令来实现，但这些入口的类型必

须相同(NEAR 或 FAR)，否则就无法按需要返回。出现在 PROC/ENDP 对之外的返回指令，都被当作类型为 NEAR 的返回。

与符号表中几乎所有的名字一样，过程名也可以被清除，这将从符号表中删除该名字的定义，也就为把该名字用于新的目的提供了可能性。

必须注意，利用跳转指令跳出某个过程时，其返回地址(一个或两个字)仍然留在栈上，所以在后面使用栈的时候，必须把这一因素考虑在内。

RET 在物理上，不需要是 PROC 源程序中的最后一条指令，但是，假如过程执行到 ENDP，而它并没有自动的或隐含的返回(RET)作用，这时就返回不成了，因此，需要注意把控制转移到 RET。

过程和段可以相互嵌套，即过程可以完全地包含某个段，而段也可以完全地包含某个过程。但它们不可以交叉覆盖，即内层过程或段，一定要在外层过程或段结束之前结束。例如，这个序列是有效的：

```
STARTER  SEGMENT
FIRST    PROC NEAR
FIR_1;    .
          .
          .
FIR_3;    .
          NEXT PROC NEAR
     NEXT1;    .
          MID SEGMENT
                .
                .
          MID ENDS
     NEXT44;  RET
          NEXT ENDP
FIR-77;   .
          .
          .
          RET
FIRST     ENDP
STARTER  ENDS
```

而下面这个序列是无效的：

```
STARTER    SEGMENT
FIRST        PROC NEAR
                .
                .
                .
             NEXT PROC NEAR
```

```
                    .
                    .
                    .
                  RET
FIRST     ENDP
STARTER   ENDS
                    .
                    .
                    .
                  RET
NEXT      ENDP
```

过程可在其出现的地方，或被调用的地方执行.如在执行指令"CALL FIRST"之后，将执行 FIR_1 处的指令，假定在 FIRST 当中，没有跳转或调用指令，所以后面就要执行到 FIR_3，接下来是 NEX_1。如果 NEX-44 处的指令，不是一条返回指令，那末下面就要执行 FIR_77。之所以强调这一点，是因为 PLM86 的规定有所不同，它规定，用 PLM86 编写的过程，只在被调用时才执行，故所有的嵌套过程都将被跳过，除非它们的名字出现在 CALL 指令当中才被执行。

§6.3.7 EQU

汇编程序自动地把值赋给作为指令标号，或变量名的符号，这个值就是当前汇编行的位置计数器的内容。

此外，利用 EQU 命令，也可以定义其他一些符号，并赋值给它们。由EQU所定义的符号，在汇编期间不能被重新定义，EQU 中需要的名字后面不可以跟冒号。

用 EQU 定义的符号，除非被清除，否则它在程序中都具有意义。

EQU 将'表达式'的值，赋给指定的名字。

```
          操作码     操作数
  名字    EQU      equ_op
```

EQU 把"名字"定义为符号表中另一个名字的同义词，或某个常量。

equ_op 可以是--个数、一个表达式、一个寄存器或某个宏代码名。例：

```
THREE   EQU  3                  (数)
XYZ     EQU  ALPHA [SI]+3       (地址表达式)
COUNT   EQU  CX                 (寄存器)
RADD    EQU  ADDR               (宏代码名)
```

汇编时，EQU 表达式的赋值，总将产生一个模除 64K 的地址，即取值范围为 −64K 到 +64K-1。

例：下面这条 EQU 命令，把名字 ONES 送到符号表中，且把 2 进制数 11111111 赋给 ONES：

```
ONES EQU 0FFH
```

在后面的源程序中，通过在表达式中引用其名字，就可以使用 EQU 赋给它的值：

```
MOV AL, 25 AND ONES
```

利用 EQU 命令，还可以为较为复杂的表达式、或地址表达式取替换名字：

```
A  EQU ARRAY [BX] [SI]
B  EQU(ABM*7+44) AND MASK ONE
SUMMA EQU ARRAY_SUMMER
```

经过这些定义之后，就可以任意地把A和B用作所示表达式的同义语，从而可在复杂的应用当中节省时间，有利于防止编码错误。如果适当地起好名字，借助于这种特征，还可以提高程序的可读程度。例如，在一个交通控制课题中，可以使用名字 TRAFFIC，而不是用 A。

§6.3.8 PURGE 命令

PURGE 命令，允许从符号表中删除某些名字，一旦注销某个名字，就可以用与前面的定义完全不同的方式，来重新定义及使用这个名字，而不会引起汇编程序的混乱或冲突。在注销之后，所有对该名字的使用，都将以最新的对该名字的重新定义为准。在注销某名字之后且还未对之进行重新定义的时候，若企图使用该名字，则给出一个无定义的符号错误标志。

PURGE 的格式为：

```
PURGE 名字1,名字2,…,名字n
```

此命令不带“名字字段”，故

```
ALTER PURGE ADD
```

是无效的，因为加上了名字 ALTER。

§6.3.9 程序连接命令

模块程序设计和重新定位特色，使我们能够对多个独立的模块进行汇编和测试，甚至可以把这许多模块连接起来，当作一个程序来运行。这样带来的一个结果，就是模块之间需要进行通讯，而建立这种通讯机构，正是程序连接命令的任务。

一个模块，可以与其他模块共享其数据地址及指令地址，实际上，能够与其他模块共享的，只是符号表中的那些项，因此，在模块中定义这些项时，必须给它们名字或标号。联系类型为 COMMON 的那些段，共享相同的存贮空间，其他那些可用于别的模块的项目，必须在 PUBLIC 中说明。

在其他需要使用这些项目的模块中，必须用一条 EXTRN 命令来说明这些项。倘若某个模块知道在其他模块中定义的数据或指令的实际地址的话，该模块就可以直接访问这些项目，但这在模块都使用重定位技术的情况下，是不大可能的。然而，利用一个名字，汇编程序就将通过重定位与连接（R&L）程序，提供该名字的地址，这样就获得了在其他模块中说明为 PUBLIC 的数据或指令的访问权。

一、PUBLIC 命令

PUBLIC 命令，使得操作数字段中列出的每个符号，都可由其他模块访问。其格式为

操作码　　操作数
PUBLIC　名字表

注意：本命令不可以带“名字”，即

PUB1 PUBLIC GETLIST

是无效的。

例：

PUBLIC SIN, COS, TAN, SQRT

名字表中的每一项，都必须是个名字，这些名字在程序中的某个地方被赋值为数、变量或标号（包括过程 PROC）。名字表中的各个名字之间，用逗号隔开。在一个程序模块中，同一个名字至多仅可被说明为 PUBLIC 一次。

PUBLIC 命令，可在程序模块内的任何地方出现。

在汇编完成时，如果名字表中的某些项，在符号表中还没有对应的输入，则说明它没被定义，且标志为出错。

二、EXTRN 命令

EXTRN 命令，为汇编程序提供了一张符号表，其中的这些符号要在本模块中用到，但它们却是在其他模块中定义的。对于这些符号，汇编程序要负责与该模块之外建立联系，而不是把它们当作无定义引用的错误。

操作码　　操作数
EXTRN　外部引用 1，外部引用 2，…

“外部引用”的格式为：

名字：类型

操作数表中的每一项，都指定了一个符号，该符号要在本模块中引用，但其定义却在另一个模块当中，EXTRN 中规定的类型，必须与定义中的类型相同。操作数表中各项之间，是用逗号分开的。

其中的“类型”是必要的，它可以是 BYTE，WORD，DWORD，NEAR，FAR 或 ABS 中的某一个，ABS 表示它是一个纯数值而不是某个变量或标号。

注意：该命令也不可以带名字，如

MOJL1 EXTRN V:BYTE

是非法的。

如果在此模块中，也给操作数表中的某个符号定义的话，其影响相当于在某个程序中，把同一符号定义过多次，这是不允许的，汇编程序将给出一个出错标志。

反过来，也不可以使用既没在该模块中定义，又未曾出现在 EXTRN 中的符号，否则，汇编程序将指出这是一个无定义的访问。

尽管 EXTRN 命令可以出现在程序模块内的任何地方，但是为了避免发生向前引用的问题，通常以把它放在靠头上为好。

在一个程序模块中，只可以把某个符号说明成外部的一次，但并不需要在该模块中一定要使用它。但是，不可以把同一个名字同时说明成外部的（EXTRN）及公共的（PUBLIC）。

例：

EXTRN ENTRY: BYTE, ADDRTN:FAR, BEGIN:WORD, NUMBER: ABS

三、NAME 命令

NAME 命令的作用就是为本次汇编所产生的目标模块起了一个名字。格式为：

操作码	操作数
NAME	模块名

NAME 命令，在操作数字段中，需要一个模块名，模块名必须遵守定义符号的有关规则，它可以与具有其他用途的符号相同，在引用模块和确定模块的先后次序时，这些模块名都是不可缺少的。

在程序当中，命令 NAME，至多只能出现一次，它也不可以带有名字字段，故

```
MODNAM NAME RATES
```

是无效的。

例：

```
NAME MAIN
```

模块名可以与具有其他某些特殊用途的符号相同，例如：

```
NAME AX
```

也是有效的。

由上面的讨论可以看出，在进行程序的模块化设计时，利用 NAME 命令，为各模块起好名字，从而为这些模块的连结，打下了基础。

四、汇编程序结束命令 END

END 命令确定源程序的终点，结束汇编程序的一次扫描。END 命令的格式为：

操作码	操作数
END	标号(可供选择)

在每个源程序中，只允许出现一个 END 语句，并且要求该 END 语句，是源程序的最后一个语句。假设它不是最后一条，那末在它后面的所有语句都不起作用，汇编过程中的段及过程标号的匹配、向前引用所需的定义等等要求，都有可能实现不了，从而引起许多语句出错。

如果在 END 语句中，给出标号的话，那末该标号的值，就将被用作程序执行的起始地址。倘若不给出标号，也就不知道起始地址，这时就假定该模块不是主模块(main module)。当要把许多分开的程序模块结合到一起时，其中只有一个模块可以指定程序的始址，这个模块也就是所谓的主模块。

§6.4 表 达 式

ASM 86 允许多种类型的表达式出现在程序当中，它对有效表达式的规则是很精确的，利用这些规则，能够确定所需表达式的有效性。

首先需要给出下面几个定义：

变量：定义为某个单元的名字，该单元的内容为数据，它的定义中不可使用冒号。例如：

SOUP DW 2

SALAD LABEL BYTE

一个变量的有关信息是它的段分量、偏移量和类型分量。

标号：它也被定义为某个单元的名字，但该单元的内容为指令，它的定义常需使用冒号。例如：

ADD_INGREDIENTS: MOV AX, SOUP

但也并非总是如此，例如在

FOO PROC FAR　　和

BAZ LABEL NEAR

当中，FOO 及 BAZ 也都是标号。标号具有 4 种属性，它们是段分量、偏移量、CS 的设定值以及距离分量(即 NEAR 或 FAR)。

数字：定义为一个数字值的名字，而不再是一个存贮器单元。若已经给某个名字一个精　的数值，例如：

DELTA EQU 77 或

B　　EQU 11

那么 DELTA 和 B 就代表数字而不是变量或标号。

在这种汇编语言中，变量和标号两者与数字之间的区别是十分重要的，我们把变量和标号归为一类表达式，并将它们称为地址表达式。地址表达式是值为存贮器地址的一种表达式。与标号一样，变量也是地址表达式。地址表达式有三个分量：

1. 段分量；
2. 偏移量；
3. 类型分量。

这 3 个分量，是每种合法地址表达式的必要组成部分。

另一类表达式，是数字表达式。数字表达式产生的是一个数值结果。3 是一个数字，4＊5 (即 20)也是如此，它们都属于数字表达式的范畴。尽管地址表达式的每个分量，都是数字值，但它本身的值却不是一个数字值。

为了搞清楚地址表达式和数字表达式之间的区别，考虑 OFFSET 这个操作符(更完整的定义在后面给出)：一个地址表达式的 OFFSET，把该表达式的偏移量，作为一个数字值返回。

在下面这个程序中，我们可以使用 OFFSET 算符，来说明地址表达式与数字表达式之间的区别：

```
        ASSUME CS:Code, DS:data
data    SEGMENT AT 55H
        DB 0,1,2                          (三个字节)
e_byte  DB 0FFH
data    ENDS
Code    SEGMENT
start:  MOV AX, data                      (段基址(55H)送 AX,
        MOV DS, AX                        然后送 DS 寄存器)
one:    MOV AL, 3                         (数字 3 送 AL)
```

```
two:   MOV AL, e_byte                          (地址表达式 e_byte)
three: MOV AL, OFFSET e_byte                   (数字 3)
Code ENDS
       END start
```

在该程序中，最容易混淆的两个表达式，是 e_byte 和 OFFSET e_byte，其中，e_byte是一个地址表达式，指令"two"将把在存贮器单元 e_byte 中的值 0FFH，送到 AL 寄存器中；而 OFFSET e_byte是一个数，标号为" three" 的 MOV 指令，将把地址表达式 e_byte的偏移量部分送到 AL 寄存器当中。可见，这两条指令是不相同的，前一条指令要从开始于55H的数据段的第3个字节中取得源操作数，该源操作数是 0FFH。而标号为"three"的MOV指令，源操作数是立即数3，它是 MOV 指令的一个部分。

标号为"one"和"three"的两条指令是相同的。

从这个例子中可以看到，一条8086的指令若使用一个地址表达式，则意味着该操作数将来自存贮器单元，而使用一个数，就意味着该数本身被用作为立即数。

地址表达式和数值表达式的区别，在汇编语言的其他地方也是十分重要的。如 DUP 中的重复计数和绝对段的段号，必须是数而不能是地址表达式。

§6.4.1 数值的允许范围

数值的最大取值范围是 -0FFFFH 到 0FFFFH，所有的算术操作，都使用 2 的补码进行运算。越出取值范围的值，将得到一个出错信息。下面的表格给出了由 ASM 86 所执行的运算的几个重要特性：

1. NOT 0FFFFH=0

2. AND、OR 和 XOR不产生越出范围的结果。

例： 0000FH AND 0FFF0H=0

0FFFFH XOR 0FFFFH=0

3. 其他运算符都有可能产生越出取值范围的结果，若结果真的超出取值范围，则给出一个溢出信息。

由于地址表达式有3个数字值分量，任何部分越出范围都是非法的，即若发生这种情况，就会发出一个出错信息。

由 DB 所定义的变量，只能从 −256 到 255 之间取值。介于 −256 和 −129 之间的数，将被作为介于 1 到127之间的正数来存放，其映照关系是：-129=127，…，−255=1，−256=0。

DW 接受整个范围内的数值。任何介于 −32,769 和 −65,535 之间的数值，将被作为介于 0 和 32,767 之间的正数来存放，映照关系是 −32,769=32,767，…，−65,535=1。

§6.4.2 算符优先规则

表达式的计算，是从左向右进行的，优先级别高的运算符将优先得到处理，当相邻的两个运算符的优先级相同时，左边的一个先计算。

利用括号可以改变运算的次序，在括号内的表达式部分首先计算。如果括号是嵌套的，则

先计算最里层的子表达式。例如

15/3+18/9=5+2=7

15/(3+18/9)=15/(3+2)=15/5=3

下表以优先级的递增次序，列出了 ASM 86 的所有运算符：

1. SHORT
2. 逻辑 OR，XOR
3. 逻辑"与" AND
4. 逻辑"非" NOT
5. 关系运算符：EQ，LT，LE，GT，GE，NE
6. 加/减：+，-（一目和二目）
7. 乘/除：*，/，MOD，SHL，SHR
8. HIGH，LOW
9. 变量控制算符："name:"，PTR，OFFSET，SEG，TYPE，THIS
10. 圆括号表达式，LENGTH，SIZE，WIDTH 和 方括号。

在最后两类运算符中(9 与 10 中)，可以包括 BYTE、WORD、DWORD、NEAR、FAR、$ 和 两种类型的地址引用，即：

一个符号名

名 1[下标表达式]

地址表达式可以含有用方括号括起来的 BX、BP、SI 或 DI，下标表达式的结果，必须是某个纯粹的数，或仅包含这些寄存器，有关下标方面的内容，将在后面进行讨论。

以字母表示的运算符与其操作数之间，必须以空格分开，以免在程序代码中，将它们看成是某个符号名。

§6.4.3 算符综述

某些高优先级的算符，不对纯数字进行操作，下面这 8 个算符，就只能用于地址表达式，它们是：段跨越前缀（"name:"）、OFFSET、SEG、用作下标偏移量的方括号、WIDTH、LENGTH、SIZE 和 TYPE，其中 WIDTH 仅用于记录。

乘法和逻辑运算符，只能对纯数字实施操作。剩下的 4 类运算符是关系、加法、高位/低位(HIGH、LOW)和指示器(PTR)，它们可以与纯数字或与来自同一段(仅允许这样)的标号和变量一起使用。子表达式，诸如那些在括号里的表达式，最终会计算成数或变量。这样，就可以判断由原始表达式指出的剩下的操作之正确与否了。

"THIS"和"$"，是允许使用这个段的程序计数器的当前值的另外两个操作。运算符 THIS，总是与某个"类型"一起使用，即与 BYTE、WORD、DWORD、NEAR 或 FAR 一起使用。若有：

```
A EQU THIS BYTE
```

则表示把 A 定义为类型是 BYTE 的一个变量，A 的地址是当前段和在这个段中的偏移量，即程序计数器的当前值指向这一代码行。与此等效的一个语句是：

```
A LABEL BYTE
```

若写成：

B EQU THIS NEAR

它把B定义成一个类型为 NEAR 的标号，这样，跳转或调用 B 指令，至多只需要一个字以用作偏移量，B 的地址也就是当前段及其该段中的当前偏移量。与此等效的一个语句是

B LABEL NEAR

一、关于属性的回顾

变量是一个存贮器单元的名字，它的内容是当作数据处理的。变量具有三重属性：段分量、偏移量和类型分量。段分量是代码块的名称，此变量就是在该代码块中被定义的；偏移量就是从该段的开头到定义该变量的行之间的字节数目；类型也就是在定义的基本单位中所包含的字节数目，即字节为1，字为2，双字为4。

标号也是某个存贮器单元的名字，只是这种存贮器单元中的内容用作指令，而不是数据。标号有4重属性，即段、偏移量、距离及 CS 的设定值，这里段和偏移量的意义与变量中的意义相同；距离属性可以是 NEAR 或 FAR，NEAR 表示该标号只可以在同一个 CS 设定的段内被引用，故引用此标号仅需一个字（作为偏移量），FAR 表示要访问该标号需要两个字：第一个为该标号在其段内的偏移量，第二个是标号所在段的段基址，当访问此标号时，必须把这个段基址装到 CS 中去，这种 CS 内容的替代过程是由一个长跳转或长调用指令自动处理的（"长"就是指段间，或 FAR）；属性 CS 设定称为段基址（名），在执行程序时，必须告诉汇编程序这方面的信息，汇编程序利用该 CS 设定值可以确定：使用当前 CS 的内容，哪些标号是可以访问的（NEAR），哪些标号是不能这样访问的（FAR），若为后者，则必须给 CS 填入适当的值，才能访问该标号，实现长跳转或调用。

二、加法运算符，＋和－

这些运算符根据一定的规则，对数、变量和标号执行17位（符号位加上16个数据位）算术整数加和减。

1．变量、标号、数总是可以加上或减去某个数字值，当把一个数加到某个变量（或标号）上时，得到的结果也是一个变量（或标号），其偏移量是此数字值与操作数变量（或标号）的原偏移量之和。

2．仅当变量和标号在同一段中时，两者才可以做减法。

3．变量和标号不能相加。

4．基址寄存器和变址寄存器可以相加，例如［BX＋SI］是合法的，对这样的寄存器或表达式，可以加上或减去某个数值，但不能从数中减去寄存器。

例：

1. MOV AX, ARRAY_START+6

这条指令把在 ARRAY_START 之后的第4个字，传送到 AX 当中。

2. TABLE2 DW NEW+17 DUP (2)

其中 NEW 必须是一个绝对的数值。

三、方括号及寄存器 BX、BP、SI 和 DI

寄存器 BX、BP、SI 和 DI，可以用作通用寄存器或变址寄存器。当把它们用作变址寄存器时，寄存器中的值，表示从某个段寄存器开始的偏移量。有无方括号，是区别这两种用法的显著标志，若某个变址寄存器出现在方括号中，则表示应把寄存器的内容用来计算偏移量。

例：

```
MOV AX, BX    ;把寄存器 BX 的内容,送到 AX 中去。
MOV AX, [BX] ;这条指令把一个字从存
              ;贮器送到 AX 中去,这个
              ;字位于段基址在 DS 中的
              ;数据段中, BX 的内容是
              ;从 DS 开始的偏移量。
```

汇编语言允许 [BX]作为一个合法的地址表达式单独出现，由于有下列缺省规则，故这种用法中的表达式是合法的：

寄存器	使用的段寄存器	类型
[BX]	DS	?
[BP]	SS	?
[SI]	DS	?
[DI]	DS	?

我们知道，每一种地址表达式，都具有 3 个分量：SEGMENT (段分量)、OFFSET (偏移量)、TYPE (类型分量)。上面这些带方括号的变址寄存器的表达式的段分量，是某个段寄存器；偏移量是变址寄存器的内容；类型是未知的。ASM 86使用指令中另一个操作数的类型，来确定这种表达式的类型。

方括号在汇编语言中，还有另外一种用途，即用于下标表达式当中，这将在后面介绍。这里只讨论方括号表达式作为地址表达式单独出现的情况。

例：

```
MOV AX, [BX],  ;由于 AX 是一个字,故也把
               ;[BX] 的类型,规定为字(WORD)
MOV CL, [DI]   ;由于 CL 是一个字节,故将
               ;[DI] 的类型,定为字节(BYTE)
```

由上面的讨论可以看出，仅在满足下面两个条件的情况下，才能这样来确定操作数的类型：

1. 有另外的一个操作数；
2. 这个另外的操作数的类型，是无二义的，即其类型是可确定的。

所以，下面的几个例子，就没有给出足够的信息以确定类型：

```
INC  [BX]      ;是把一字节还是一字加 1?
MOV [SI],3     ;是把字节 3(8 位)送入偏移量
               ;在 [SI] 中的字节,还是把字 3
               ; (16 位)送入偏移量在 [SI] 中的字?
JMP [BP]       ;是间接段间还是段内跳转?
               ;即 CS 有必要替换吗?
```

在这种情况下，汇编程序将会发出出错信息："INSUFFICIENT TYPE INFORMATION TO

DETERMINE CORRECT INSTRUCTION",表示利用给出的信息无法确定类型分量的值。

在上面关于加法算符的解释中，我们曾经指出一个基址寄存器和一个变址寄存器可以相加。此外，这些寄存器或表达式，可以加上或减去一固定的或浮动的数值，但这些只能在方括号内进行。下面的规则规定了当方括号表达式用作为地址表达式使用时，在方括号内哪些是允许出现的：

1. 仅当出现基址或变址寄存器时，才能出现数；

2. BX 和 BP 不能出现在同一个表达式当中，同样，SI 和 DI 也不能在同一个表达式中出现；

3. BX 或 BP 可以单独出现，也可以与数、SI 或 DI 一起出现；

4. SI 或 DI 可以单独出现，也可以与数、BX 或 BP 一起出现；

5. 对于数的操作不受限制，但在这里对于基址或变址寄存器只能做加法。

因此，下面的表达式是正确的：

```
[BX+DI+(SIZE a)/2]
[BP]
[BX+7]
[7+BP]
[SI-100H]      ;这是合法的，100H-SI 则不合法
[SI]
[DI+SP]
[BX+SI]
[SI+OFFSET block]
```

下面的这些表达式是不正确的：

```
[BX*7]
[BX+BP]
[BP-SI]
[3-BX]
[BX+DI*TYPE a]
```

段寄存器的确定规则是这样的：

若 BP 在表达式中，则使用 SS，否则使用 DS。

当仅有一个寄存器出现在方括号中时，方括号表达式的类型，如同上面描述的一样来决定。当然，在方括号内的表达式，也允许使用一个比单个基址或变址寄存器更复杂的变址。地址表达式 [BX+SI] 表示 BX 和 SI 内容的和，将用作为从 DS 寄存器开始的偏移量。[BP+4] 表示把 4 与 BP 相加，并将结果用作从 SS 寄存器开始的偏移量。至于 [BX-40] 则表示从 BX 中减去 40H，且结果将用作从 DS 寄存器开始的偏移量。下面给出两个较为实用的例子。

例：

在至少包含一个元素的数组中，找出其中最大的数值。下面这段程序就是完成该任务的，其中数组元素的类型为字。大致过程是这样的，首先把数组的第一个元素，当作临时的最大值，然后把每个元素和当时的临时最大值比较，若数组中的某个元素比临时最大值大，则该元素就成为临时最大值。当程序查遍数组中的所有元素时，临时最大值也就是所求的最大值。临时最

大值放在 AX 寄存器当中，BX 寄存器存放数组中下一个元素的偏移量。

```
    ASSUME CS:code, DS:data
data SEGMENT
values DW 2100 DUP (?)
count DW  ?
data   ENDS
code   SEGMENT
start:  •                          ;(初始化段寄存器,并把数读入数组,计数器中含有
        •                          ;读入的值的个数)
        •
        MOV BX, OFFSET values      ;(把起始偏移量送入 BX)
        MOV CX, count              ;(把数组中值的个数送入 CX)
        MOV AX, [BX]               ;(把数组中的第一个元素值作为临时最大值)
        JMP testlp
find_max:
        ADD BX, 2                  ;(使 BX 指向数组的下一个元素)
        CMP AX, [BX]               ;Compare(把当前的最大值与下一元素比较)
        JG testlp                  ;(若 AX 仍较大,则再测试下一个元素)
        MOV AX,[BX]                ;(否则, AX 得到新的一个最大值)
testlp: LOOP find_max              ;(继续测试下一个元素值)
done:
        •
        •
        •
code:  ENDS
        END start
```

例：

下面这个过程，将在 CRT 上打印出一个字符串，在调用名为 CRT (外部定义) 的过程时，字符放在 AL 当中。字符串以一个 null(0) 结束，SI 含有从 DS 开始的偏移量。

```
        ASSUME CS:code
code  SEGMENT PUBLIC
        EXTRN crt: NEAR
print  PROC NEAR
        MOV AL, [SI]               ;(把下一个待打印的字符送入 AL)
        CALL crt                   ;(打印该字符)
        CMP BYTE PTR [SI+1], 0     ;(检查下一个字符是否为 0)
        JE done                    ;(若是,结束打印过程)
        INC SI                     ;(否则,指向下一个字符)
        JMP print                  ;(并转到循环的开头,处理这下一个字符)
```

```
done: RET
print  ENDP
code   ENDS
```

四、变量控制算符

变量控制算符有这样几个："名:"、PTR、THIS、SEG、TYPE 和 OFFSET。在这 6 个算符中，有 2 个算符能够改变某个变量或标号的一种属性，但这通常只是在一条指令的执行期间是有效的。这样的 2 个算符是段跨越前缀字节（"名:"）和 PTR 算符。下面介绍段跨越前缀的用法。

每一条改变控制流或读/写存贮器（包括堆栈）的指令，都要用到一个段寄存器来计算所需的存贮器地址，汇编程序必须确定使用哪一个段寄存器。对于每条指令来讲，这种选择依赖于指令的寻址方式，以及当前各段寄存器中的设定值，汇编程序本身有一个固定的算法来分析地址表达式，并由此确定应该选择哪个段寄存器。

若存在多个正确的选择，例如，在多个段寄存器含有同一个段基址时，情况也是如此，在这种情况下，汇编程序总是试图选择最短的代码，即尽可能地不用前缀字节。

若所要访问的存贮器单元，只能用一个不同于硬件缺省规则规定的段寄存器实现访问时，则需要一个前缀字节（由汇编程序提供）。也就是说：

1. 若所要的存贮器单元，可以用缺省的段寄存器进行访问时，则不需要段跨越前缀字节；
2. 若使用当前任何段寄存器的设定值，都不能访问该单元时，则表示该引用出错；
3. 若通过一个与缺省的段寄存器不同的段寄存器可以访问该单元时，则需要一个段跨越前缀字节。

分析地址表达式和硬件缺省规则如下：

1. 任何具有类型 NEAR 的标号地址表达式，总是使用 CS 寄存器，不能使用段跨越前缀；
2. 任何类型为 FAR 的标号地址表达式，总将导致 CS 和 IP 寄存器取得新的值，它也不能使用段跨越前缀；
3. 使用到 SP 寄存器的所有指令，总是隐含地使用 SS 寄存器，不能使用段跨越前缀。这些指令包括 PUSH、POP、CALL、RET、IRET；
4. 若一条串指令，使用 DI 寄存器指示一个操作数，则总是用 ES 寄存器来寻址该操作数，此时，也不能使用段跨越前缀。

所有的其他情况都表示对存贮器中数据的引用，即地址表达式的类型为 BYTE、WORD 或 DWORD；

5. 若地址表达式使用 BP 寄存器（即在方括号内使用到 BP），则缺省的段寄存器是 SS。在其他情况下，缺省的段寄存器皆为 DS。

在前缀表达式"segreg: 地址表达式"当中，segreg 是某个段寄存器名。由此生成一个新的地址表达式，其段分量就是前缀表达式中的段寄存器。若"segreg"就是地址表达式的缺省段寄存器，就不产生段跨越前缀字节（因为这是多余的）；否则，即当 "segreg" 不同于缺省段寄存器时，就要生成一个前缀字节。

例：

```
ASSUME CS:code, DS:data
data SEGMENT
m_byte DB ?
data ENDS
code SEGMENT
    MOV AL, [BP]            ;[BP]的段分量为 SS,不要产生段
                            ;跨越前缀字节。 SS 是缺省的段寄
                            ;存器
    MOV AL, DS:[BP]         ;SS 是缺省的, DS 是表达式
                            ;中所直接使用的。需要
                            ;一个前缀字节以便使用
                            ;DS 段寄存器。
    MOV AL, m_byte          ;因为没有使用 BP,故 DS
                            ;为缺省的段寄存器。m_byte
                            ;的段分量为 "data", 而 "data"
                            ;又被设定在 DS 中,故不
                            ;产生前缀字节。
    MOV AL, ES:m_byte       ;DS 是缺省的,ES 是直接使
                            ;用的。故为了使用 ES寄
                            ;存器,必须产生一个前
                            ;缀字节。
    ASSUME DS:NOTHING, ES:data
    MOV AL, m_byte          ;DS 是缺省的。"data"不在 DS 中
                            ;而在 ES 中,故需为 ES 产生
                            ;一个段跨越前缀字节。
    MOV AL, DS:m_byte       ;DS 是缺省的,也是直接使用
                            ;的,故不产生前缀字节。
code ENDS
```

需要注意的是,上述例子只是说明在需要一个段跨越前缀时，汇编程序是如何作出抉择的,而不是为了说明如何正确使用 ASSUME 语句，也不是为了解释何时使用段跨越前缀算符。实际上,通常只有在下面这些情况下,才把段寄存器用作地址表达式的前缀:

1. 在一个不是堆栈段的段中使用 BP 寄存器作为变址寄存器。前已述及， 地址表达式[BP]的隐含(缺省)段分量是SS,但若将 BP 用作某个其他段内的变址时,就必须使用一个段跨越前缀。例如，DS:[BP]就表示把 BP 用作当前数据段中的一个变址;

2. 利用 SI 指出串指令的一个操作数。因为这类操作数通常处于附加段当中,故此时就要使用前缀字节"ES:";

3. 若一个程序使用附加段中的数据，则在使用方括号内的寄存器表达式去引用数据时，都需要一个段跨越前缀。例如:在指令 "MOV AX, [SI]" 中,[SI] 的段分量是 DS 寄存器，但如果要把 SI 用作为附加段中的一个变址寄存器,则需要加上前缀，即 [SI] 就要写成 ES:

[SI]。唯一的例外情况就是对于串指令的 DI 操作数而言，前缀"ES:"是不必要的，因为这时串指令总是用 ES 作为目标操作数的段基址。

当发现某些向前引用不可避免时，可以使用一个变量，其段的定义在后面给出。在这种情况下，可以把段名放在 ASSUME 语句中，并可在有关指令中把它用作一个段前缀，例：

```
    ASSUME CS:CODE, DS:DATA, ES:LATER_SEG
            .
            .
            .
    MOV AX, LATER_SEG
    MOV ES, AX
            .
            .
            .
    MOV AX, LATER_SEG:LATER_VARIABLE
            .
            .
            .
    MOV DX, ES:LATER_VARIABLE
            .
            .
```

在上面这个例子中，给出了两种对 LATER_VARIABLE 的引用方法，其中之一把 LATER_SEG 用作段跨越前缀，另一个则把 ES 用作前缀，它们都是正确的。倘若以后要改变关于段寄存器的选择，如想使用 DS 代替 ES，则对于具有适当的 ASSUME 来讲，只需在 ASSUME 中作一次改变，就能处理所有这种类型的引用；而对使用"ES:"前缀的情况来讲，就需要逐行修改；LATER_SEG 最终必须被看作为一个段名，否则出错。在一条 EQU 语句当中，有可能存在一个段跨越前缀，这样一个 EQU 语句，就会影响多条指令。例如，当遇到：

```
    NEW1 EQU ES:ARRAY[SI]
```

时，以后每次使用 NEW1 都将用到 ES 是段寄存器这一事实。

关于组，这里也要进行一些讨论，读者若没有学过有关组方面的内容，或不想使用组的话，就可以跳过下面这个部分，而去阅读有关 PTR 的说明。

组使得用一个基地址来访问组内的多个段成为可能，自然，组的最大长度也是 64K 字节，这样，用一个 16 位的偏移量，就可以访问整个组空间。所以，构成一个组的各个段的长度之和不得超过 64K。

这些段在组中的最终次序，在汇编时是不能确定的，因此，从组的基地址到一个变量或地址表达式的偏移量必定是一个浮动量，它是由 LOC 86 程序利用汇编程序提供的信息自动确定的。

为了使用组和段跨越前缀，必须记住在一个组中的变量或地址表达式具有浮动地址这一必然性。举例来说，若 G 是一个组，S 是 G 中的一个段，而 IDENT 是 S 中的一个变量，则 G：IDENT 是一个浮动的实体，它的段属性是 G，偏移量是从 G 头上到 IDENT 之间的字节

数类型就是 IDENT 原来的类型。

五、PTR运算符

PTR 运算符产生一个变量或标号。新变量的偏移量以及段分量与 PTR 右边的操作数的对应分量相同，类型由 PTR 左边的操作数指出。

例：

在用语句

```
WARRAY DW 9 DUP (0)
```

定义了 9 个字的向量之后，可能希望把它们当成 18 个字节，而不是 9 个字来访问。这时利用一般的指令如：

```
MOV AL, WARRAY[SI]
```

将是非法的，因为类型要发生冲突：AL 的类型是字节(BYTE)即 1，而 WARRAY 的类型是字(WORD)，即 2。但是，

```
MOV AL, BYTE PTR WARRAY[SI]
```

这个语句却是合法的，这是因为它显式指定所需要的是字节，而不管原来 WARRAY 的字的定义。

这种指令，并不改变原来的定义，其属性只有在这类指令当中，才发生变化。

同样，假设已知在某个其他段当中，要定义 LABEL77，那末在当前段中，就能够出现这种形式的语句：

```
JMP FAR PTR LABEL77
```

在这条 JMP 指令当中，需要两个字作为转移地址，而不是正常情况下假设的一个字。

如果在程序中，需要很多这种类型的指令，那末每次都写出这串"…PTR…"势必很麻烦。幸好有一个 EQU 命令，利用它可以为"…PTR…"定义一个代名词，例如：

```
BARRAY EQU BYTE PTR WARRAY
```

这样，在需要"BYTE PTR WARRAY"的地方，都可以用"BARRAY"来代替，显然这要方便得多。例如：

```
MOV AL, BARRAY[SI]
```

假设 SI＝3，那末这条指令将把 BARRAY 中的第 4 个字节送至 AL。

如果把 PTR 与纯数值一起使用，就将得到变量或标号，其偏移量就是给出的数值，类型为指定的类型，段分量为 0。这种表达式只有当前面有一个段前缀字节或操作符 TYPE 或 OFFSET 时，才是合法的。

例：

DS:BYTE PTR 77 可以暂时定义一个字节变量，它位于 DS 段中，偏移量是 77。如果没有段前缀(DS)，该表达式就没有段分量属性，因此是不合法的。

```
MOV AL, DS:BYTE PTR [77]
```

把当前数据段的第 77 个字节中的内容送至 AL。

假设已知段 ROUT2 中的一个字，其偏移量与段 ROUT1 中的字节变量 PATH1 相同，那末为了使用这个字，可写出下面的语句：

```
ASSUME ES:ROUT2
MOV AX, ROUT2: WORD PTR OFFSET PATH1
```

上面几个例子所给出的方法都是合法的，但使用起来不能算是很方便，若有其他较方便的方法，则最好采用它们。

下面这种类型的表达式也是合法的：

```
NEAR PTR VARIABLE_NAME
```

或

```
BYTE PTR LABEL_NAME
```

前一个，使程序员能够把执行控制，传送到原来定义为数据的某个区域，但这种尝试常常会招致错误。后一个使码子可以动态地被访问和测试，更主要的作用是，它使程序员可以在执行期间，改变指令的内容，同样，它也常会引起错误。因此，只有在确实必要的情况下，才使用这种表达式。

PTR 运算符经常与变址寄存器一道使用，下面就来分析一下这种用途的必要性。

要确定某些括号表达式的类型是不可能的，我们说下面这三种用法都是不合法的：

```
INC [SI]        ;字节增还是字增?
MOV [DI], 3     ;传送的是一个值为 3 的字,还是字节呢?
JMP [BX]        ;段内间接跳转,还是段间间接跳转?
```

利用算符 PTR 就可以解决上面的多义性问题：

```
INC BYTE PTR [SI]          ;由 SI 指向的那个字节增 1
MOV WORD PTR [DI], 3       ;把由 16 个 2 进位表示的 3,送到 DI 指向的那个字中
JMP DWORD PTR [BX]         ;执行间接的段间跳转
```

六、操作符"THIS"

正如前面指出的那样，操作符 THIS，产生一个变量或标号，其类型就在 THIS 中指定，偏移量及段分量就是汇编的当前值。

各个变量或标号的定义都包括一个类型属性，如：

```
DATA_TABLES     SEGMENT   PARA  "DATACLASS"
        A         DB          100 DUP (0)
        X         DW          300 DUP (47)
        Y         DD          100 DUP (13)
LOC1:
        .
        .
        .
DATA_TABLE      ENDS
```

表示 A 是一个字节变量组成的向量，X 是一个字变量的向量，Y 是双字变量组成的一个向量，LOC1 是一个 NEAR 标号。

在定义原来的名字的同一地点，可以利用"THIS"来定义一个类型不同的候补名字。正象 PTR 中讨论的那样，当原始定义为字时，有时希望把它们当字节访问，反过来的情况也是存在的。

下面的定义，说明了候补名字的这种命名法：

```
DATA_TABLES        SEGMENT    PARA    'DATACLASS'
          WA                  EQU     THIS WORD
          A                   DB      100 DUP(0)

          XB                  EQU     THIS BYTE
          B                   DW      300 DUP(47)

          WY                  EQU     THIS WORD
          YB                  EQU     THIS BYTE
          Y                   DD      100 DUP(13)
LOC1:
          .
          .
          .
          .
DATA_TABLES ENDS
```

WA 允许把A中的字节对，当作字来访问，A 和 WA 的偏移量及段分量是相同的，其不同点如下表所示

变 量	段 分 量	偏 移 量	TYPE	LENGTH	SIZE
WA	DATA_TABLES	0	2	1	2
A	DATA_TABLES	0	1	100	100
XB	DATA_TABLES	100	1	1	1
B	DATA_TABLES	100	2	300	600
WY	DATA_TABLES	700	2	1	2
YB	DATA_TABLES	700	1	1	1
Y	DATA_TABLES	700	4	100	400
LOC1	DATA_TABLES	1100	NEAR	—	—

与此类似，利用 XB 可以单独访问B中的每个字节；利用 WY 可以访问向量Y中的各个字 YB 可以访问其中的每个字节；WY、WB 和Y具有相同的段分量和偏移量，其区别仅在类型、长度(length)及规模(size)这几个方面。

七、SEG, TYPE 和 OFFSET

这些算符产生数值，它们所得到的分别是变量或标号的三种属性中的某一个属性值。

在上面那个例子中，由于变量都是在段 DATA_TABEL 中被定义的，故有 SEG A=SEG X=SEG Y=DATA_TABEL。运算符 SEG 可以用在 ASSUME 语句中，在构造地址时也能够使用该算符。

变量的 TYPE 给出的是其每个单位所包含的字节数，所以，

```
TYPE A=TYPE XB=TYPE YB=1
TYPE B=TYPE WA=TYPE WY=2
TYPE Y=4
```

标号的 TYPE，只可以取 NEAR 或 FAR 为值。因此，由于 LOC1 是一个标号，而不是一个变量，故从例中可以看出

```
TYPE LOC1=NEAR
```

前面已经多次讲过变量或标号的偏移量。它表示从其段名值到该变量、或标号的地址之间的距离(以字节为单位)，赋给段名的实际值，总是一个段号，但是该段的实际始址，可能要比段号高0到15个字节(由 LOCATE 程序给出)，这里的决定因素是 Segment 命令中的"定位类型"。这就意味着，汇编时所看到的偏移量，到实际装入或执行时，已经发生了变化。假如该段的类型为 PUBLIC，则表示它要与其他模块中的同名段相结合，所以有可能需要加上更多的字节到偏移量上去。

由于以上原因，当希望引用某个变量或标号的偏移量时，使用汇编给出的相对偏移量通常是不正确的，而必须使用 OFFSET 操作符，使 LOCATE 能够填进最终的实际偏移量。

下面是个典型的例子，它把偏移量用作某个向量的首地址，并将其送至基址寄存器当中，然后，通过一个变址寄存器来访问向量中的元素：

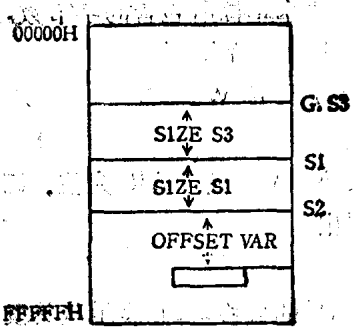
```
ASSUME DS:SEG B
        .
        .
        .
        MOV BX, OFFSET B
        MOV SI, 0
MORE:   .
        .
        .
        ADD AX, [BX+SI]
        .
        .
        .
        JMP MORE
```

尽管向量B在 DATA_TABELS 中，有一个值为 100 的相对偏移量，但是 OFFSET 算符可以保证对 B 向量的访问是正确的，哪怕B的最终绝对偏移量不是 100 也要。

此外，还可以使用 OFFSET 算符，产生正确的组内偏移量。假设某组 G 含有段 S1、S2、S3，变量 VAR 在 S2 当中，则表达式 OFFSET VAR 将给出 VAR 在段 S2 内的偏移量。由于 S2 在G当中，有时希望以组G为基址来访问 VAR，即访问 G:VAR。

在 LOCATE 为G中的所有段确定了绝对地址之后，表达式 OFFSET G:VAR 将给出运行时的当时偏移量，从而满足上面的要求。表达式中的"G:"是绝对必要的。如果不加"G:"，OFFSET 算符仅仅提供 VAR 在其段内的偏移量，而不是给出其组内偏移量。

例：下图可以说明 OFFSET G:VAR 的形成方式。



OFFSET G:VAR 的值，包括 S3、S1 及 OFFSET S2:VAR 的和，但该和还不一定就等于 OFFSET G:VAR，因为还要考虑到“定位类型”以及 S3、S2 及 S1 的终点位置等有关因素。

假设 S3 的定位类型是 BYTE，长度是 24 个字节(18H)，S1 和 S2 的定位类型是 WORD，每个长 55 个字节 (37H)。变量 VAR 是 S2 中的第 5 个字，即其相对偏移量等于 8。在 LOCATE 期间，给 G 和 S3 的地址是 12343H，即基址为 12340H(段号为 1234H)，偏移量等于 3。

这样 S3 的最后一个字节的地址是 1235AH，因为 S1 的定位类型是字，即 S1 的第一个字节必须落入某个偶数地址当中，所以 S1 不能紧跟在 S3 之后，不是从 1235BH 开始而是开始于 1235CH，没有用到 1235BH 这个单元。与此类似，S2 开始于 12394H，单元 12393H 没被使用到。

由此可见，OFFSET G:VAR 的结果是：

S3 所占字节数+S1 所占字节数+OFFSET S2:VAR+修正量=24+55+8+5=92 (5CH)

其中的修正量(5)，是对 G 的初始偏移量(3)和定位 S1 及 S2 时留下的未用字节数(2)的修正。这样，偏移量(5CH)加上 G 的基地址(12340H)，就形成了 VAR 的地址(1239CH)，它刚好是 S2 的第 8 个字节的地址。

八、圆括号、LENGTH、SIZE、WIDTH、方括号

这 5 个操作符，具有很高的优先级。其中的 WIDTH，仅可用于记录及各记录字段，§6.2 已经对此进行过讨论。

表达式中的圆括号指出一个子表达式，由于使用了圆括号，在处理整个表达式中的其他算符之前，就将处理圆括号中的子表达式，例如，在 A*(B+C) 当中，在做乘法之前，先要计算出加法的结果。

操作符 LENGTH，给出的是在定义某变量时分配给它的单位数目(不管类型为何)。比如在前面的例子 DATA_TABLES 当中，

LENGTH A=100，LENGTH Y=100

操作符 SIZE 应用于某个变量，给出的是在定义该变量时所定义的字节数目。通过下面这个公式，可以由单位数和其类型来计算这个数值：

SIZE 名字=LENGTH 名字*TYPE 名字

例如，在前面的例子中有：

```
SIZE A=100
SIZE B=600
SIZE Y=400
```

作为使用这些操作符的一个例子，请看这个代码序列，其作用是将段 ES 中的一个存贮器块清 0，其中的 BLOCK 仅表示一个字节向量：

```
ASSUME ES:SEG BLOCK, CS:RE_INIT_SEG
        MOV     DI,     0
        MOV     AX,     SEG BLOCK
        MOV     ES,     AX
        MOV     CX,     LENGTH BLOCK
ZER:    MOV     BLOCK[DI], 0
        ADD     DI,     1
        LOOP    ZER
```

如果希望在几个不同的模块中，使用该序列去处理不同的字节或字向量，那就不应该使用

```
ADD DI, 1
```

而应该用

```
ADD DI, TYPE BLOCK
```

代之，这样就得到下面这个代码序列：

```
ASSUME ES:SEG BLOCK, CS:RE_INIT_SEG
        MOV     DI,     0
        MOV     AX,     SEG BLOCK
        MOV     ES,     AX
        MOV     CX,     LENGTH BLOCK
ZER:    MOV     BLOCK [DI], 0
        ADD     DI,     TYPE BLOCK
        LOOP    ZER
```

尽管下面这个序列要花费较多的执行时间，但是它可以保留原来的 ADD 语句不动，而使它们的功能等价，不过要对原序列做这样两个改变：

1. 把 LENGTH 改为 SIZE
2. 把 BLOCK[DI] 改为 BYTE PTR BLOCK[DI]

从而得到代码序列：

```
ASSUME ES:SEG BLOCK, CS:RE_INIT_SEG
        MOV     DI,     0
        MOV     AX,     SEG BLOCK
        MOV     ES,     AX
        MOV     CX,     SIZE BLOCK
ZER:    MOV     BYTE PTR BLOCK[DI], 0
        ADD     DI,     1
```

```
        LOOP    ZER
```

下面这段代码具有相同的作用：

```
        LEA     DI, BLOCK
        MOV     AX, SEG BLOCK
        MOV     ES, AX
        MOV     CX, SIZE BLOCK
        MOV     AL, 0
        CLD
    REP STOS BYTE PTR BLOCK
```

表达式"BLOCK[DI]"中的方括号，不同于单独使用时的方括号(如"[DI]")，它用作下标，与 FORTRAN 或 PL/M 等高级语言中的下标(索引)概念相似，但不完全相同。方括号中的表达式用作索引，意思是最终要将它加到出现在方括号左边的地址表达式的偏移量上去。

索引左边，必须是某个地址表达式，索引操作的结果，总是一个地址表达式，且必须具有数据类型(BYTE、WORD 或 DWORD)，而不可以是标号。

因此，BLOCK[DI] 是一个地址表达式，且其偏移量等于 [OFFSET BLOCK+DI]。此外，由于 [BX] 是一个合法的地址表达式，所以地址表达式 [BX] [SI] 也是合法的。与 [BX] [DI]等价的一个表达式是 [BX+DI]，但前者包含下标，而后者没有，可是结果是相同的。

这类地址表达式的段分量和类型，是与索引左面的地址表达式的相应分量相同的。

回想一下方括号中的合法表达式的第一条规则：

如果至少出现一个基址寄存器或变址寄存器，那末数字就可以出现。

但在用作索引时，这条规则可以放松一些。即如果方括号是起下标的作用，那末就允许数值表达式在方括号中单独出现。所以 BLOCK[3] 是合法的，它代表 BLOCK+3。

例：对于两个规模相同的字节向量：

```
        ASSUME CS:SWITCH_SEG, DS:DATA_TABLES
            MOV  CX, LENGTH ARRAY1
            MOV  BX, OFFSET ARRAY2
            MOV  SI, 0
    NEXT:   MOV  AL, ARRAY1[SI]
            MOV  BYTE PTR[BX][SI], AL
            INC  SI
            LOOP NEXT
```

上面这段代码将 ARRAY1 中各个字节的内容，送到 ARRAY2 的相应字节当中。为了使该代码段，能应用于相同规模的字节或字向量，只需把其中的"INC SI"改为"ADD SI, TYPE ARRAY1"。假设两个向量的规模不同，那就应使循环以较小的那个向量为准，即"MOV CX"命令和标号为 NEXT 的指令中都必须用较小的那个向量名字，而把另一个向量名字放到"MOV BX"指令当中。

倘若两者的类型不同，就不能照用上面的代码段了，而应该使用 PTR 算符及另一个不同的变址器。下面给出了三个例子：

```
DATA_TABLES    SEGMENT PARA 'DATACLASS'
 A    DB        100 DUP (0)
 B    DW        300 DUP (47)
 Y    DD        100 DUP (13)
      .
      .
      .
DATA_TABLES ENDS
 SWITCH    SEGMENT PARA CODEICLASS
           .
           .
           .
           ASSUME CS:SWITCH, DS:DATA_TABLES
           MOV CX, LENGTH A
           MOV BX, OFFSET B
           MOV SI,  0
           MOV DI,  0
例1   NEXT: MOV AL, A[DI]
            MOV [BX][SI], AL
            ADD SI, TYPE B
            ADD DI, TYPE A
            LOOP NEXT
```

上面这段循环，将A的100个字节中的每个字节送到B的头100个字的各低字节中，即有A[*]=LOW. B [*].

```
例2   NEXT:  MOV AL,A[DI]
             MOV [BX][SI+1], AL
             ADD SI, TYPE B
             ADD DI, TYPE A
             LOOP NEXT
```

这段代码与前面的例子相似，不同的只是现在装满的是B的头100个字的所有高字节，即有A[*]=HIGH B[*].

下面这段代码将使得B的头100个字节中的内容与A的100个字节相同。

```
例3 NEXT:   MOV  AX, WORD PTR A[DI]
            MOV  [BX][SI], AX
            ADD  SI, TYPE B
            ADD  DI, TYPE B
            CMP  DI, SIZE A
            JGE  A_USED_PROC
            LOOP NEXT
```

```
A_USED_UP:  CALL NEW_PROC
            .
            .
            .
SWITCH ENDS
```

也可以不用比较及跳转指令，而是把"LENGTH A"的一半送到 CX 当中，即把 ASSUME 语句的下面一行改为

```
MOV CX, LENGTH A/2
```

或

```
MOV CX, LENGTH A SHR 1
```

同样，利用成块传送的串指令，也能够实现例 3 的要求。下面给出利用串指令的两种实现方法：

1.

```
ASSUME CS:SWITCH, DS:DATA_TABLES,
&      ES:DATA_TABLES                    ;(串指令需要 ES)

MOV CX, LENGTH A
LEA SI,  A
LEA DI,  B
CLD
X3: REP MOVS BYTE PTR B, A
```

2.

```
ASSUME  CS:SWITCH, DS:DATA_TABLES,
&       ES:DATA_TABLES
MOV CX,  LENGTH A/TYPE B
LEA SI, A
LEA DI, B
CLD
X3: REP MOVS B, WORD PTR A   ;(传送次数是 1 中的一半)
```

前面主要介绍了方括号的索引作用，那末方括号是否还具有其他作用呢?答案是肯定的。例如当使用 [BX] 时，则表示使用其偏移量在 BX 中的那个单元(的数据)，即要访问的数据存放在这个单元当中。下面这个语句也是有效的：

```
MOV AH, DS:[BX][SI]
```

它使得累加器的高字节接收数据，该数据的地址是 SI 的内容、加上 BX 的内容、再加上 16 与 DS 的内容之乘积，即该数据在 DS 段中。如果写出这样一个语句：

```
MOV AX, DS:[BX][SI]
```

那末，它就把 DS 中的一个字，送到全字累加器当中，该字的地址，是偏移量在 BX 中的那个单元之后的第 SI 个字节 。

方括号的另外一个用途，就是将它用在跳转或调用指令当中。大家知道，跳转与调用总

是在代码段当中进行的，即总是把 CS 的内容，用作其段基址的段号。若有语句：

JMP BX (直接跳转)

控制就将转移到偏移量在 BX 中的那个位置(在当前 CS 段内)，因此，假设 BX 含有 0C12H，则 JMP BX 就跳到 CS 始址后面的第 0C12H 号单元。

然而，在下面这个语句中：

JMP WORD PTR[BX]; (间接跳转)

方括号表示"使用偏移量在 BX 中的那个字的内容"，这种用法，意味着所需要的数据在 DS 段中，故把 BX 的内容用作访问指定字的一个偏移量(相对于 DS 中的段地址)，然后，再把这个字用来实现跳转，代替指令指示器中的内容。

在前例中，BX 含有 0C12H，"JMP WORD PTR [BX]" 不是把 0C12H 用作对 CS 的偏移量，而是把 DS:0C12H 的内容，用作 CS 的偏移量。设 DS 段中第 0C12H 个字中含有 0FAFH，则 JMP WORD PTR [BX]就把控制传送到当前 CS 段中的第 0FAFH 号单元。

[BX] 前面的"WORD PTR"指出要把一个字用作偏移量，保持 CS 的内容不变，这种结构的另一种用法为：

JMP DWORD PTR [BX]

它要用到 DS 中由 BX 指向的两个字，头一个字是如上所述的偏移量，第二个字被用来代替 CS 的内容，从而实现段间跳转。

下面这种形式也是合法的：

JMP WORD PTR [BX][SI]

执行该指令之后，IP 的内容将变为偏移量在 BX 中的单元之后的第 SI 个字节所指向的那个字((IP) = ((DS) + (BX) + (SI)))。与此类似，在跳转指令中，还可以使用带下标的地址表达式，例如：

JMP TABLE [BX][DI]

其中，表中各项的类型必须是字或双字。

九、其他操作符

本节的剩下部分将讨论下面一些操作符：SHORT、OR、XOR、AND、NOT、*、/、MOD、SHL、SHR、HIGH 和 LOW。

所有常数都是以 17 位数的形式存贮在机器内部的，其中最左边的一位表示常量的符号(0 为正，1 为负)，其余的 16 位表示数值。操作数的符号要受到逻辑和移位运算符的影响；这将在后面讨论。

1. SHORT

SHORT 告诉汇编程序为了保存表达式的终值，只需要一个字节，也就是说，只有当表达式的结果是一个单字节值时，才为它生成机器代码；若其结果大于一个字节，就会得到一个错误信息。假如使用 SHORT 的某个表达式还要经过进一步的运算，那末该运算符就将失去所有影响，把 SHORT 用于向后引用也没有什么作用。

2. OR, XOR

OR 和 XOR 分别为操作数的逻辑“或”与“异或”运算符。例如：

```
  1101 0110 B                    1101 0110 B
OR 0101 0101 B                 XOR 0101 0101 B
-------------      但是        -------------
  1101 0111 B                    1000 0011 B
```

这种逻辑操作都是按位进行的，只要操作数中有一个操作数的某位为 1，那末“或”的结果的对应位也为 1，若所有操作数的该位都为 0，结果的对应位才为 0。对于异或操作而言，只有当两个操作数对应位的取值不同时（一个为 0，一个为 1），该位的结果才为 1；否则结果位等于 0。上例的写法（垂直方向）有利于各位间的比较，较为直观，但在指令当中，常用的格式是水平方向的。即，

```
VALUE_D6H  EQU   1101 0110 B  ;=0D6H
VALUE_55H  EQU   0101 0101 B  ;=55H
MOV   AX, VALUE_D6H OR VALUE_55H; AX=D7H.
MOV   BX, 1101 0110 B XOR 0101 0101 B; BX=83H.
```

3. AND

运算符 AND，对两个操作数实施逻辑“与”操作，即只有当两个操作数的某位全为 1 时，结果的对应位才等于 1。

例：

```
    1101 0110 B        1 1111 1111 1111 1011  (-5)
AND 0101 0101 B    AND 0 0000 0000 0001 0101  (21)
-------------      -----------------------
    0101 0100 B        0 0000 0000 0001 0001  =17
```

有时把 AND 用来从某个较大的值中选出某些位，或某个位模式，即抽取出需要的位。比如说，把 00001111B 和存贮器中的一个字节相“与”，结果的低 4 位将与该存贮字节的低 4 位相同，而高4位全等于 0：

```
    1011 1101 B        1001 0110 B
AND 0000 1111 B    AND 0000 1111 B
-------------      -------------
    0000 1101 B        0000 0110 B
```

当把 AND 和 OR 结合使用时，先进行 AND 运算：

```
MASK1 EQU 00001111B
1. MOV MEM_WORD, 10010111B AND MASK1 XOR 1110B
2. MOV MEM_WORD, 10010111B XOR MASK1 AND 1110B
```

第 1 条指令把 00001001B 送到 MEM_WORD 中，因为：

```
    1001 0111 B        0000 0111 B
AND 0000 1111 B    XOR 0000 1110 B
-------------      -------------
    0000 0111 B        0000 1001 B
```

而第 2 条指令把 10011001B 送到 MEM_WORD 当中，因为：

```
    1111 B             1001 0111 B
AND 1110 B         XOR 0000 1110 B
-------------      -------------
    1110 B             1001 1001 B
```

4. NOT

逻辑运算符 NOT (逻辑"非")形成的是其操作数的1的补码(取反)，即原来为0时结果等于1；原位是1时，结果等于0。

NOT 0101 1011 B=1010 0100 B

当把 NOT 与 AND 或 OR 结合使用时，首先处理逻辑运算符 NOT；

MOV MEM_WORD, 97H AND NOT MASK1

将把10010000B送到 MEM_WORD 当中，这是因为：

NOT MASK1=NOT0000 1111 B=1111 0000 B

97H=10010111B AND 1001 0111 B

1001 0000 B

MOV DX, 97H AND NOT MASK1 XOR 1110 B

将把1001 1110 B送至 DX，因为 XOR 是最后执行的。但

MOV BX, 97H XOR NOT MASK1 AND 1110 B

则把97H送至 BX。在上面这个语句中，先做 NOT，再做 AND，最后做 XOR，AND 得到的是一个全0的结果，所以 XOR 的结果就是97H。

显而易见，下列关系是成立的：

A OR 0=A

A XOR 0=A

A AND 0FFFFH=A

其中，A 代表任何数值。

5. 关系运算符：EQ,NE,LT,LE,GT,GE

关系运算符对两个操作数进行比较，如果给定的关系为真，则结果为全1；否则结果为全0。运算符 EQ、NE、LT、LE、GT、GE 分别表示(按此次序)等于、不等、小于、小于等于 (即不大于)、大于、大于等于(即不小于)。为了对变量或标号进行比较，这些变量或标号，必须在相同的段当中被定义，这是因为关系运算符是对这些操作数的偏移量进行比较的。

在表达式中使用关系运算符的一种方法，就是与 AND 一起使用：

MOV AL, 25 AND A NE B

根据A的值，是否等于B的值，上面这个语句可有两种解释：

MOV AL, 25 (A 的值不等于B的值)

或 MOV AL, 0 (A 的值等于B的值)

TAB DW((NEW LT OLD) AND OLD+(NEW GE OLD) AND NEW) DUP(1)

说明 TAB 是一个字向量，其长度是 NEW 和 OLD 中的较大者。

但是，应该注意，不可以把数值与变量或标号进行比较。

6. 乘法和移位运算符

* 乘法

/ 除法。(除数为0时出错)

MOD 取模。结果是除法运算的余数，例 7MOD3=1，操作数只可以是绝对数值。

5+30*2

(25/5)+(30*2)

5+(−30*−2)

都等于 65D(41H)，即字符A的 ASCII 码。

SAMPLE_NUMBER MOD 8

若 SAMPLE_NUMBER 有值25，那末上面这个表达式的结果就等于1。

上面介绍了三个乘法运算符，接下来讨论移位运算符。

y SHR x　表示把操作数'y'右移'x'位

y SHL x　表示把操作数'y'左移'x'位

移位运算符不同于环移指令，它们在操作数的一头直接添0，而不"回收"移出该字节的任何位。移位操作符与其操作数之间必须用空格分开，其中的操作数 y 与 x 都必须是绝对的数值。

例：设 NUMBER 的值为 01010101，则

NUMBER SHR 2　的结果为 00010101

NUMBER SHL 1　的结果为 10101010

显然，左移一位相当于乘2，右移一位的作用相当于除2。因此左移N位相当于乘N次2，右移N位相当于除了N次2。若移位次数是一个负数，则其作用为反向移位，即：

NUMBER SHR −2

相当于

NUMBER SHL 2

SHL 和 SHR 都不影响符号位。

7. 字节分离算符

字节分离操作符，有这样两个：

HIGH　分离出16位值的高8位。

LOW　分离出16位值的低8位。

有时往往需要处理字中的一个字节，这也正是 HIGH 和 LOW 的功能。但在8086的程序设计当中，并不提倡使用这两个操作符，而只把它们用作8080到8086转换的支持工具。

此外，如果把 HIGH 或 LOW，再次作用于某个可重定位量(称为 RQ)，就将产生下面的结果：

LOW LOW RQ=LOW RQ

LOW HIGH RQ=HIGH RQ

HIGH LOW RQ=0

HIGH HIGH RQ=0

§6.5 宏指令代码

宏指令，就是事先定义过的一个代码块，其中的大部分指令和数值，都是固定的。在宏指令出现的地方，机器总是自动地对它们进行汇编，这样一来，程序员就无需在每次需要该序列时都重写一遍这样一些指令及数据。

然而，在这种定义当中使用到的有些名字，并非是固定不变的，在引用这种宏指令的同一

代码行中所给出的一些名字或数值，将取代这些“参数”。这种参数也常被叫做“哑参量”或“形式参数”，实际上，它们仅仅为实参保留一些空间。因此，形式参数指明了何处及如何使用这样一些实参。

与使用指令一样，宏指令也是用其名字来引用的，例如：

```
    ⋮
MOV   BX, WORD3
MAC1  PARAM1, PARAM2
ADD   AX, WORD4
    ⋮
```

上面出现的 MAC1，就代表某个宏指令的应用，而该宏指令是前面某个时候定义过的。很显然，它需要两个参数，也就是说，在引用这条宏指令的时候，上面给出的两个实参将取代定义中的两个形参。

实质上，上面的 MOV 及 ADD 指令也是宏指令。汇编程序的整个指令集，都被定义和说明成大量的宏指令。一旦明白这一过程，就可以加入指令，甚至可以改变汇编程序所提供的某些部分的内容。

用来说明汇编语言的这种宏指令，叫做代码宏指令，以此来与文本宏指令(text macros)相区别。对后者，程序员应该比较熟悉了，因为在前面的汇编语言当中，已经包括了这样一种功能。在此，我们不再讨论文本宏指令，下面的讨论将集中在代码宏指令的产生和使用上。

在宏指令定义的时候，将这些宏指令编码成极其紧凑的格式，以使所有定义过的代码宏指令，可以同时存放在存贮器当中。每个定义都确定了一种特定的参数组合且仅与这组参数相匹配。另外的一些参数组合可以通过重新定义代码宏指令来说明。宏指令名字相同的多重定义链在一起。这样，在调用该宏指令时，就将对整个链中的每条连接作检查，以求得到参数(操作数)一致的那条宏指令。

由于 8086 指令集由代码宏指令组成，所以很自然地把被调用的代码宏指令称做“指令”，而把其实参称为“操作数”。

例如，ADD 指令可以把任何通用寄存器或存贮器单元，用作源操作数或目标操作数，此外它还可对立即数进行操作。为了满足这些要求，就要定义 11 条代码宏指令，产生 11 条不同的机器指令，它们分别适合于这些不同的场合及操作数组合。根据源程序中实参的不同，选择出一条正确的宏指令(形参与实参一致)。具体情况，下面将进行详细讨论。

宏指令的定义，开始于确定其名字的那一指令行：

```
CODEMACRO name [formal_list]
```

或

```
CODEMACRO 名字 PREFIX
```

这里 formal_list 为一形参表，每个形式参数的格式为：

```
form_name: specifier_letter [modifier_letter] [range]
(形参名:说明符[修饰字符][取值范围])
```

这些方括号，指出它们为可选择的项，在语句当中，这些方括号并不出现，但 CODEMACRO 和名字这两者是需要的。形式参数是可选择的，倘若形参出现，那末每个形参后面，必须跟上一个说明符字母 A、C、D、E、M、R、S 或 X。在说明符字母之后，又有一个可选择项：修饰

字母，它可以是 b、d 或 w。再接下去是一个可选择的取值范围说明符，它由一对圆括号及其中的一个数字或用逗号分开的两个数字组成。说明符、修饰字符及取值范围下面还要讨论。

当没使用形参时，就可以给出关键字 PREFIX，表示该宏指令将用作其他指令的一个前缀，这也是可供选择的，REP 和 LOCK 就是 8086 指令集中的两个前缀。

宏指令的定义以下面这行结束：

ENDM 名字

在该行中，名字这一项，也是可供选择的，但是加上这个名字是有好处的，它使程序结构变得较为清楚，对代码的调试、查错也有好处。不过，若给出该名字，它就必须与 CODEMACRO 中的名字一致。

在宏指令定义的第一行，到最后一行中间的那个部分是宏指令体，每调用一次该宏指令，就将汇编和取代指令体中的位模式及形式参数。在宏指令当中，只允许下面几种伪指令出现：

1) SEGFIX
2) NOSEGFIX
3) MODRM
4) RELB
5) RELW
6) DB
7) DW
8) DD
9) 记录初始化(Record initialization)

下面给出几个简单的宏指令的例子：

```
Codemacro STC
DB 0F9H; this sets the carry flag (CF) to 1 (设置进位标志位)
Endm STC
Codemacro PUSHF
DB 9CH; pushes all flags into top word on stack (标志字压入栈顶)
Endm PUSHF
Codemacro ADD dst:Ab, src:Db
DB 04H
DB src
Endm ADD
```

头两个例子，只是允许用名字来代表机器指令，显然，这要比记一串数字容易一些，正因为如此，我们往往把这些名字称为助记符。

第三个例子，是定义 ADD 的 11 条宏指令中的一种，即它定义了 11 种加法中的一种。它有两个形式参数，即"dst"和"src"，分别用作目标操作数和源操作数，这些形参可用任何方式给出，例：

```
Codemacro ADD anything:Ab, other:Db
DB 04H
DB other
```

Endm ADD

这个定义无论在功能上，或在格式上，都与上面第三个例子相同。

1．说明符

每个形式参数都必须有一个说明符字母，以便指出与形参一致的操作数类型，这样的说明符共有以下8个：

1) A：表示累加器，即 AX 或 AL。

2) C：表示代码，即只可以是标号表达式。

3) D：数据，即是一个用作立即数的数。

4) E：有效地址，它可为M(存贮器地址)或R(寄存器)。

5) M：存贮器地址，它可是一个变元(有或无变址)，也可以是一个括起来的寄存器表达式。

6) R：仅仅表示通用寄存器，而不可以是一个地址表达式，不可是用括号括起来的某个寄存器，也不可以是一个段寄存器。

7) S：仅表示某个段寄存器，即为 CS, DS, ES 或 SS。

8) X：直接存贮器寻址，它是一个无变址的简单变量名。

究竟哪些操作数与哪些说明符相匹配呢？有关这一问题的讨论将在后面进行。

2．修饰符

可供选择的这种修饰字符，进一步对操作数提出了要求，它涉及到待处理数据的取值范围，与由操作数产生的代码量也有关系。修饰符的意义依赖于操作数的类型：

对于变元而言，修饰符要求操作数具有一定的类型，规定"b"代表字节、"w"表示字、"d"代表双字。

对标号而言，修饰符要求产生的目标代码有一定的量，其中"b"代表以 NEAR 标号为基础的一个8位相对偏移；"w"也用于 NEAR 标号，但它的偏移范围不再在－128到127之内。

对数来讲，修饰符要求该数值具有一定的取值范围，这里，"b"表示取值范围是－256到255，"w"为其他数值。注意不允许出现"Dd"说明符－修饰符对。

在本书当中，说明符是用大写字母表示的，而修饰符则用小写字母表示，这样以便于码子的区分，但也并非非这样不可，因为它们也就象字符串之外的所有源程序代码一样，汇编程序并不区别大写和小写字母。

3．取值范围说明

倘若给出取值范围，它可是单个表达式，也可是用一个逗号分开的两个表达式，其中的每个表达式都必须是一个寄存器或一个真正的数，即不可以为一个地址。例：

1) Codemacro IN dst:Aw, port:Rw (DX)

2) Codemacro ROR dst:Ew, count:Rb (CL)

3) Codemacro ESC opcode:Db (0,63), adds:Eb

例1) 是IN (输入)指令的4种宏指令中的一种。它表明若某个寄存器用来确定输入一个字的转接口，那末只可以用DX，这样它才能与宏指令一致起来。其他的任何寄存器都不可作此用途，否则，将指出该行源程序有错。例如：

IN AX, BX

是不合法的。

例 2) 是 4 个右环移宏指令之一，表示待环移的字，可以是除段寄存器之外的任何字寄存器，或存贮器中的任何字，待右移的位数放在“count”当中，“count”在这里被定义为一个字节寄存器，且规定它是 CL，这样就不可以使用任何其他寄存器了（例：ROR DL 是一条错误指令）。

例 3) 表示 ESC 指令的第一个参数“opcode”必须是一个立即数字节，其取值范围为 0 到 63。

4. 段跨越前缀 (Segfix) 控制

SEGFIX 是一条包含在某些宏指令定义当中的伪指令，它让汇编程序去确定为了访问一个给定的存贮单元是否需要一个段跨越前缀字节，若需要，那末该前缀就作为指令的第一个字节；若不需要也就不做任何动作。

该伪指令的格式为：

SEGFIX formal_name

这里，formal_name 是形式参数的名称，它表示存贮器地址，正因为它是一个存贮器地址，该形参必须有 E、M 或 X 这样的说明符。

在不存在段跨越前缀字节的情况下，8086 的硬件就使用 DS 或 SS，究竟用哪个要根据使用的基址寄存器 (若使用的话) 来确定：BP 意味着用 SS，BX 意味着用 DS，不使用基址寄存器时，也意味着用 DS 作为段基 (当然，这要包括三种可能：只用 SI、只用 DI 或无变址)。汇编程序必须确定通过这种硬件隐含的段寄存器是否可以达到待访问的存贮单元。

汇编程序检查存贮器地址表达式的段特征，其中的存贮器地址表达式，是作为实参给出的，至于段的表征值，可能是一个段，也可能是一个组 (group) 或某个段寄存器。

假设它是一个段，那末汇编程序就要确定该段，或包含该段的某个组是否已被设定成为此硬件隐含的段寄存器。如果是这样的话，就不再需要段跨越前缀；否则汇编程序就要检查其他一些段寄存器的设定情况，寻找包含该段的段或组，若找到这样的段或组，就得加上这个段跨越前缀字节，否则给出一个出错信息。

倘若段特征值为一个组，汇编程序也执行与上类似的动作，只是这里不包括“包含组”的情况，因为组本身必须被设定为某个段寄存器。否则出错。

如果存贮器地址表达式的段特征值是一个段寄存器，那末汇编程序就看它是否就是硬件隐含的那个段寄存器，若是这样的话，就不需要段跨越前缀字节；否则，就要发出这样一个前缀字节，说明要使用这个指定的段寄存器。

段跨越前缀的码字为：

00100110：ES 前缀
00101110：CS 前缀
00110110：SS 前缀
00111110：DS 前缀

5. Nosegfix

NOSEGFIX 仅用于某些特殊操作数，在那些指令当中，前缀是不合法的，因为这些指令中的操作数除了可用 ES 寻址外，不能使用任何别的段寄存器。这种情况只有在串指令 CMPS、MOVS、SCAS 和 STOS 的目标操作数寻址时才会发生。

NOSEGFIX 的格式为：

NOSEGFIX segreg, formal_name

这里"segreg"是4个段寄存器 ES、CS、SS、DS 当中的一个,"formal_name"是存贮器地址形参的名字,作为存贮器地址,它应有说明符 E、M 或 X。

若汇编程序在汇编一条指令时,碰到一个 NOSEGFIX, 它所需要完成的唯一动作,就是一次出错检查,这条伪指令不产生目标代码。

汇编程序首先检查对应于 "formal_name" 的实参(存贮器地址)的段特征值。若段特征值是某个段寄存器,它必须与 "segreg" 相同;若段特征值为一个组,那末它就必须被设定成"segreg";若该段特征值是一个段,那末该段或包含该段的某个组,就必须被设定成"segreg";如果这些测试条件都没得到满足,则表示从"segreg"不可到达(访问)"formal_name",故报告出错。

由上面列出的那些串指令所使用的"segreg"值为 ES。

6. Modrm

这条伪指令,导致汇编程序产生一个 ModRM 字节,该字节位于许多 8086 指令的操作码之后,它具有下面这些信息:

1) 变址类型或指令使用的寄存器号码。

2) 还使用到哪个寄存器,以及选择该指令的其他一些信息。

MODRM 字节,由三个字段组成。

mod 字段,占据该字节的最高两位,它与 r/m 字段组合起来,可以得到 32 种可能值:8 个寄存器及24种变址方式。

reg 字段, 占据 mod 字段右面的 3 位,它所确定的是某个寄存器或用作操作码信息的 3 个补充位。reg 字段的意义是由指令的第一个字节(操作码)确定的。

r/m 字段,占据 MODRM 字节的最后 3 位,它可以指定某个寄存器作为一个操作数,它也可以与 mod 字段一道,形成寻址方式编码。

该命令的格式是这样的:

MODRM formal_or_number, formal_name

这里"formal_or_number",是一个形参名或一个数,"formal_name"是另一个形参的名字。

"formal_or_number" 代表装入 ModRM 字节的 reg 字段的内容,假如它是一个数,那末在每次调用这条宏指令的时候,都将在 reg 字段中装入相同的值,这时,该数就是确定硬件执行哪条指令的操作码的扩充。如果 "formal_or_number" 是一个形参,那末就把对应的操作数(通常是一寄存器号码)插入。

"formal_name" 代表一个有效地址参数,汇编程序首先得确定所提供的操作数是一个寄存器、一个变元、还是一个变址变元,然后再设置 mod 和 r/m 字段的内容,以使其能正确地表达该操作数。假如该操作数还有一个 8 位或 16 位的偏移量的话,汇编程序也应该产生这样一个偏移量。

下例为一个使用了 ModRM 的8086 指令:

```
Codemacro ADD dst:Rw, src:Ew
Segfix src
DB3
MODRM dst, src
```

```
Endm ADD
```

其中的说明符 Rw 和 Ew 表明：只有当引用中的实参，是一个整字通用寄存器(用作目标操作数)和一个整字的源操作数(存贮器地址或通用寄存器)时，才能满足这个宏指令的要求。

例 1：

```
ADD DX, [BX] [SI] 成为
00000011  10010000
76543210  76543210
```

第一个字节指出，这是一条将存贮器字加到一个寄存器中去的 ADD 指令。根据最低两位的取值，可得到 4 种不同形式，这里只给出了其中之一。如果第 1 位为 0，那末 ADD 就是从一个寄存器加到一个寄存器或存贮器单元中去；若第 1 位为 1，则 ADD 就将某寄存器或存贮器单元的内容加到一寄存器当中。最低位，即第 0 位，指出相加的数据是字节(0)还是字(1)。

第二个字节为 MODRM 字节，第 5, 4, 3 三位为 DX 的编码 010，第 7, 6 位是 mod 字段，编码是 10，R/M 字段的内容是 000。

如果源代码中包含一个变元，如：

```
ADD DX, MEMWORD [BX] [SI]
```

那末此偏移量 MEMWORD，就紧接在 MODRM 字节的后面，先是低位字节，后是高位字节。

例 2：ADD DX, [DI]

```
00000011  10010101
76543210  76543210
```

作为例子，考虑这种情况：目标操作数是存贮器中的字，而源操作数是一立即数。相应的宏指令代码为：

```
Codemacro ADD dst:Ew, src:Dw
Segfix dst
DB81H
MODRM0, dst
DW src
Endm ADD
Codemacro ADD dst:Ew, src: Db (-123, 127)
SEGFIX dst
DB83H
MODRM0, dst
DB src
Endm ADD
```

由于数据既可是字节也可以是字，因而为该指令及数据所产生的目标代码也是不相同的。

此外，下面这些指令的 MODRM 行所得到的“formal_or_number”字段的内容为 0，即三位全 0，而在前面的两个例子当中，它所确定的是目标操作数，为了表示 DX，故其内容为 010。

例 3：ADD [DI], 513

```
10000011 10000101 00000001 00000010
```

例4：ADD BYTE PTR [BX] [SI], 4

10000001 10000000 00000100

立即数(字或字节)位于 MODRM 字节之后。

7. Relb 和 Relw

这些伪指令用于调用过程或跳转指令当中，其作用就是命令汇编程序产生所需的位移量——从该指令末端到作为一个操作数的标号之间的“距离”，它以字节为单位。也就是说，RELB 产生一个字节的位移量，RELW 产生两个字节的位移量，所谓位移量也就是该指令的指针值(PC，它指向该宏指令代码的末端)与目标地址之间的距离。

这两条伪指令的格式如下：

RELB formal_name

或

RELW formal_name

其中，“formal_name”是一个带有说明符“C”(代码)的形参名。

汇编语言假设所有的 RELB 和 RELW 命令，都紧跟在宏指令代码的操作码字节之后，就象 8086 指令集中的 JUMP 和 CALL 指令那样。为了确定(在与宏指令代码匹配过程中)位移量的始点，它也需要这样一个假设，从而可以确定一个操作数为“Cb”或“Cw”。尽管汇编程序也允许 RELB 和 RELW 出现在宏指令定义中的其他地方（例如在一个多指令的宏指令中)，但是这样一来，在引用该宏指令时，就要冒出错的风险了。如果一个“b”被匹配成“w”，就浪费了一个字节；若“w”被匹配成“b”，就报告出错。

RELB 和 RELW 的应用举例：

```
Codemacro JMP place:Cw
DB 0E9H
RELW place
Endm JMP
Codemacro JE place:Cb
DB 74H
RELB place
Endm JE
```

它们都是直接跳转到 CS 段中的某个标号。第一个宏指令代码中的形参说明符，表示目标是在当前 CS 段中的一个 NEAR 标号 (若为 Cd 就表示 FAR)，它意味着一个 16 位的位移，能够到达段中的任何一个单元。RELW 计算其距离，并把这个字结果送到指令字节 0E9H 之后。

假设目标的偏移量为 513，那末该宏指令代码就将产生指令：

11101001 00000001 00000010

距离的计算从 RELW 的末端开始。

需要注意，只有在标号是在同一个 “ASSUME CS:” 作为跳转的名字下进行汇编时，才有可能匹配。也只有这样，才真正产生目标代码。

第二个例子是一个条件转移指令，即仅在条件满足时才跳转，例子为若相等则跳转，即跳转的条件是 ZF=0。条件转移中的跳转距离，限制在一个字节之内，即往前不能跳过 127 个

字节，往后不能跳过 128(−128) 个字节。

假设目标是往前跳 99 个字节，则该宏指令代码就将产生指令：

01110100 01100011

距离的计算从上面两个字节之后算起。

8. DB, DW 和 DD

这些命令类似于宏指令代码定义之外出现的那些 DB, DW 和 DD，但是，由它们所接收的操作数之间也存在着一些差别。

它们的命令格式是：

DB cmac_expression

或

DW cmac_expression

或

DD cmac_expression

这里 cmac_expression 可以是一个赋以定值的表达式、一个形参名、或一个具有点记录字段移位结构(dot_recordfield shift construct)的形参名。

第一种情况(定值)表示每次引用该宏指令代码定义时，都汇编同一个数值；形参意味着要汇编对应的实际操作数；点记录字段移位结构表示实际操作数首先要移位，然后才插进去，这一点后面还将谈到。

这些宏指令代码初始化操作数受到一定的限制，在此表格以及 DUP 计数，都是不允许出现的。

9. 记录初始化(Record Initializations)

记录初始化命令，为我们提供了在宏代码定义当中，对短于一个字节的位字段实施控制的手段，其格式为：

record_name [cmac_expression_list]

其中：record_name 是前面已经定义过的记录名，cmac_expression_list 是一个用逗号分开的 cmac_expression 表。在列出该表时，方括号无需写出，这里它仅表示此表是可供选择的。就象前面所讲过的那样，cmac_expression 是一个数、一个形参、或是一个移位形参。此外，该表也可为空，这时，记录的字段值与 RECORD 定义中所作的规定相同。

这条伪指令命令汇编程序将一个字节或字装配起来，就象表达式表所规定的那样，来使用常数值及提供的操作数。待装入的数值有可能填不满这些记录字段，这时并不报告出错，只是使用低位部分而已。

10. 利用点运算符实现参数移位

如前所述，移位形参可以作为 DB,DW 或 DD 的一种特殊的操作数，也可以作为记录初始化的操作数分量。移位形参的结构形式为：

formal_name, record_field_name

formal_name (形参名)对应的操作数，是一个绝对数值(absolute number)，record_field_name 是某个记录字段的名称。如果利用由该记录字段定义的移位计数，那末，当引用相应的宏指令代码时，汇编程序就将给这个表达式赋值。

在 8086 指令集中，ESC 指令用到这一特性，它允许与使用相同总线的其他设备进行通

讯。若给出的是一个地址，ESC 就将该地址放到总线上；若给出的是一个寄存器操作数，就不再需要放地址到总线上去。这样就能执行外部设备发来的命令，有无操作数都可以。在 ESC 的宏指令代码当中，这些命令是用 0 到 63 中的数来表示的，其解释工作由外设完成。

```
R53 Record RF1:5, RF2:3
R233 Record RF6:2, mid3:3, RF7:3
Codemacro ESC opcode:Db(0, 63), addr:E
Segfix    addr
R53       <11011B, opcode, mid3>
ModRM  opcode, addr
EndM
```

宏代码体中的 R53 行，产生这样一个 8 位模式：高 5 位为 11011B，低 3 位装入实参，其中实参就是"opcode"(操作码)右移 mid3(=3)位所得到的结果。

例：假设希望使用 ESC 指令，"opcode"为39，"addr"为 MEMWORD，其偏移量是在 ES 中的 477H，用 DI 进行变址寻址。

```
ESC 39, ES:MEMWORD [DI]
    SEGFIX addr 成为 ES:=00100110B
    39=00111001B
    opcode, MID3= (000) 00111
    R53<11011B, opcode, mid3> 变成 11011111B
    对 [DI] 而言, MOD=10, R/M=101
```

MODRM opcode, addr 把 "opcode" 送到 modrm 字节的第 5—3 位，modrm 的其余各位 (7, 6, 2, 1, 0) 将装入适当的 mod 和 R/M。

由于 opcode 是 6 位的，而待装入的字段仅 3 位(第 5—3 位)，故仅用低 3 位，即 111，略去高位部分 100 (因为 opcode=39=100111B)。

因此，"MODRM opcode, addr" 就变为 10111101B，后跟位移量 MEMWORD，即 0111011100000100。

这样，我们就得到了 ESC 的整个目标代码如下：

```
0010 0110  (第 1 字节)
1101 1111  (第 2 字节)
1011 1101  (第 3 字节)
0111 0111  (第 4 字节)
0000 0100  (第 5 字节)
```

注意：opcode 的 6 位分成了两个部分，即位于第 2 字节的最后 3 位以及第 3 个字节的第5,4,3三位。

11. PROCLEN

PROCLEN 可作为一个特殊的操作数，如果当前的 PROC (过程)被说明成 NEAR，那末它就等于 0，若把当前 PROC 说明为 FAR，它就等于 0FFH，位于 PROC…ENDP 外面的码字被看作是 NEAR。为了从 CALL 返回到一个 NEAR 或 FAR 的过程，在产生正确的机器指令时，RET 宏指令使用了这样一个算符。

```
Codemacro RET
R413    <0CH, PROCLEN, 3>
Endm    RET
```

这个宏指令没有使用较为熟悉的 DB 或 DW 存贮器分配指令，而是利用了前面定义过的记录。同样，可以用 DB 实现这一功能，区别在于尖括号中给出的初值，以此让记录的每个字段，都获得自己的初始值。由于在上面的尖括号内给出了 3 个以逗号分开的数值，故可以推论：记录当中至少包含了 3 个字段。

作为汇编程序的初始动作之一，需要定义这样的一些记录，以此去定义汇编程序的指令集合。例：

```
R53   Record RF1:5, RF2:3
R323  Record RF3:3, RF4:2, RF5:3
R233  Record RF6:2, Mid3:3, RF7:3
R413  Record RF8:4, RF9:1, RF10:3
```

最后一行 R413 定义了一个由 3 个字段组成的 8 位记录，高 4 位(7, 6, 5, 4)叫做 RF8，接下来的一位(第三位)叫做 RF9，低 3 位(2, 1, 0 位)叫做 RF10。当把 R413 用作存贮器分配命令时，必须在尖括号内为所有字段设置初值，这是因为在定义当中，没给记录的任何字段设置初值的缘故。

在上面的 RET 的宏代码中，字段 RF8 设置成 0CH=1100，RF10 设置为 3=011，记录字节的第三位即 RF9 的内容，要根据当前的 PROC (RET 就在该过程中出现)的类型确定，若当时过程的类型为 NEAR，则 RF9 为 0；若类型为 FAR，则 RF9 为 1。

必须注意，PROCLEN 给出的是一个 8 位的结果，全 0 或全 1，然而 R413 仅仅使用了一位。字段的宽度确定了它所使用的位数。

12. 指令与宏代码的匹配(Matching of Instructions to Codemacros)

下面的讨论，可算是对 8086 汇编语言的核心进行的。为了产生代表多条硬件指令的单个汇编指令助记符，应该仔细地对给定指令 (如 ADD 指令) 的宏代码定义链进行排序，还要结合考虑对操作数的类型需求。

指令与某特定的宏代码定义之间的匹配算法是这样的：

1) 在第一遍扫描时，给实参赋值，那些包含向前引用的参量需经特殊处理。

2) 倘若某个实参是一个没有相关类型的寄存器表达式 (如[BX])，或者是一个隐含地利用了累加器的指令 (如"MOV, 3") 的话，那末就要检查其他参量，看其是否至少有一个包含修饰符(确定类型)的参数。一般说来，数配上"b"是不够的；但数与"w"相配、显式给出寄存器和所有带类型的变量到确实能够区分修饰类型。假如没能发现修饰参数，就将发出出错信息"INSUFFICIENT TYPE INFORMATION TO DETERMINE CORRECT INSTRCTION"，表示没能获得匹配。

3) 为了获得指令与其宏代码定义之间的匹配，先要搜索给定指令的宏代码定义链，从该指令的最后一个宏代码定义开始往后搜索。要取得匹配，实参的数目就必须与定义中的形参数目一致，同时还要求实参与形参的说明类型、修饰符规定及取值范围一致起来。细节如下：

a. 说明符 SPECIFIERS

向前引用(Forward reference) 与 C, D, E, M, X 一致

AX 和 AL 与 A, E, R 一致

标号与C一致

数与D一致

非变址变元与 E, M, X 一致

变址变元和寄存器表达式与 E, M 一致

除段寄存器之外的寄存器与 E, R 一致

段寄存器 CS, DS, ES, SS 与S一致

b. 修饰符 MODIFIERS

修饰符的匹配特征依赖于已匹配的说明符的类型：

对数来讲，-256 到 255 之间的数只与"b"匹配，其余的数值仅与"w"匹配。

对标号而言，那些具有同一个 CS 设定 (assume) 且离宏代码头不超过-126到+129的 NEAR 标号，只与"b"相匹配；其余那些具有相同的CS 设定的 NEAR 标号，仅与"w"相匹配；CS 设定不同的那些 NEAR 标号，不会与任何修饰符匹配；FAR 标号与"d"相匹配。

对变元来说，类型 BYTE 与"b"一致。

类型 WORD 与"w"一致。

类型 DWORD 与"d"一致。

其他数值类型不会与修饰符匹配。

向前引用与任何修饰符都相匹配，除非前面加上这样一些信息：BYTE PTR、SHORT、FAR、PTR 等等。

没有规定类型信息的变址寄存器表达式(例[BX])，可与"b"或"w"相匹配。

c. 取值范围(RANGES)

取值范围说明符，仅仅对数值或寄存器这种参数才是有效的，即说明符应为 A、D、R、S。假如某说明符后面跟有一个形参，则实参的值必须与之一致；若跟有两个形参，则要求实参的值落入这两个取值范围说明符之内。为了检查这种一致性，假定作为实参传递的寄存器具有下列数值：

AL：0

CL：1

DL：2

BL：3

AH：4

CH：5

DH：6

BH：7

AX：0

CX：1

DX：2

BX：3

SP：4

BP：5

SI: 6
DI: 7
ES: 0
CS: 1
SS: 2
DS: 3

如果存在这样的取值范围说明符，则向前引用就不会与形参相匹配。

4) 倘若发现指令与其宏代码的说明有一个相匹配，那就估算一下为产生目标代码所需的字节数目。向前引用的变元，除非显式给出前缀信息，都被假定为不需要段跨越前缀字节，此外，还假设包括向前引用的那些 Mod/RM 需要 16 位的位移量，但当向前引用有 SHORT 修饰时，就假定需要 8 位的位移量。

5) 在第二遍扫描过程中，对此宏代码链的搜索又进行一遍，与第一遍扫描一样，它也从链的末端开始。但向前引用的分解，可能导致与一个不同的宏代码相匹配。

6) 在第二遍扫描中，发出指令产生的目标代码。如果输出的字节数，超过了 4) 中所作出的估计数，则发出一个出错信息且拒绝给它另外一些字节，因此指令不完整，程序也就不好运行；假如使用的字节数少于估计数，就在多余的字节中放上 90H (NOP 空操作指令)。

与许多别的指令一样，ADD 指令提供了一个很好的宏代码匹配的例子，它的 11 种宏代码定义处理这样一些情况：

	目标操作数	源操作数
①	存贮器字节	立即数字节
②	存贮器字	立即数字节(不在 −128 与 127 中间)
③	存贮器字	立即数字节(从 −128 到 127)
④	存贮器字	立即数字
⑤	AL	立即数字节
⑥	AX	立即数字节
⑦	AX	立即数字
⑧	存贮器字节或字节寄存器	字节寄存器
⑨	存贮器字或字寄存器	字寄存器
⑩	字节寄存器	存贮器字节或字节寄存器
⑪	字寄存器	存贮器字或字寄存器

在宏代码定义当中，利用说明符—修饰符这样的两个字母组合来代表上面这些情况，这样一来，就可较为方便地来扫描、理解这些宏指令代码。对应于上面的 11 种情况，下面给出它们各自的宏代码的第一行：

```
1. Code Macro ADD dst:Eb, src:Db          (TO EA byte FROM data byte)
2. Code Macro ADD dst:Ew, src:Db          (TO EA word FROM large data byte)
3. Code Macro ADD dst:Ew, src:Db(-128,127) (TO EA word FROM signed data byte)
4. Code Macro ADD dst:Ew src:Dw            (TO EA word FROM data word)
5. Code Macro ADD dst:Ab, src:Db          (TO AL FROM data word)
6. Code Macro ADD dst:Aw, src:Db          (TO AX FROM data byte)
```

```
 7. Code Macro ADD dst:Aw src:Dw        (TO AX FROM data word)
 8. Code Macro ADD dst:Eb, src:Rb       (TO EA byte FROM register byte)
 9. Code Macro ADD dst:Ew, src:Rw       (TO EA word FROM register word)
10. Code Macro ADD dst:Rb, src:Eb       (TO register byte FROM EA byte)
11. Code Macro ADD dst:Rw, src:Ew       (TO register word FROM EA word)
```

其中 EA 代表一个有效地址表达式，它既可是某个存贮器单元地址，也可以是一个寄存器。

宏指令代码的顺序是很关键的。例如，指令“ADD AX, 3”不仅与第 6 个定义相符，同时也与第 2 个定义相符，这是因为 AX 同时满足 Ew 及 Aw 的定义。由于定义 6 产生的目标代码较短，因此应该在选择定义 2 之前先选中定义 6，正因为如此，所以把它安排在后面，这样当汇编程序从定义 11 往上搜索时，就将先碰上定义 6。

设下面的用户符号已由下列特性定义：

```
BYTE_VAR     字节变元
WORD_VAR     字变元
WORD_EXPR    存贮器地址表达式
B_ARRAY      字节变元
```

则以下的汇编程序指令，就将与上面的宏指令代码定义具有一定的对应关系，指令右面的数字，表示该指令对应的宏代码编号。

```
ADD  AX, 250                    → 6
ADD  AX, 350                    → 7
ADD  BX, WORD_EXPR              →11
ADD  BX, DX                     →11
ADD  BYTE_VAR, AL               → 8
ADD  BYTE_VAR,254               → 1
ADD  WORD_VAR, CX               → 9
ADD  DH, BARRAY[SI]             →10
ADD  CL, BYTE_VAR               →10
ADD  AL, 3                      → 5
ADD  WORD_VAR, 35648            → 4
ADD  WORD_VAR, OFFSETB_ARRAY    → 4
ADD  [BX][SI], AH               → 8
ADD  [BP], CL                   → 8
ADD  DX, [DI]                   →11
ADD  AX, [SI][BP]               →11
ADD  WORD_VAR, 3                → 2
ADD  WORD_VAR, 255              → 3
```

§6.6 汇编语言程序设计举例

通过前面几节的介绍，可以说 8086 汇编语言所具有的一些特征、结构、规则与细节都呈现

在读者面前了，当然某些地方可能讲得不够清楚，不太容易理解，为了尽力弥补这些不足，我们准备在本节中奉献给读者几个汇编语言程序，作为例子，希望有助于大家掌握汇编语言，学会用8086的汇编语言来设计程序(系统程序或用户程序)。

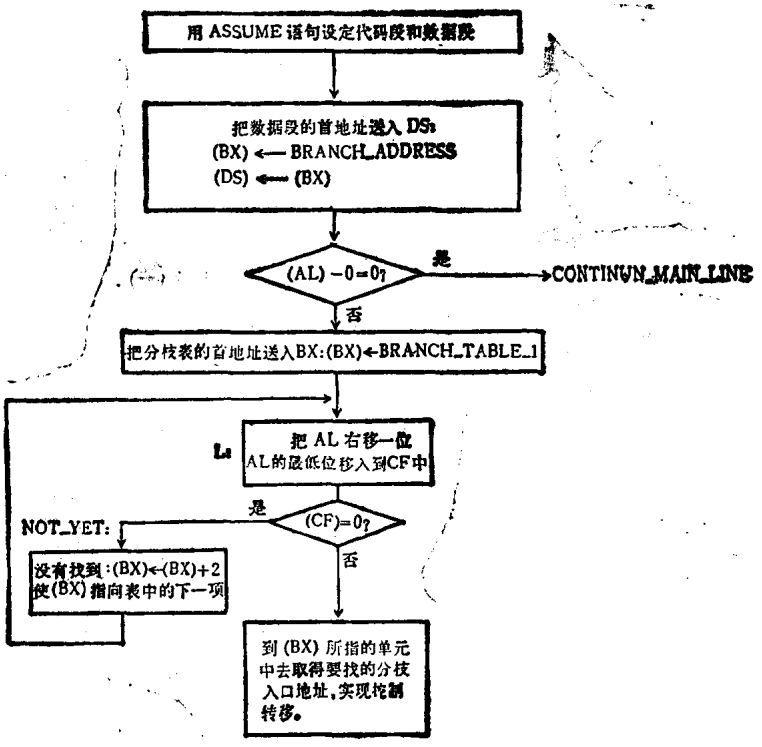
例1：

例1是一个依据累加器哪一位是1，把控制转移到8个可能的分枝中去的程序。

```
BRANCH_ADDRESSES SEGMENT    (BRANCH_TABLE_1 被定义为 8 个分枝入口
                                    表的首地址)
                BRANCH_TABLE_1    DW ROUTINE-1
                                  DW ROUTINE-2
                                  DW ROUTINE-3
                                  DW ROUTINE-4
                                  DW ROUTINE-5
                                  DW ROUTINE-6
                                  DW ROUTINE-7
                                  DW ROUTINE-8
                BRANCH_TABLE_2    DW PROCESS-31
                                  DW PROCESS-61
                                  DW PROCESS-81
                                  ⋮
BRANCH_ADDRESSES ENDS
    PROCEDURE_SELECT SEGMENT
    ASSUME  CS:PROCEDURE_SELECT,
&           DS:BRANCH_ADDRESSES
    MOV  BX, BRANCH_ADDRESSES
    MOV  DS, BX                   ;moves above segment base-address into
                                  ;segment register DS.
                                  (把上面的段基地址送入段寄存器 DS)
    CMP  AL, 0                    ;this test assures that some bit of AL has
    JE   CONTINUE_MAIN_LINE       ;been set by earlier instructions to specify
                                  ;a routine (prior insts. not shown).
                                  (这个测试确保 AL 中的某些位已被以前的
                                  指令置位，以指示某一个分枝 (以前的指令
                                  没有表示出来))
    LEA  BX, BRANCH_TABLE_1       ;BX set to location holding address of
                                  ;first routine.
                                  (把分枝入口地址表首地址送入 BX)
L:  SHR  AL, 1                    ;puts least-significant bit of AL into the
                                  ;carry flag (CF).
```

```
                                    (把 AL 的最低位移到进位标志 (CF) 中)
        JNB    NOT_YET              ;if CF=0, the ON bit in AL has not yet
                                    ;been found.
                                    (若 CF=0，在 AL 中置位的位还没有找
                                    到)
        JMP WORD PTR [BX]           ;if CF=1, then control is transferred (see
                                    ;explanation below).
                                    (若 CF=1,则转移控制,到[BX]所指的单
                                    元中去找相应的入口)
NOT_YET: ADD BX, TYPE BRANCH_TABLE_1
                                    ;if no transfer, then the bit that is ON
                                    ;has not yet been found, so BX is set to
                                    ;point to the next entry in the address-
                                    ;table by adding 2.
                                    (若不转移,则置位的位还没有找到,因此通
                                    过加2把BX设置为指向地址表中的下一
                                    项).
        JMP     L                   ;Jump to L to shift and retest
                                    (跳转到L，再进行测试)
CONTINUE_MAIN_LINE:                 ;we reach here only if no bit was set to
                                    ;indicate a desired ;rontine
                                    (仅当没有任何位被置位来指出一个所要的
                                    分枝时,才到达这里)
ROUTINE_1:
                    .
                    .
                    .
ROUTINE_2:
                    .
                    .
                    .
ROUTINE_3:
                    .
                    .
                    .
PROCEDURE_SELECT ENDS
```

例1的主要流程可以用框图表示如下：



例 2：

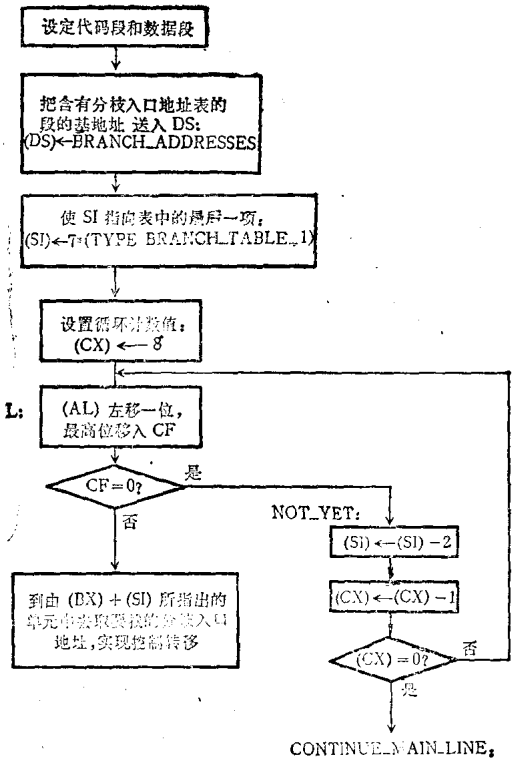
例 2 实现和例 1 一样的控制转移，只是使用了不同的技巧来选择有关的地址。它把 AL 的高位移入 CF 并把 SI 寄存器用作进入分枝表的变址器。

```
BRANCH_ADDRESSES SEGMENT
        BRANCH_TABLE_1      DW ROUTINE_1
                            DW ROUTINE_2
                            DW ROUTINE_3
                            DW ROUTINE_4
                            DW ROUTINE_5
                            DW ROUTINE_6
                            DW ROUTINE_7
                            DW ROUTINE_8
        BRANCH_TABLE_2      DW PROCESS_31
                            DW PROCESS_61
                            DW PROCESS_81
                            ⋮
BRANCH_ADDRESSES ENDS
PROCEDURE_SELECT SEGMENT
    ASSUME CS:PROCEDURE_SELECT,
&           DS:BRANCH_ADDRESSES
```

```
        MOV   BX, BRANCH_ADDRESSES
                                        ;base-address of segment containing lists
                                        (把包含表的段的基地址送入段寄存器 DS)
        MOV   DS, BX
        LEA   BX, BRANCH_TABLE_1        ;base-address of list of branch addresses
                                        (把分枝表的基地址送入 BX 寄存器)
        MOV   SI, 7* TYPE BRANCH_TABLE_1
                                        ;points initially to last such entry in list
                                        (把表的末项地址送到 SI 中),
        MOV   CX, 8                     ;loop-counter allowing shifts maximum
                                        (循环计数器最多允许移位 8 次)
L:      SHL   AL,1                      ;shifts high-order AL bit into CF
                                        (把 AL 的高位移入 CF)
        JNB   NOT_YET                   ;if CF=0, routine represented by that bit
                                        ;not desired
                                        (若 CF=0,由该位代表的分枝不是所要的)
        JMP   WORD PTB [BX][SI]         ;if CF=1, transfer to procedure represented
                                        ;by most recent bit tested
                                        (若 CF=1, 把控制转移到由最近测试的位
                                        代表的分枝)
NOT_YET:SUB SI, TYPE BRANCH_TABLE_1
                                        ;adjust index register to point to "next"
                                        ;branch-address
                                        (调整变址寄存器以指向 "下一个" 分枝地
                                        址)
          LOOP    L                     ;decrement CX, if CX>0, transfer to L so
                                        ;as to shift AL and retest
                                        (把 CX 内容减 1, 若 CX>0 转移到L, 以
                                        便移位 AL,再测试)
CONTINUE_MAIN_LINE:                     ;we reach here only if no bit was set to
                                        ;indicate a desired routine
                                        (仅当没有位被设置来指出所要的分枝, 我
                                        们才到达这里)
        ROUTINE_1:
            ⋮
        ROUTINE_2:
            ⋮
```

```
ROUTINE_3:
        ⋮
PROCEDURE_SELECT ENDS
```

例2的主要流程框图如下：



当主程序要调用某些过程的时候，往往要给过程传送一些参数，这些参数可以放在存贮器中，也可以放在寄存器中。但是，在许多应用中，保留一些寄存器用于此目的是不合算的，因为要频繁地执行保护和恢复这些寄存器内容的操作。保留一些存贮器，也是很不经济的，因为这些参数只是暂时需要的。存放这种参数的理想的地方就是存贮器的一个特殊区域——堆栈，它可以很方便地存放和删除暂时不需要的数据。下面三个例子，就是用不同的方法传递参数，读者可以仔细地体会它们各自的优缺点。

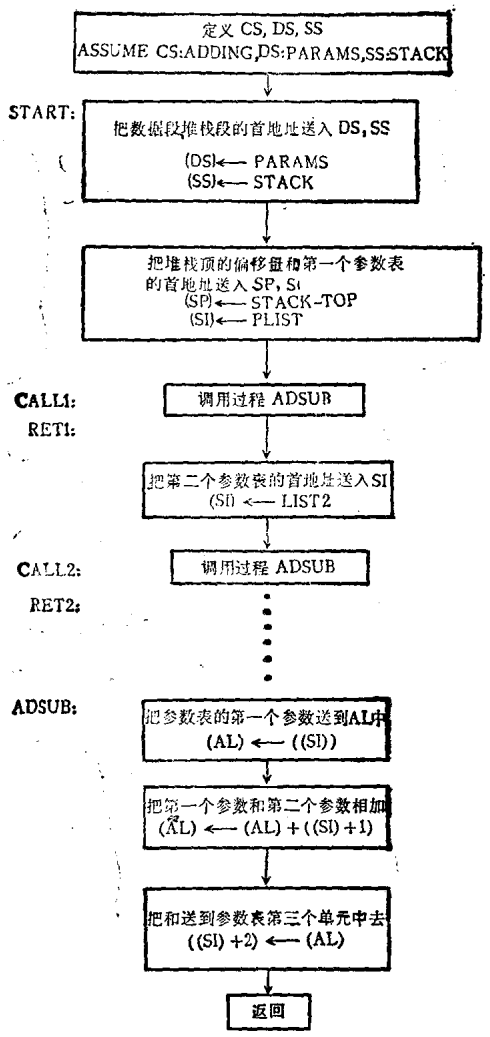
例3：

这是一个使用存贮器传递参数的例子。在这个例子中，先把要传递的参数放在某一个数据区域中，然后把该区域的首地址传送给过程。

```
PARAMS SEGMENT
PLIST   DB   6          (PLIST 是参数表的首地址)
        DB   8
        DB   ?
LIST2   DB   10         (LIST2 是参数表的首地址)
        DB   35
```

```
        DB    ?
          ⋮
PARAMS ENDS
STACK SEGMENT (定义一个堆栈段，空出四个字，STACK_TOP 指着堆栈顶地址)
      DW 4 DUP (?)
STACK_TOP LABEL WORD
STACK    ENDS
ADDING  SEGMENT
ASSUME  CS:ADDING, DS:PARAMS, SS:STACK (设定 CS, DS, SS)
START:  MOV  AX, PARAMS       (把数据段首地址装入 DS)
        MOV  DS, AX
        MOV  AX, STACK        (把堆栈段首地址装入 SS)
        MOV  SS, AX
        MOV  SP, OFFSET STACK_TOP  (把堆栈顶地址装入堆栈指示器 SP)
        MOV  SI, OFFSET PLIST (把第一个参数表的首地址送到 SI 中)
CALL 1: CALL  ADSUB
RET 1:                        (调用过程 ADSUB)
          ⋮
        LEA  SI, LIST2        (把第二个参数表的首地址送入 SI)
CALL 2: CALL  ADSUB           (调用过程 ADSUB)
RET 2:
          ⋮
ADSUB   PROC
        MOV  AL, [SI]     (把参数表的第一个参数送到 AL 中)
        ADD  AL, [SI+1]   (把第一个参数和第二个参数相加，它们的和放在 AL
                           中)
        MOV  [SI+2], AL   (把和送到参数表的第三个单元中去)
        RET
ADSUB   ENDP
          ⋮
ADDING  ENDS
        ENDSTART
```

例 3 的执行过程可用框图表示如下：



例 4：

这是一个用过程 GENAD 实现把参数相加的例子，所要相加的数组的地址，通过 BX 寄存器传送到过程，而数组中元素的个数，则通过 CX 寄存器传送到过程中去，相加的结果当过程返回时，放在累加器 AL 中。

```
INITIAL_PARAMETERS    SEGMENT
RESULT  DB  0
PARM    DB  6, 82, 13, 16
INITIAL_PARAMETERS    ENDS
GPR     EQU  GENERAL_PROCEDURES
PR1     EQU  INITIAL_PARAMETERS
GENERAL_PROCEDURES SEGMENT
            ASSUME CS:GPR, DS:PR1     (使用 EQU 语句中得到的短的等价名)
    ;The procedure is placed first, to avoid forward referencing
    ;the FAR procedure GENAD86. Note that the program start address
```

```
;is after the procedure at label "START".
```
(过程被放在开头,以避免向前引用 FAR 过程 GENAD86。注意程序的起始地址在过程之后的标号"START"处。)

```
GENAD86 PROC FAR
        PUSH SI                 ;save current value of SI on the stack
                                ;(discussed below), so that this routine
                                ;can use this registerfreely, restoring its
                                ;original contents just prior to returning
                                ;control to calling routine.
```
(把 SI 的当前值保护在堆栈中,以便在过程中可以自由地使用 SI 寄存器,在返回主程序前要恢复它的值)

```
INIT:   MOV  AL, 0              ;initialize AL to receive sum.
        MOV  SI, 0              ;initialize SI to point to first array element
```
(初始化 AL,以便接收和。初始化 SI,以指示数组的第一个元素)

```
MORE?:  ADD  AL, [BX] [SI]      ;add next array element to sum BX points
                                ;to the start of the array, and SI selects
                                ;an element of the array
```
(把下一个数组元素加入到和中去,BX 指向数组的起始地址,SI 选择数组中的元素。)

```
        INC  SI                 ;have SI index the next array element
```
(使变址器 SI 指向数组的下一个元素。)

```
        LOOP MORE?              ;continue looping until CX is zero (all
                                ;array elements have been added in to AL)
```
(进行循环直到 CX 为零。)

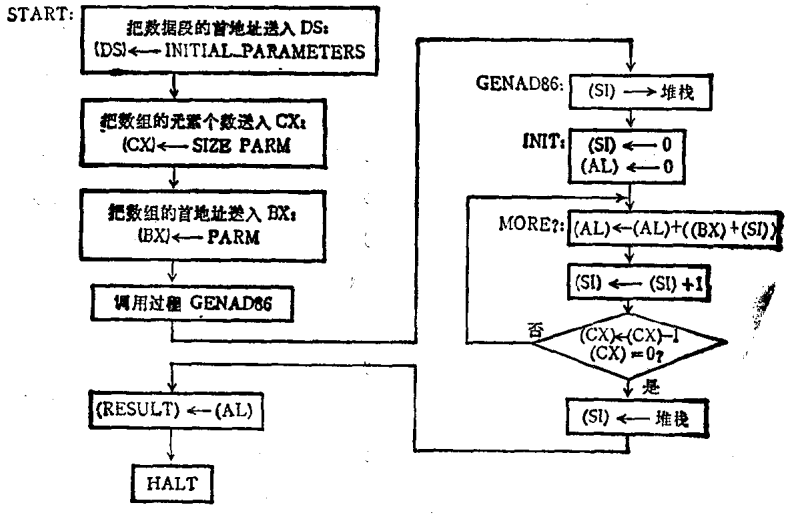
```
        POP  SI                 ;restere orginal contents of SI.
```
(恢复 SI 的原来内容)

```
        RET                     ;transfer to instruction immediately
                                ;following CALL
```
(返回到 CALL 语句后面的指令)

```
GENAD86 ENDP
    ;Program execution starts here (due to the label "start" named on the END
    ;directive below).
    ;Point DS to the INITIAL PARAMETERS segment, and call GENAD86
    ;with the array PARM.
```
(程序在这里开始执行(因为标号"Start"被在下面 END 伪指令中命名)。指定 DS 为 INITIAL_PARAMETER 段,并用数组 PARM 调用 GENAD86。)

```
START:  MOV AX, INITIAL_PARAMETERS
        MOV DS, AX
        MOV CX, SIZE PARM          ;number of elements is passed in CX
                                    (元素个数传送到 CX)
        MOV BX, OFFSET PARM        ;address of array PARM is passed in BX
                                    (把数组 PARM 的地址传送到 BX)
        CALL GENAD86
        MOV RESULT, AL             ;Sum is returned in AL
                                    (在 AL 中的结果送到结果单元)
        HLT                         ……程序结束……
GENERAL_PROCEDURES ENDS
        END START
```

例 4 的执行过程可用框图表示如下：



例 5：

在这个例子中，说明了如何使用堆栈来传递字节数参数和数组首地址。

```
params_pass   SEGMENT STACK
              DW  12  DUP (?)      ;reserve 12 words of stack space
                                    (保留堆栈空间的 12 个字)
last_word     LABEL  WORD          ;last_word is the offset of top of stack
                                    (last_word 是堆栈顶的偏移量)

params_pass   ENDS
data_items    SEGMENT
first    DB 11, 22, 33, 44, 55, 66
second   DB 4, 5, 6
third    DB 94, 88
result   DX ?
```

```
data_items    ENDS
stk_usage_xmpl SEGMENT
                ASSUME CS:stk_usage_xmpl, DS:data_items, SS:params_pass
genaddr PROC FAR
        PUSH  BP                ;save old copy of BP
                                (把老的 BP 保护到堆栈中)
        MOV   BP, SP            ;move tos to BP (see figure 1)
                                (把堆栈顶指示器值送到 BP 中(见图 1))
        PUSH  BX                ;save BX,so ok to use BX in genaddr
                                (保护 BX,使 BX 可以在过程中使用)
        PUSH  CX                ;save CX, so ok to use CX in genaddr
                                ;(figure 2)
                                (保护 CX 使 CX 可以在过程中使用 (见
                                图 2))
        MOV CX, [BP+6]          ;get count of number of bytes in array
                                (把数组中的字节计数送到 CX)
        MOV BX, [BP+8]          ;get address of array of bytes
                                (把字节数组的首地址送到 BX 中)
        MOV AX, 0               ;AX:=0, AX holds running sum in adder
                                ;loop
                                (把 AX 清 0, 在循环中 AX 存放和)
adder:  ADD AL, [BX]            ;add in the first byte
                                (加入第一个字节)
        ADC AH, 0               ;and add any carry into AH
                                (把可能有的进位加到 AH 中)
        INC BX                  ;point to next byte to be added in
                                (使 BX 指着下一个要加的字节地址)
        LOOP adder              ;CX:=CX-1; IF CX <> 0 THEN GOTO
                                ;ADDER
                                (CX:=CX-1,若CX≠0,则转到ADDER)
        POP CX                  ;The registers must be restored in the
                                (恢复 CX 寄存器)
        POP BX                  ;reverse order they were pushed.
                                (恢复 BX 寄存器)
        POP BP                  (恢复 BP 寄存器)
        RET 4                   ;return, popping off the 2 WORD
                                ;parameters
                                (返回,并弹出 2 个字的参数)
genaddr ENDP
```

```
stk_usage_xmpl ENDS
caller          SEGMENT
                ASSUME CS:caller, DS:data_items, SS:params_pass
start:  MOV AX, data_items          ;paragraph number of data segment to AX
                                    (把数据段的段号送给 AX)
        MOV DS, AX                  ;and then to DS
                                    (再把它送给 DS)
        MOV AX, params_pass         ;paragraph number of stack segment to AX;
                                    (把堆栈段的段号送给 AX)
        MOV SS, AX                  ;and then to SS
                                    (再把它送给 SS)
        MOV SP,offset last_word     ;offset of the stack_top to the SP;
                                    (把堆栈顶的偏移量送给 SP)
        MOV AX, OFFSET first        ;offset of first to AX;
                                    (把 first 的偏移量送给 AX)
        PUSH AX                     ;then onto the stack
                                    (然后送到堆栈中)
        MOV AX, SIZE first          ;number of bytes in first array to AX
                                    (把 first 数组的字节数送给 AX)
        PUSH AX                     ;then onto the stack
                                    (然后送到堆栈中)
        CALL genaddr                ;Call the far procedure
                                    (调用过程)
        MOV  result,AX
             ⋮
        MOV  AX, OFFSET second      ;same as above except doing second
        PUSH AX                     (同上,但对首址为 second 的数组执行)
        MOV  AX, SIZE second
        PUSH AX

        CALL genaddr
        MOV result, AX
             ⋮
        MOV  AX, OFFSEST third      ;same as above except doing third
        PUSH AX                     (同上,但对首址为 third 的数组执行)
        MOV  AX, SIZE third
        PUSH AX
        CALL genaddr
        MOV result, AX
             ⋮
        HLT
caller  ENDS
        END  start
```

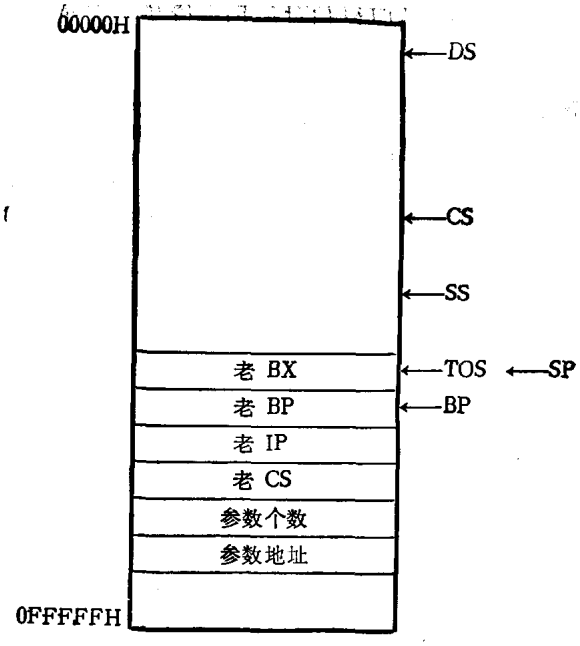


图 1

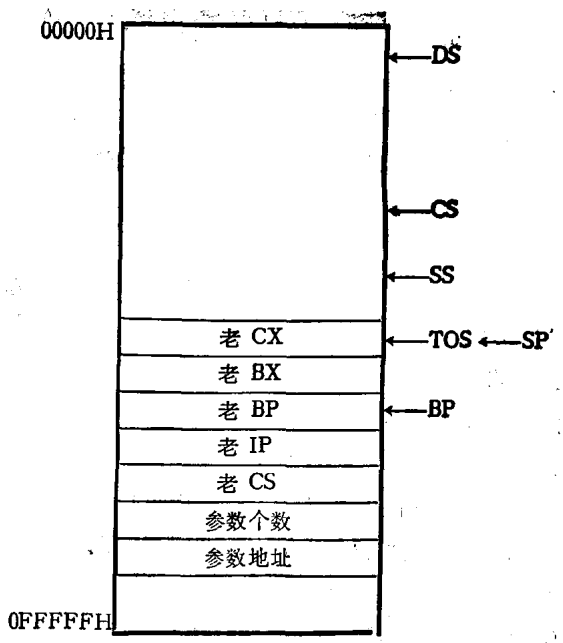
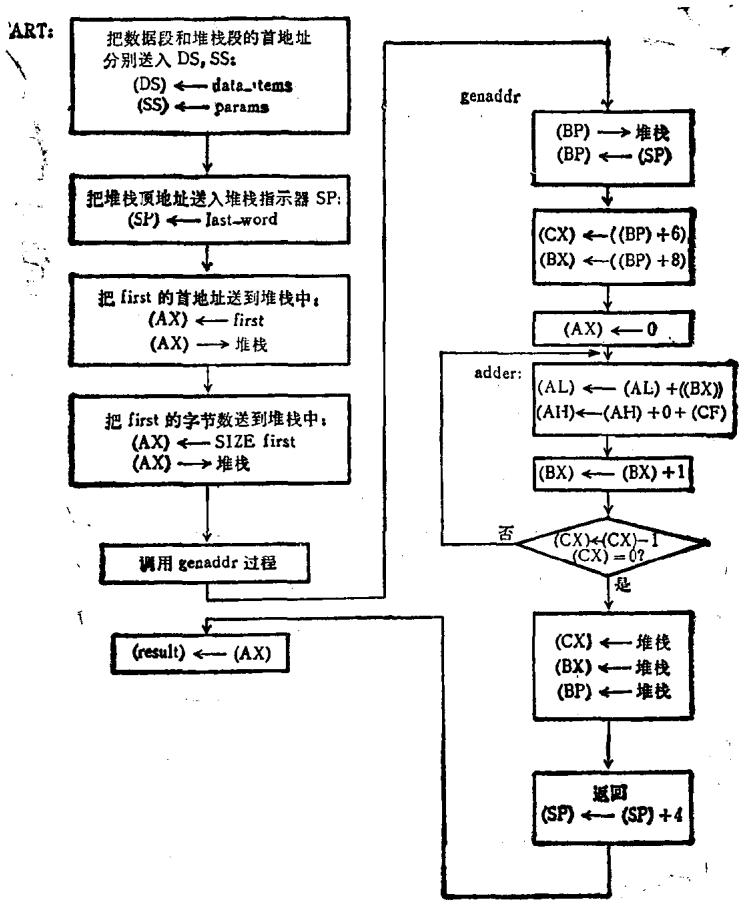


图 2

例 5 的执行过程可用框图表示如下：



读者在阅读这个程序的时候，可以用手算跟踪堆栈进退的情况，以增加对使用堆栈传递参数的感性认识。

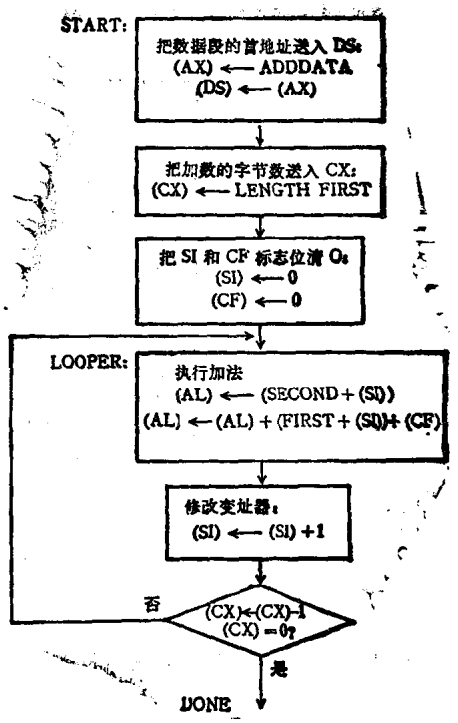
进位标志和 ADC（带进位加）指令，可以用来做任意长度的加法，若把 ADC 指令换成 SBB（带借位减）指令，就可以把做加法的程序，改成做减法的程序，例 6，例 7 就是这方面的例子。

例 6A：

这是一个做两个加数长度相等的加法的程序。

```
ADDDATA     SEGMENT
FIRST    DB  8AH, 0AFH, 32H
SECOND   DB  90H, 0BAH, 84H
ADDDATA     ENDS
MULTIBYTE_ADD  SEGMENT
ASSUME   CS:MULTIBYTE_ADD.
&        DS:ADDDATA
START:   MOV AX, ADDDATA
         MOV DS, AX
         MOV CX, LENGTH FIRST      ;Number of bytes in each addend.
                                   ;Controls of loop iterations
                                    (在每个加数里的字节数，以控制循环次数)
         MOV SI, 0
         CLC                       ;Clears any prior caary.
                                    (清除以前的进位)
LOOPER:  MOV AL, SECOND[SI]        ;Each successive byte replaces AL
                                    (每一个后续字节替代 AL 中的内容)
         ADC FIRST[SI], AL         ;Parallel byte added with carry.
                                    (对应字节带进位加)
         INC  SI                   ;Index incremented by 1
                                    (变址器加 1)
         LOOP LOOPER               ;CX=CX-1. Repeat till CX=0.
                                   ;then fall thru to DONE
                                    (CX=CX-1，重复直到 CX=0，然后到达
                                   DONE.)
DONE:
                    ⋮
MULTIBYTE_ADD  ENDS
            END  START
```

例 A6 的执行过程可用框图表示如下：



例 6B:

例 6B 是一个执行两个不等长加数相加的程序。在执行不等长加数相加的过程中，较长加数的最高字节的进位可能被丢失。这可以设置一条检查标志的指令，或在较长数上附加一个更高字节来解决。

```
ADD_DATA_2  SEGMNT
FIRST DB 11, 22, 33                (定义字节数组 FIRST)
NUM 1 DW LENGTH FIRST              (NUM 1 的内容定义为 FIRST 数组的长度)
SECOND DB 99, 88, 77, 66, 55       (定义 SECOND 字节数组)
NUM2 DW LENGTH SECOND              (NUM2 内容定义为 SECOND 数组的长度)
ADD_DATA_2  ENDS
MULTI_TWO  SEGMENT
ASSUME     CS:MULTI_TWO
&          DS:ADD_DATA_2
START:     MOV AX, ADD_DATA_2      (把数据段首地址装入 DS 寄存器)
           MOV DS, AX
;The routine determines which number is longer and stores the result there.
;The size in bytes of the smaller number controls LOOP1, i.e. where both
;numbers do have a byte of data to be added.
;The difference in size controls LOOP2, which is needed if there is a final carry
```

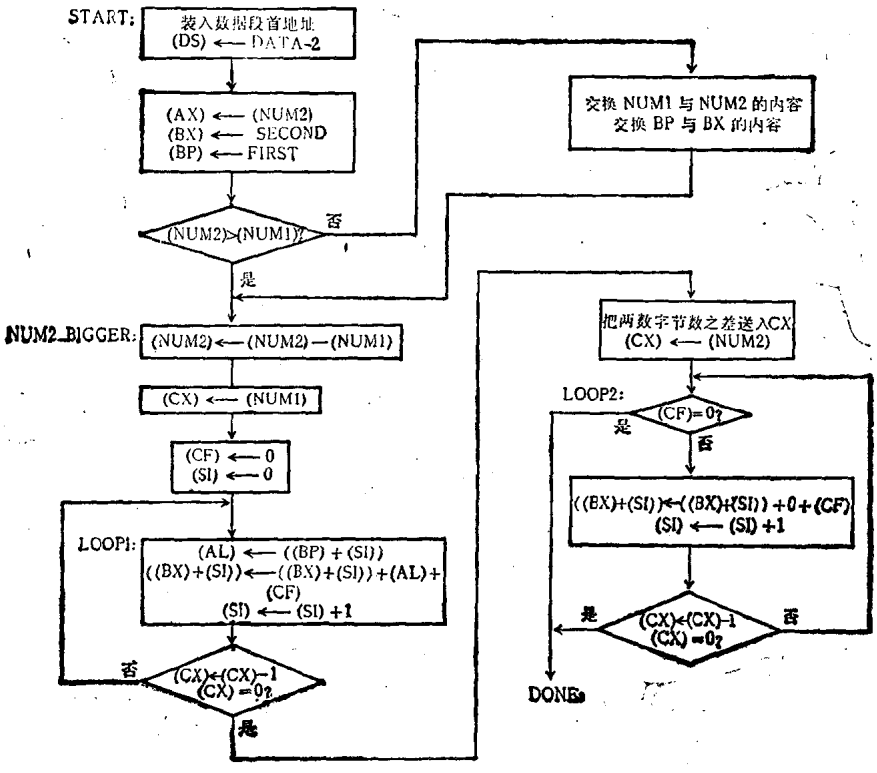
(此过程决定哪一个数较长，并把结果存放在那里，较少字节数的字节数控制LOOP1

即，在那里两个数要进行字节相加。两个数的字节数差控制 LOOP2，以处理最后的进位）

```
        MOV AX, NUM2            ;Initially assume NUM2 larger, and
                                (最初假设 NUM2 较大并)
        LEA BX, SECOND          ;give BX address of longer number
                                (把较长数的地址送给 BX)
        LEA BP, FIRST           ;BP address of shorter number.
                                (把较短数的地址送给 BP)
        CMP AX, NUM1            ;Check assumption
                                (检查假定)
        JGE NUM2_BIGGER         ;continue with values as they are unless
                                ;N2<N1
                                (若 N2<N1 则继续往下做，否则转往标号
                                NUM2_BIGGER)
        XCHG AX, NUM1           ;Switch NUM2 and NUM1, exchanging
        XCHG AX, NUM2           ;through AXNUM2 now>NUM1.
                                (通过 AX 交换 NUM2 和 NUM1，至此，
                                NUM2>NUM1)

        XCHG BX, BP             ;Must also now switch addresses referred
                                ;to, so that number of bytes still
                                ;corresponds with correct number, and
                                ;sum goes to longer place.
                                (交换引用地址以使字节数仍与正确的数相
                                对应并且和能放到较长的数的单元中)
NUM2_BIGGER: MOV CX, NUM2
        SUB CX, NUM1            ;NUM2 now gets difference of sizes. Use
                                ;smaller number of bytes for control add.
        MOV NUM2, CX
        MOV CX, NUM1            (NMU2 单元中现在存放着字节数之差使
                                用较小数的字节数控制相加次数)

        CLC                     ;Clear carry of possible prior setting.
                                (清除事先可能设置的进位)
        MOV SI, 0               ;Initialize index to bytes of addends. Then
                                ;SI=SI+1.
                                (初始化 SI)
LOOP1:  MOV AL, DS:[BP][SI]     ;Get byte of shorter number
                                (取得较小数的字节)
```

```
        ADC   [BX][SI], AL      ;Add it to relevant byte of longer number.
        INC   SI                ;Then SI=SI+1
                                (把它和较长数的相应字节带进位相加，然
                                后把 SI 的内容加 1)

        LOOP  LOOP1
        MOV   CX,  NUM2         ;Number of bytes yet unused in longer
                                ;number
                                (把在较长数中的未使用字节数送入 CX)
LOOP2:  JNB   DONE              ;If no carry, CF=0, then done.
                                (若进位 (CF)=0，则转到 DONE)
        ADC BYTE PTR [BX][SI], 0 ;Add carry to remaining bytes
        INC   SI                ;of longer number. Then SI=SI+1;
                                (把进位加到剩余的字节，然后把 SI 的内容
                                加 1)

        LOOP  LOOP2
DONE:
              :
MULTI_TWO  ENDS
           END  START
```

例 6B 的执行过程可用框图表示如下：



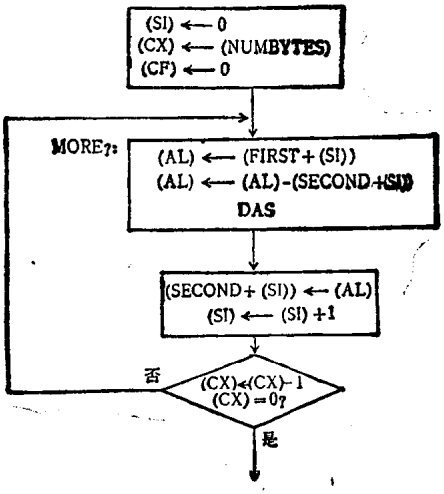
请读者找出哪里可能会丢失最高字节的进位，并设法修改之。

例 7：

这是一个 8086 过程的核心部分，它对于压缩 10 进制数实现减法。

```
        MOV  SI, 0                    (把 SI 清 0)
        MOV  CX, NUMBYTES             (把字节数送入 CX)
        CLC                           (清除进位位)
MORE?:  MOV  AL, FIRST[SI]            (把字节数组 FIRST 的字节送入 AL)
        SBB  AL, SECOND[SI]           (把 AL 的内容减去 SECOND 数组的相应
                                       字节的内容，并把结果送到 AL 中)
        DAS                           (10 进制减法调整)
        MOV SECOND[SI], AL            (把调整后的十进制结果送到 SECOND 数
                                       组的字节单元中去)
        INC  SI
        LOOP MORE?
```

例 7 的执行过程可用框图表示如下：



例 8：

这是一个使用 8086 中断过程的例子。假定在应用于某种控制的 8086 系统中设置了 6 种中断处理过程，其中 4 种是传感器，2 种是警报器，每种设备都可以向 8086 提出外部中断。下面所示的中断处理过程是任意的，即并不是固定在 8086 中，而是可以依假定的应用而改变的。这段程序仅是举例说明了处理各种重要和紧急情况的若干种可能性。

```
ASSUME CS:INTERRUPT_PROCEDURES, DS:DATA_VAR
DEVICE_1_PORT     EQU  0F000H
DEVICE_2_PORT     EQU  0F002H
DEVICE_3_PORT     EQU  0F004H
DEVICE_4_PORT     EQU  0F006H
WARNING_LIGHTS    EQU  0E000H
```

```
CONTROL_1          EQU  0E008H
                   EXTRN CONVERT_VALUE:FAR
                                        ;Positioning this EXTRN here indicates
                                        ;that CONVERT_VALUE
                                        ;is outside of all segments in this module.
                                        (在这里放上 EXTRN 表明 CONVERT_
                                        VALUE 是在这个模块的所有段之外)

INTERRUPT_PROC_TABLE SEGMENT BYTE AT 0
                   ORG 08H              (规定从该段的第 8 个单元开始)
                   DD  ALARM_1          ;non_maskable interrupt type 2
                                        (不可屏蔽中断类型 2)
        ;One 64K area of memory contains pointers to the routines that handle
        ;interrupts. This area begins at absolute address zero. The address for the routine
        ;appropriate to each interrupt type is expected as the contents of the double
        ;word whose address is 4 times that type. Thus the address for the handler of
        ;non-maskable-interrupt type 2 is stored as the contents of absolute location 8.
        ;These addresses are also called interrupt vectors since they point to the
        ;respective procedures.
        (存贮器的一个 64K 区域含有处理中断的过程的指示器,这一区域开始于绝对地址
        0。每种中断类型的处理过程的入口地址放在地址为类型数乘以 4 的双字中，这样
        非屏蔽中断类型 2 的处理过程的地址是在第 8 个单元开始的双字中。这些地址也
        称为中断向量,因为它们指向各自的过程)
        ORG 80H                         (自 80H 开始放中断处理过程的入口地址)
        ;the first 32 interrupt types (0—31) are defined or reserved by INTEL for
        ;present and future uses. (See the 8086 User's Manual for more detail.)
        ;User-interrupt type 32 must therefore use location 128(=80H) for its interrupt
        ;vector.
        (前 32 种中断类型是 INTEL 保留的，用户中断类型 32 必须用单元 128 (=80H)
        来放中断向量)
          DD ALARM_2                    ;INTERRUPT TYPE32
          DD DEVICE_1                   ;INTERRUPT TYPE33
          DD DEVICE_2                   ;INTERRUPT TYPE34
          DD DEVICE_3                   ;INTERRUPT TYPE35
          DD DEVICE_4                   ;INTERRUPT TYPE36
                                        (把中断处理过程的入口地址放入向量表)
INTERRUPT_PROC_TABLE ENDS
DATA_VAR SEGMENT PUBLIC
EXTRN  INPUT_1_VAL:BYTE, OUTPUT_2_VAL:BYTE,
```

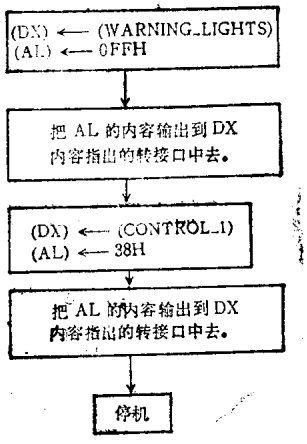
```
&       INPUT_3_VAL:BYTE, INPUT_4_VAL:BYTE
EXTRN   ALARM_FLAG:BYTE, INPUT_FLAG:BYTE
        ;The names above are used by 1 or more of the procedures below, but the
        ;location or value referred to is located (defined) in a different module. These
        ;EXTeRNal references are resolved when the modules are linked together,
        ;meaning all addresses will then be known. Declaring these EXTRNs here
        ;indicates what segment they are in.
        (上面的名被下面的一个或多个过程使用,但它们的位置或值却放在另一个模块中,
        这些外部引用,在这些模块被连接起来时就消除了,那时所有的地址都知道了。在
        这里说明 EXTRN 是为了指出它们在哪些段中)
DATV_VAR ENDS
        ;The names below are defined later in this module. The PUBLIC directive makes
        ;their addresses available for other modules to use.
        (下面的名在这个模块的后面定义。PUBLIC 伪指令使它们的地址可以为其它的模
        块使用。)
PUBLIC ALARM_1, ALARM_2, DEVICE_1, DEVICE_2, DEVICE_3,
&       DEVICE_4
INTERRUPT_PROCEDURES SEGMENT
ALARM_1 PROC FAR
        ;The routine for type 2, "ALARM_1" is the most drastic because this interrupt
        ;is intended to signal disastrous conditions such as power failure. It is non-
        ;maskable, i. e., it cannot be inhioitld by the CLear Interrupts (CLI) instruction.
        (关于类型2"ALARM_1"的过程是最紧急的,因为这种中断要通知一种灾难性的
        情况,诸如掉电。它是不可屏蔽的,即它不能用清除中断 (CLI) 指令来禁止。)

        MOV DX, WARNING_LIGHTS
        MOV AL, 0FFH
        OUT DX, AL                      ;turn on all lights
                                        (打开所有的灯)
        MOV DX, CONTROL_1
        MOV AL, 38H
        OUT DX AL                       ;turn off machine
                                        (关掉机器)
        HLT                             ;stop all processing
                                        (停止全部处理)
ALARM_1 ENDP
ALARM_2 PROC FAR
        PUSH DX
        PUSH AX
```

```
                    :
        MOV  DX, WARNING_LIGHTS
        MOV  AL, 1                  ;turn on warning light*1 to warn operator
                                    ;of device
                                    (打开警告灯*1以警告设备的操作
        OUT  DX, AL                 者)
        MOV  ALARM_FLAG, 0FFH       ;set alarm flag to inhibit later processes
                                    ;which may now be dangerous
                                    (设置警告标志以禁止以后可能有危险的处
        POP  AX                     理)
        POP  DX
        IRET                        ;return from interrupt this restores the
                                    ;flags and returns control the interrupted
                                    ;instruction stream
                                    (中断返回,这条指令恢复标志,把控制返回
                                     到被中断的指令流)
ALARM_2 ENDP
DEVICE_1 PROC
        PUSH DX
        PUSH AX
        MOV  DX, DEVICE_1_PORT
        IN   AL, DX                 ;get input byte from device_1
                                    (从设备1取得输入字节)
        MOV  INPUT_1_VAL, AL        ;store value
                                    (存放值)
        MOV  INPUT_FLAG, 2          ;this may alert another routine or device
                                    ;that this interrupt and input occurred
                                    (这条指令可以警告另一个过程或设备这种
                                    中断和输入发生了.)
        POP  AX
        POP  DX
        IRET
DEVICE_1 ENDP
DEVICE_2 PROC
        PUSH DX                     ;when this interrupt-type occurs.
        PUSH AX                     ;the action necessary is to notify device_
                                    ;2-port of the event
                                    (当这种类型的中断生时, 必要的动作是把
```

```
                                        这件事通知 device-2-port 转接口)
        MOV  AL, OUTPUT_2_VAL           ;get value, to output to device_2_port
        MOV  DX, DEVICE_2_PORT          (取得值,并向 device-2-port 转接口输出)

        OUT  DX, AL
        POP  AX
        POP  DX
        IRET
DEVICE_2 ENDP
DEVICE_3 PROC
        PUSH DX                         ;when a device_3 interrupt occurs
        PUSH AX                         ;only the lower byte at the port is
                                        ;of value
                                        (当一个 device_3 中断发生时，在转接口
        MOV  DX, DEVICE_3_PORT          只有最低字节具有值)
        IN   AL, DX
        AND  AL, 0FH                    ;mask off top four bits
                                        (屏蔽高 4 位)
        MOV  INPUT_3_VAL, AL            ;store value for use by later routines in
        POP AX                          ;another module
                                        (把值存放好以备在另一模块中的程序使
                                        用)

        POP DX
        IRET
DEVICE_3 ENDP
DEVICE_4 PROC                           ;a device_4 interrupt provides
        PUSH DX                         ;a value which needs immediate device_4
        PUSH CX                         ;conversion by another procedure
        PUSH AX                         ;before this interrupt-handler can
                                        ;allow it to be used at input_4_val
                                        (device_4 中断提供一个值,该值在能被中
                                        断处理程序用于 input_4_val 之前,需要
        MOV  DX,DEVICE_4_PORT           由另一个过程加以立即转换.)
        IN   AL,DX
        MOV  CL, AL
        CALL CONVERT_VALUE              ;converts input value in CL to new result
        MOV  INPUT_4_VAL, AL            ;in AL and saves that result in input_
                                        ;4_val
        POP  AX                         (把 CL 中的输入值转换为在 AL 中的新
```

```
        POP   CX                    值，并把该结果保护到 input_4_val 中去)
        POP   DX
        IRET
DEVICE_4 ENDP
INTERRUPT_PROCEDURES ENDS
        END
```

例 8 ALARM_1 过程的框图可以表示如下：



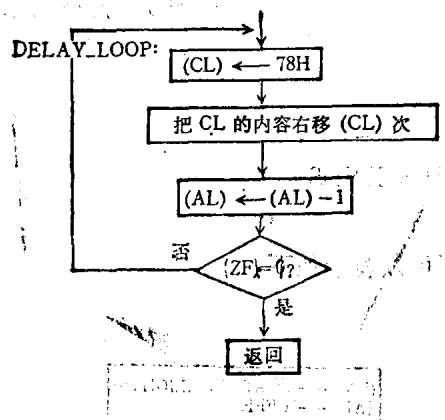
读者可以试着自己画出其它几个过程的执行框图。

例 9：

这是一个时间循环的例子，延迟时间的数量由在 AL 寄存器中传送的字节参数设置，时间是参数 *100 微秒，这里假定 8086 的主频是 8MHZ。

```
ASSUME CS: TIMER_SEG
TIMER_SEG SEGMENT
TIME PROC
DELAY_LOOP: MOV  CL, 78H          ;shift count for supplying
                                   (把移位计数送入 CL)
            SHR  CL CL            ;proper delay via SHR countdown
                                   (通过 SHR 进行规定的延迟)
            DEC  AL               ;decrement timer count
                                   (把时间计数器减 1)
            JNZ  DELAY_LOOP
            RET
TIME        ENDP
TIMER_SEG   ENDS
            END
```

例 9 的程序执行过程可用框图表示如下：



下面的几个例子，解释了在执行期间 SDK86 系列的监督程序和键盘、显示器通讯时，所使用的过程的类型。

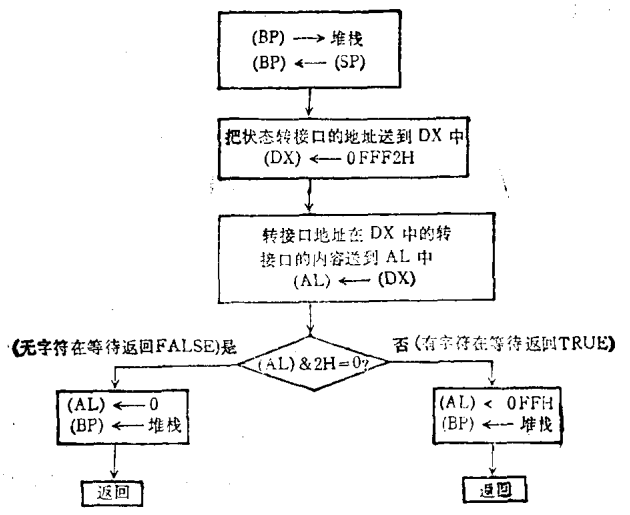
例 10：

SIO_CHAR_RDY，测试是否有一个输入字符正在等待处理。

```
SIO_CHAR_RDY PROC NEAR
    PUSH  BP                    ;save old value
                                (保护老值)
    MOV   BP, SP
    MOV   DX, 0FFF2H            ;address of status port to DX
                                (状态转接口的地址送入 DX)
    IN    AL, DX                ;input from status port
                                (从状态转接口中输入)
    TEST  AL, 2H                ;is read-data-ready line=1, i.e., character
                                ;pending?
                                (测试 read-data-ready line 是否为 1 即是
                                否有字符在等待着?)
    JNZ   @1                    ;if so, retrun TRUE;
                                (若是,就返回 TRUE,转到@1)
    MOV   AL, 0                 ;if not, return FALSE: AL=0
                                (若不是,就返回 FALSE: AL=0)
    POP   BP                    ;restore old value
                                (恢复老值)
    RET                         ;done no char waiting
                                (返回,没有字符在等待.)
@1:
    MOV   AL, 0FFH              ;return TRUE:AL=all ones
                                (返回 TRUE: AL=全 1)
    POP   BP                    ;restore old value
                                (恢复老值)
```

```
    RET                                ;done, char is waiting;
                                       (返回,有字符在等待着.)
SIO_CHAR_RDY ENDP
```

例 10 的执行过程可用框图表示如下：



例 11：

这个程序是上面一个程序的另一种写法，在这里对于上面直接指定的一些数用了一些名来代替，这些名在后面的例子中还会用到。使用这些名可以使该过程更易读和易懂。

```
    TRUE            EQU  0FFH
    FALSE           EQU  0H
    STATUS_PORT     EQU  0FFF2H
    DATA_PORT       EQU  0FFF0H
    ASCII_MASK      EQU  7FH
    CONTROL_S       EQU  13H
    CONTROL_Q       EQU  11H
    CARR_RET        EQU  0DH
SIO_CHAR_ROY2   PROC   NEAR
        PUSH  BX                   ;save old BX value
                                   (保护老的 BX 值)
        MOV   BL, TRUE             ;prepare for one result
                                   (为一个结果作准备)
        MOV   DX, STATUS_PORT      ;check the facts
                                   (检查字符是否)
        IN    AL, DX               ;char waiting???
                                   (正在等待?)
        TEST  AL, 2H               ;if 2nd bit ON, char is waiting
                                   (若第 2 位置位,则字符在等待)
        JNZ   RESULT               ;hence skip over FALSE set-up
```

```
                                    (若是在等待,则跳过 FALSE 的设置)
        MOV   BL, FALSE             ;here if 2nd bit was OFF. hence no cha
                                    ;waiting
                                    (这里,若第2位是0,则设置 FALSE)
RESULT: MOV   AL BL                 ;AL receives whichever result
                                    (AL 取得两种结果中的一种)
        POP   BX                    ;restore old BX value
                                    (恢复 BX 的值)
        RET                         (返回)
SIO_CHAR_RDY2   ENDP
```

读者可以试着自己画出这个程序执行的框图。

例 12:

这个程序输出一个字符,除非 SIO_CHAR_RDY 报告,有一个输入字符在那儿要被先处理。

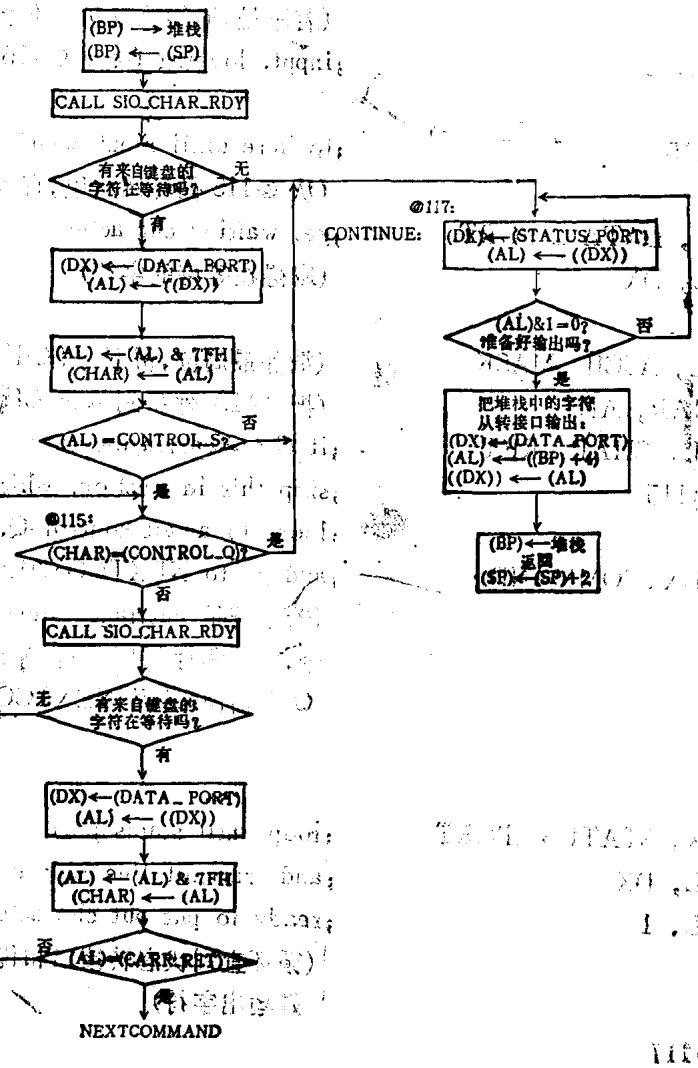
```
SIO_OUT_CHAR PROC NEAR
        ;This routine outputs an input parameter to the USART output port when
        ;USART is ready for; output transmit buffer empty. The input to this routine
        ;is on the stack.
        (当 USART 转接口的输出缓冲器空时,这个程序把一个输入参数输出到 USART
        输出转接口。这个程序的输入是在堆栈中。)
        PUSH  BP                    (保护者的 BP 值)
        MOV   BP SP
        CALL SIO_CHAR_RDY           ;keyboard input pending?
                                    (键盘输入正在等待吗?)
        RCR   AL 1                  ;put low-bite into CF to test if no input
        JNB   @117                  ; char waiting from keyboard, go to
                                    ;output loop
                                    (把低位放入 CF,以测试是否没有来自键
                                    盘的字符正在等待着,若没有就转到 @117
                                    输出循环)
        MOV   DX, DATA_PORT         ;char waiting, get it
        IN    AL, DX                ;char to AL from that port strip off high
                                    ;bit, leaving
        AND   AL, ASCII_MASK        ;ASCII code
                                    (字符正在等待,就从该转接口中把字符取
                                    到 AL 中来,并清除最高位,留下 ASCII 码)
        MOV   CHAR, AL              ;save char
                                    (把字符保护到 CHAR 单元中)
        CMP   AL, CONTROL_S         ;is char control_S?
```

```
                                    (字符是 CONTROL_S 吗?)
        JNZ   @117                  ;if this halt-display signal is not rec'd,
                                    ;continue output at @117
                                    (若这个信号不是 CONTROL_S，就转到
                                    @117)
@115:
                                    ;if control-S rec'd, must await its release
                                    (若是 Control-S，必须要等到它释放)
        CMP   CHAR, CONTROL_Q       ;Control-Q received?
                                    (取得的是 CONTROL_Q 吗?)
        JZ    @117                  ;if this continuation-signal rec'd, to do
                                    ;next output
                                    (若是,转 @117 做下一个输出)
        CALL  SIO_CHAR_RDY          ;keep checking for new keyboard
                                    (若不是,继续检查新的键盘输入)
        RCR   AL, 1                 ;input, looping from @115

        JNB   @115                  ;to here until input waiting
                                    (从 @115 循环到这里,直到有输入在等待)
        MOV   DX, DATA_ PORT        ;get waiting character
        IN    AL, DX                (取得在等待的字符)

        AND   AL, ASCII_MASK        (清除最高位,留下 ASCII 代码)
        MOV   CHAR, AL              (把 (AL) 保护到单元 CHAR)
        CMP   AL, CHAR_RET          ;if char=carriage-return,
        JNZ   @115                  ;skip this instruction, which
                                    ;loops to await control-Q,
        JMP   NEXTCOMMAND           ;and go to NEXTCOMMAND
                                    (若字符是 carriage-return，则跳过这条指
                                    令,它使程序循环,一直等到 CONTROL_
                                    Q 来临,并转到 NEXTCOMMAND)
@117:
CONTINUE:
        MOV   DX, STATUS_PORT       ;loop until status port
        IN    AL, DX                ;and transmit line indicate
        TEST  AL, 1                 ;ready to put out character
                                    (循环直到状态转接口和传输线表明已准备
                                    好输出字符)
        JZ    @117
```

```
        MOV   DX, DATA_PORT      ;output port address to DX
                                  (输出转接口地址送到 DX)
        MOV   AL, [BP]+4         ;character from stack to AL
                                  (把字符从堆栈送到 AL 中去)
        OUT   DX, AL             ;output character in AL through
                                  (通过 AL 输出字符)
        POP   BP                 ;restore original BP value
                                  (恢复原来的 BP 值)
        RET   2                  ;repositions SP behind prior parameter
                                  (在前面的参数之后,再设置 SP)。
SIO_OUT_CHAR   ENDP
```

在这个程序中调用了例 10 中的过程。

例 12 的执行过程可用框图表示如下：



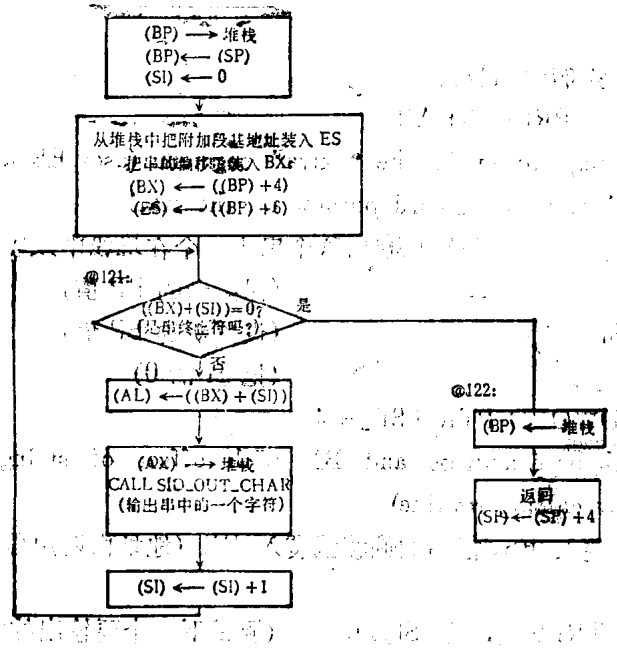
例 13：

这是一个输出整个字符串的过程。

```
SIO_OUT_STRING    PROC NEAR
        ;Outputs a string stored in the "extra" segment (uses ES as base). the string
        ;being pointed to by a 2-word pointer on the stack
        (输出一个存放在"附加"段中的串,这个串由一个在堆栈中双字指示器指出
        PUSH  BP                          (保护老的 BP 值)
        MOV   BP, SP                      (设置新的 BP 值)
        MOV   SI,0                        (把 SI 清 0)
        LES   BX, DWORD PTR [BP]+4
        ;load ES with base address and BX with offset of string (addresses pushed
        ;onto stack by calling routine)
        (把段基地址装入 ES,把串的偏移量装入 BX (地址由调用程序推入堆栈))
@ 121:
        CMP BYTEPTRES: [BX] [SI], 0       (测试下一个要输出的字符是否为全 0)
                                          ;terminator character is ASCII null=all
                                          (终止字符是 ASCII null=全 0 吗?
        JZ    @122                        ;zeroes if done, exit
                                          (若是全 0,就终止)
        MOV AL, BYTEPTRES: [BX][SI]       ;put next char on
                                          (把下一个字符送到 AL 中)
        PUSH AX                           (把 AX 推入堆栈,为调用过程作准备)
        CALL SIO_OUT_CHAR                 ;stack for output by this called procedure
                                          (把在堆栈中的字符输出)
        INC   SI                          ;point index to next char
                                          (使变址器指向下一个字符)
        JMP   @121
@ 122:
        POP   BP                          (恢复 BP 老值)
        RET   4                           ;alter return, resets SPbehind former
                                          ;parameters
                                          (返回,并使 SP 复位到以前参数的后面)
SIO_OUT_STRING    ENDP
```

注意，例 13 中调用了例 12 的过程来输出串中的每一个字符。

例 13 的执行过程可用框图表示如下：



习　　题

6.1 下列标识符哪些是有效的？哪些是不合法的？并说出理由。

John,	GAMMA,	8DGL,	X3.9
DJS-053,	PL/I,	?A3,	$NUM
A+B,	8086,	PUBLIC,	SJTU

6.2 指出并纠正下面这个程序段中的错误：

```
MY-DATA:    SEGMENT
A           DB      '8', '7', '5', '4' '9'
B           DB      '6'
C           DB      LENGTH (A) DVP [?]
MY_CODE     END
MY_CODE     SEGMNT
            ASSUME  CS:MY_DATA DS:MY_CODE
GO1:        MOV     AL, MY_DATA
            MOV     DS, AL
            CLD
            MOV     SI, OFFSET A
            MOV     DI, OFFSET C
            MOV     CX, LENGTH A
            AND     B, OFH
            MOV     BYTE PTR (DI), 0
CYCLE       LODS    A
            AND     AX, OFH
```

```
                MUL     AX, B
                AAM
                ADD,    , [DI]
                AAA
                STOS    C
                MOV     [DI], AH
                JCXZ    CYCLE
                HLT
                END     GO!
MY_DATA         ENDS
```

用 MCS-86 汇编语言编写程序，解决下列问题：

6.3 将 AX 中的 16 进制数转换为 ASCII 码，并将结果放到某个存贮区域中（4 个字节）。比如说，设 AX=2A49H，则执行程序后，使存贮器中某 4 个字节的内容成为：39H，34H，41H，32H，

6.4 将两个非紧凑的 BCD 串加起来。例如：在数据段中有这样两个变量说明语句：

```
STRING_1        DB      '1', '7', '5', '2'  (值为 2571)
STRING_2        DB      '3', '8', '1', '4'  (值为 4183)
```

把它们加起来（按位分别相加，注意需作 10 进制调整）、结果放到 STRING_2 中。

6.5 写一程序，完成任务

$$C \leftarrow \sum_{i=1}^{n} A_i * B_i$$

设所有变量 C、A_i、B_i 都放在存贮器当中，n 存放在存贮器单元 N 中。要求在执行过程中不破坏 N、A_i、B_i 的内容。

6.6 编一程序，计算

$$P(X) = a_n X^n + a_{n-1} X^{n-1} + \cdots + a_1 X + a_0$$

其中系数 a_i、参数 X 及值 n 都存放在主存中，结果 P(X) 也要求送到存贮器中去。

提示：利用关系

$$P(X) = ((\cdots (a_n X + a_{n-1}) X + \cdots + a_2) X + a_1) X + a_0$$

6.7 给定一字向量 A_1，A_2，…A_n，试写一程序将它们按大小排序，其中 n 存放在字节变量 N 当中。要求不另占存贮单元。

6.8 假定有 0-100D 之间的数 50 个，存放在一个以 GRADE 为起始地址的 50 个单位的字数组中。GRADE+i 的内容代表学号为 i+1 的学生成绩。要求建立一个以 RANK 为起始地址的 50 字数组，其中 RANK+i 的内容是 50 人的班级中学号为 i+1 的学生的名次（一个学生的名次等于成绩高于这个学生的人数加 1）。试编写这段汇编语言程序。

第七章　8086 的高级语言程序设计

§7.1　谁需要高级语言

8086 的程序可以用下面几种方法编写：一是利用机器语言（求出每条指令的 2 进制编码），这种方法很枯燥，很不方便，又容易出错，而且实现起来也比较困难，所以在实际应用中用得比较少；第二种方法就是利用汇编语言来编制程序，程序员只要从概念上理解指令，利用汇编程序生成2进制目标代码，总的看来，这种方法还是可以接受的；第三种方法就是利用所谓的高级语言来设计程序，这时程序员可以把注意力集中在对问题的理解与分析上，而利用编译程序把高级语言翻译成汇编语言最终得到 2 进制目标代码，这种方法比较易于为大多数程序员所接受。但这并不意味着不存在喜欢用汇编语言设计程序的程序员，他们往往为自己能够将一段 1025 字节的程序，压缩成 1024 字节而感到其乐无穷。确实，当感到存贮器空间紧张时，我们非常需要这样的程序员，因为他们惯于精打细算。但是我们常常需要的（尤其是在现在和将来）是尽可能快地编制和调试大型程序，而不是尽可能少用几个字节来完成这些任务。正是为了满足这种需要，引进了高级语言和编译程序。

高级语言除了具有上述优点外，高级语言程序的支持者还经常这样来美化高级语言程序：

1．高可靠性——这种程序正确完成任务的概率较大。

2．易于维护——对程序做些修改的困难不会很大。

3．自成文件——易于阅读、理解。

我们准备用一个程序作为例子，来说明这些优点，该程序要求的是大于 100 且能被 3 除的最小数字，很显然，答案应该是 102，但让我们来看看 8086 是如何求得该结果的。

解决这个问题的一种方法是这样的，从 0 开始，重复加 3 直到结果超过 100，即可解决，开始置字节 X 为 0，当 X 小于或等于 100 时，重复加 3 到 X。

利用高级语言，程序员可以直接从上面对解的文字描述出发写出程序，若用 PL/M-86 程序设计语言，写出的程序是这样的：

```
DECLARE X BYTE;
X=0;
DO WHILE X<=100;
    X=X+3;
END;
```

值得注意的是在编写这段程序时，根本就没有考虑 8086 的指令集。但这段程序应被送到编译程序，由编译程序为我们产生 8086 的指令。

现在我们再利用汇编语言来编写这段程序，这就需要把对问题解的描述转换成对应于 8086 指令的一些步序：

1．为 X 保留一个存贮器字节；

2．送一个 0 到该字节中去；

3．把 X 中的值与 100 相比较；

4. 如果比较的结果指出 X 大于 100，那就转移到程序的终点；
5. 加 3 到 X；
6. 跳回到第 3 步；
7. 程序结束。

这样我们就能用 8086 的汇编语言 ASM 86 来写出该程序了：

```
1. X        DB    ?
2.          MOV   X, 0
3. CYCLE:   CMP   X, 100
4.          J..   DONE
5.          ADD   X, 3
6.          JMP   CYCLE
7. DONE:    HLT
```

其中，第 4 步还没有写完整，这是因为我们还吃不准它到底该是什么，我们所希望的是若 X 超过 100 则跳转，前一步(第 3 步)是把 X 与 100 比较，由于比较意味着从 X 中减去 100(目减源)，也就是说如果 X 大于 100，则在做了减法之后得到的是一个大于 0 的数，因此第 4 步似乎应是 JG (大于则转)指令。另一方面，我们也许会把比较指令 "CMP X, 100" 理解成从 100 中减去 X ，这样一来，如果 X 大于 100，得到的就将是一个小于 0 的数，这时，第 4 步好象应该是 JL (小于则转)指令，究竟何者是正确的呢？选对指令的概率为 0.5。但是，如果使用高级语言，我们就不会被这种类型的问题所困扰了。实际上，第 4 步应为 JA 指令。

为了公平地评价汇编语言的程序设计，让我们对上面这段程序做些修改。首先，在第一次执行到第 4 步的时候，跳转不会发生，所以我们可以把第 3 步及第 4 步移到循环的末尾，移去第 6 步，从而省掉了一条指令。

```
1. X          DB    ?
2.            MOV   X, 0
3. REPEAT:    ADD   X, 3
4.            CMP   X, 100
5.            JNA   REPEAT
6.
7.            HLT
```

为了进一步优化该程序，我们所能做的另一件事情就是使用 AL 来代替存贮器字节 X，这就使第 2、3、4 步的三条指令各缩短了一个字节，此外，还释放了分配给 X 的那个字节。

```
1.
2.            MOV   AL, 0
3. REPEAT:    ADD   AL, 3
4.            CMP   AL, 100
5.            JNA   REPEAT
6.
7.            HLT
```

假如我们既没有编译程序，也没有汇编程序，那末我们就必须以 2 进制数的形式来编写所

有的指令了。作为例子，请看第5步的那条指令“JNA REPEAT”，它是一条两个字节的指令，其中第一个字节的内容为01110110，第二个字节必须给出为了到达 REPEAT 所要跳过的字节数，这就意味着我们必须知道各条指令的长度，还要求出第3步到第6步指令之间所占用的字节数目。倘若你喜欢用机器语言设计程序的话，那你就把这个问题作为练习吧！

至此，我们已经粗略地介绍了8086的三种程序设计方法，现在该作出采用何种方法这一抉择了。假如你仍然极其信奉2进制或汇编语言程序设计，而对高级语言程序设计不太感兴趣，那末本章的剩下部分对你来讲也就不会有什么诱惑力，当然，我们也不能强迫你读下去。但是如果你还相信高级语言可能会对你有所裨益的话，那就请继续往下读吧。

在实际应用当中，最通用的高级语言，也许要算 COBOL、BASIC、FORTRAN，COBOL 是用于商业数据处理的一种通用语言；FORTRAN 常常用在包含数字运算的应用当中；而 BASIC 则由于其简单性和人机对话的特色，而成为微型计算机业余爱好者所最喜爱的一种语言，BASIC 程序往往不需要先编译成机器语言程序就可以直接执行；此外还有一种语言也值得一提，它就是 PASCAL 语言，其设计目的虽然主要是为了教学，但它近来也获得了广泛的赞许。

Intel 的 PL/M 语言是第一个主要面向微机应用的高级语言，它也是可用于8086的第一种程序设计语言（甚至是在8086汇编语言之前）。本章我们将较为详细地讨论 PL/M 的8086方言——PL/M-86。

§7.2 PL/M-86 程序的结构

首先，请看一下第一个 PL/M-86 程序例子。

```
1. PROG:
2.        DO;  /* add inputs divided by 2 until totol exceeds 100 */
                  (输入数据除2，再加起来，直到和数超过100)
3.           DECLARE SUM BYTE;
4.           SUM=0;
5.           DO WHILE SUM<=100;
6.              SUM=SUM+INPUT(3)/2;
7.           END;
8.           OUTPUT(3)=SUM;
9. END PROG;
```

即使不了解PL/M-86，我们也能够读且差不多能够理解上面这段程序。第1行似乎告诉我们该程序的名字是 PROG，第9行好象证实了这一点，第2行大概是说想做 (DO) 某个事情，第9行肯定是讲我们刚把事情做完 (END)。第2行中还带有一段英文(注释)，它告诉我们下面做了些什么。第3行是在预定一个存贮器字节，其名字为 SUM，这个字节也许在数据段中，直到这里还没有产生任何码子，第4行看起来好象是第1条指令——把0送到 SUM，第5行到第7行之间似乎有联系，它们从 DO 开始到 END 结束，好象是在重复地从3号输入转接口读进数值，这些输入值除以2之后加到 SUM 中去，这一重复动作直到 SUM 超过100时才告完成，最后，即第8行也许是把 SUM 写到3号输出转接口。

从这个例子出发，我们可以试着导出 PL/M-86 程序的一般结构。PL/M-86 程序从一个名字开始，且以这个名字结束。程序本身包含在 DO 与 END 之间(尽管在程序里面可能会出现多对 DO-END)。程序由一系列语句组成，其中有些语句是说明语句(如 DECLARE SUM BYTE)，另一些为可执行语句(如 SUM=SUM+INPUT(3)/2)。程序中使用了大量的分号，它们指出一个语句的结束。PL/M-86 程序的结构如下所示：

```
名字：
  DO;
        语句;
        ⋮                    说明语句
        语句;
        语句;
        ⋮                    可执行语句
        语句;
  END  名字;
```

这里给出的程序都采用缩进格式，这种缩进格式本身，并不是程序结构的组成部分，它对 PL/M-86 编译程序来讲是无关紧要的，但它可以帮助我们能够比较容易地阅读和理解程序，所以，我们提倡采用这种程序格式。作为说明，观察下面这段非缩进的程序文本，尽管它不会给 PL/M-86 编译程序增加麻烦(实际上它将编译得更快些)，但它却给我们的理解增加了很多困难。

```
PROG: DO; /* add inputs until totol exceeds 100 */
DECLARE SUM BYTE; SUM=0; DO WHILE SUM<=100;
SUM=SUM+INPUT(3)/2; END; OUTPUT (3)=SUM; END PROG;
```

§7.3 标　记

在讨论组成 PL/M-86 程序的各种语句之前，我们先得熟悉一下语句的一些组成块。语句由标识符、保留字、界符、常量及注释组成，有时我们也把这样一些组成块称作标记。

一、标识符

标识符是由程序员自由选择的名字。在刚讨论的程序中，就有一个标识符 SUM。标识符由字母数字组成，但它的第一个字符必须是某个字母，一个标识符所允许的最大长度为 31 个字符，从实用角度可把其长度看作是无限制的，为了提高其易读性，我们可以在标识符当中任意地插入符号 $。例如，能够把标识符 NEWSTEAM 写成 NEW$STEAM 或者写成 NEWS$TEAM，究竟写成哪种形式，就要根据所想要表达的意思来确定了。下面是几个标识符的例子：

```
X
GAMMA
JACK 5
THIS$NODE
```

二、保留字

保留字看上去很象标识符，但它们在语言中具有特定的意义，因此，不可以把它们用作一般的标识符名字。在前面的样本程序中，我们已经看到了这样一些保留字：DO、END、DECLARE、BYTE 和 WHILE。因此，我们完全可以设置象 ENDING 这样的一个名字，即是说下面这个语句是有效的：

```
DECLARE ENDING BYTE;
```

但是下面这个语句却是不适当的：

```
DECLARE END BYTE;
```

表 7.1 中列出了 PL/M-86 的全部保留字。

表 7.1 PL/M-86 的保留字

ADDRESS	CASE	END	INITIAL	NOT	REENTRANT
AND	DATA	EOF	INTEGER	OR	RETURN
AT	DECLARE	EXTERNAL	INTERRUPT	PLUS	STRUCTURE
BASED	DISABLE	GO	LABLE	POINTER	THEN
BY	DO	GOTO	LITERALLY	PROCEDURE	TO
BYTE	ELSE	HLAT	MINUS	PUBLIC	WHILE
CALL	ENABLE	IF	MOD	REAL	XOR

三、分隔符(界符)

界符是出现在 PL/M-86 程序中的一些非字母数字字符，在我们的样本程序中，出现过象<=和;这样一些界符。每种界符在语言当中都具有某种特定的意义，本章要接触到界符中的大多数，表 7.2 列出了 PL/M-86 中的所有界符。

表 7.2 PL/M-86 中的界符

$	.	+	<	◇	/*
=	/	−	>	:	*/
≔	(	'	<=	;	
@	)	*	=>		

四、常量

常量是在 PL/M-86 程序中出现的一些定值。在样本程序中我们已经碰到了 0、3 和 100 这样的一些常量。常量分整型常量、浮点数常量和串常量。

整数常量可以是 0 到 65535 ($2^{16}-1$) 之间的任何整数，通常都采用 10 进制来表示整数常量，当然也可以把它们写作 2 进制(以 B 结尾)、8 进制(以 Q 结尾)或 16 进制(以 H 结尾)。对 16 进制常量而言，为了避免与标识符相混淆，规定其开头必须是数字，在常量的头上加上一个 0，就足以解决这个问题了。例如，15、1010B、27Q 和 0BFA3H 都是整数常量。

浮点数常量是一个带有 10 进制小数点的非负数值，它还可以用 E 后面再加一个数字结尾，以此指出 10 的幂次。下面是几个浮点数常量的例子：15.6、138.和 7.0E3 及 1.32E-7。这里的 E 决不会被误认为是一个标识符，因为它的前面至少有一个数字。

所谓串常量就是包在撇号内部的一个或两个字符所组成的序列(在某些特殊情况下，也允许多于两个字符的串出现，对此我们不准备进行讨论)。撇号本身也可以包含在某个串常量当中，这只要连续写两个撇号即可。'A'、'AB'、和''''都是串常量，最后一个串常量由撇号组成。串常量的值就是串中字符的 ASCII 码。例如：'A' 的值与 41H 相同 (两者取值都为 65)，'AB' 的值与 4142H 相同。由此可见，可以把串常量与数字常量交换使用。

注意常量决不能为负，有关这一点后面将做更多的讨论。

五、注释

注释是包含在界符 /* 和 */ 里面的字符序列。它们对于编译程序没有影响，但在程序中使用这样一些注释，能使我们清楚地了解程序正在做的是些什么样的工作。尽管象

```
I=0;      /* I equals zero */
```

这样的注释近乎可笑，然而不得不承认象下面这种注释

```
I=0;      /* initialize array index prior to first iteration */
```

就使程序更容易读了。

§7.4 表 达 式

在介绍语句之前，我们还要引进一个语句组成块——表达式。表达式是由前面介绍过的一些标记组成的。

我们也可以不很严格地给表达式下个定义：它是一个由操作数和运算符组成的序列，把这些操作数和运算符组合起来可以产生一个“值”。下面先来介绍一下操作数和运算符，同时还得指出它们的组合方式以及计算表达式值的手段。

如果你阅读并理解了第五章及第六章当中有关汇编语言表达式的内容的话，你也许会觉得下面与之类似的介绍甚为有趣。在汇编语言的程序设计当中，程序运行时的指令助记符(而不是表达式)就对应于待执行的指令；而高级语言中没有指令助记符，但其表达式就表示程序运行时待执行的指令序列。它们之间的一个最为主要的区别就在于：汇编语言中的表达式，是在程序的汇编过程中赋值的；而高级语言中的表达式，则是在程序的运行过程中赋值的。

一、操作数

操作数就是具有“值”的某个东西。常量可看作是最简单的操作数，因此 15、2.7E5 及 'UG' 等都是操作数。另一种操作数就是代表单个数值的变量，一般来讲，变量就是一个标识符，例如样本程序中的 SUM。与常量不同，由变量所代表的值，直到执行程序时才能够知道，并且在程序的运行过程中，变量的值也常常发生变化。还有一种操作数就是括号内的表达式。通常把这种操作数用在某些较大的表达式当中，例如 3* (SUM+2)。

需要注意的是：操作数可以取负值，而常量却决不可以取负值，当然，我们可以在常量的前面加上一个减号，但即便如此，也不可以把它(减号)看作是常量的一个部分，它仅仅是一个数值运算符而已。

二、运算符

一个运算符可以对一个或多个操作数实施运算得到一个新的“值”。PL/M-86 中的运算符，可以分成三种类型——数值运算符、逻辑运算符和关系运算符。

数值运算符包括我们非常熟悉的加(+)、减(-)、乘(*)、除(/)，还包括一个 MOD 运算符，MOD (取模运算)计算的是除法的余数，即 19/7 等于 2，而 19 MOD 7 等于 5。

当然对数值运算符的使用也还受到一定的限制，即 PL/M-86 只允许某一些操作数组合

而不允许另外的一些组合。15+2 以及 15.3 E 7+2.1 E 3 之类的操作数——运算符组合是合法的，但是 PL/M-86 禁止 15+2.1 E 3 这类组合的出现。为了便于理解哪些组合是合法的，需要我们把操作数分为多种类型。我们把变量分成字节 (BYTE)、字 (WORD)、整型 (INTEGER) 和实型 (REAL) 这样 4 种类型，变量的类型是在说明该变量时规定的。类型为 BYTE的变量能够从 0 到 255 之间取任何整数值，WORD类型的变量可以从 0 到 65535 之间取任何整数值，类型为 INTEGER 的变量的取值范围是 −32768 到 +32767，换句话说，一个 BYTE 是一个 8 位的无符号 2 进制数；一个 WORD (字) 就相当于一个 16 位的无符号 2 进制数；而一个整型变量 (INTEGER) 是一个 16 位的带符号的 2 进制数。实型 (REAL) 变量能够从给定范围内取任何实数值(小数的或非小数的)。

我们能够把类型的概念推广到常量以及变量。由前面的介绍可以知道，常量可以是由一个或两个字符所组成的字符串，也可以是浮点数或整数。一个字符的串常量的类型为 BYTE，两个字符的串常量的类型为 WORD，浮点数常量的类型为 REAL，整数常量的类型可以是 BYTE、WORD 或 INTEGER，这要根据常量的取值来确定。常量的类型归纳在表 7.3 当中，注意表中指出的整数取值范围是从 0 到 32767 之间，这是整数常量的取值范围，它与整型变量的取值范围 (−32768 到 32767) 是不相同的。

表 7.3 常量的类型

常量	类型
0<=整数值<=255	BYTE OR INTEGER (字节或整型)
255<整数值<=32767	WORD OR INTEGER (字或整型)
32767< 数值 <=65535	WORD (字)
单字符串常量	BYTE (字节)
两个字符组成的串	WORD (字)
浮点数常量	REAL (实型数)

到此我们已经把操作数(常量和变量)分成了不同的类型，这样也就可以给出数字运算符的有效操作数的组合规则了，其实规则很简单，那就是两个操作数的类型必须相同，在大多数情况下，其结果也是这种类型。例如，我们不可以把一个整型操作数(常量或变量)加到某个实型操作数上去，这就跟不可以把苹果与桔子相加一样。这里有一个例外情况，那就是一个操作数的类型为 BYTE，另一个操作数的类型为 WORD，运算之后所得到的结果的类型也将为 WORD，这也就好象我们完全可以把镍币加到银币里去一样。

由于在设计程序时，必须记住这些规则，所以初看起来这些限制给我们学习语言增加了一些困难，但是，实际情况恰恰与此相反，它们使语言变得更容易了。因为我们必须记住的仅仅只有一个通用规则——"不能混用类型"，而不必记住象"如果把这种类型与那种类型混合，那就会得到某个别的类型的结果"这样一些乏味的规则，除此之外，汇编程序还能够帮助检查某些类型的错误。不过假如你仍然坚持或确实需要混用类型的话，本语言也为你提供了一些子程序，它们使你可以改变类型，这里不准备对此多费笔墨了。

关系运算符包括等于(=)、不等于(<>)、小于(<)、大于(>)、小于等于(<=)及大于等于(>=)这样几种。用于关系运算符的有效操作数组合与数值运算符中的规定相同，也就是说，我们可以对两个字节、两个字、两个整数进行比较，还可以把一个字节与一个字相比较，但不管是哪种操作数组合，其比较的结果总是一个字节。若比较的结果为真，则结果字节的值为 0FFH；结果为假，则取 00H 作为结果值。例如 6>5 产生结果 0FFH，1.5=2.1 的结果为 00H，但是 7>2.3 是一个无效的比较操作。

关系运算的结果(真或假)对后面所要进行的测试是有用的，例如在下面的 IF 语句当中，

就要用到比较的结果：

IF X<10 THEN X=X+1;

X<10 的结果可能是 0 FF (为真)，也可能是 00 H (若为假)，正是这个结果决定了是否要执行语句 X=X+1。

逻辑运算符是一些有用的按位“与” (AND)、“或”(OR)、异或(XOR)、非 (NOT)。逻辑操作数的类型可以是字节(结果的类型也将为字节)、字(结果的类型也将为字)，还可以这样，其中一个操作数的类型为字节，而另一个为字，这样产生的结果的类型为字。例如：

10101010 B AND 11001100 B结果为 10001000 B

1100110011001100 B OR 11110000111100 B的结果为 1111110011111100 B

但 11110000 B XOR 1.7 是无效的。

有趣的是，逻辑操作数可能是关系运算的结果，请看下面这些例子：

(1<2) AND (4>3) 产生 0 FFH AND 0 FFH，故结果为 0 FFH(真)。

(6=5) OR (1>0) 等价于 00 H OR 0 FFH，故得到结果 0 FF (真)。

NOT (1=1) 等价于 NOT 0 FFH，故得到结果 00 H (假)。

这就使我们能够构造一些有用的关系组合，例如下面这个例子当中就使用了这种组合技巧：

DO WHILE (A>3) AND (A<10);

§7.5 语 句

PL/M-86 中有两种类型的语句——说明语句和可执行语句。说明语句与数据相关联，而可执行语句却与代码相关联。

说明语句引进一个目标，并用一个名字来代表该目标，若需要还可为它分配存贮器。例如：

DECLARE COST BYTE;

这个语句引进了一个变量，变量名为 COST，同时还为它分配一个存贮器字节。

说明语句不产生任何代码，但它们可以通知编译程序应为后继的可执行语句产生哪种代码。

可执行语句描述待产生的代码，例如：

PRICE=COST+3;

对于这样一个语句，编译程序将要产生的码字，也许包含下面这条指令：把 COST 里的内容送到某个寄存器中。这时就显示出说明语句的作用来了，前面那条说明语句告诉编译程序这样的一条传送指令应为字节传送，而不是字传送指令。

§7.6 可执行语句

PL/M-86 中有这样几种可执行语句，它们是赋值语句、选择语句、重复语句和一些其他语句。本节将对这些语句进行讨论。

一、赋值语句

赋值语句是最简单的一种可执行语句。它的作用就是将表达式的值，赋给某个变量，其格

式如下：

变量名＝表达式

下面是几个赋值语句的例子：

```
LENGTH=5;
WIDTH= 2*LENGTH;
```

正象 PL/M-86 不允许我们把苹果与桔子相加一样，它也不允许我们把苹果赋给桔子，也就是说，表达式与变量的类型必须相同。因此下面两个语句是合法的：

```
DECLARE COUNT BYTE;
COUNT=117;
```

但是我们不能写

```
DECLARE COUNT BYTE;
COUNT=6.5;
```

唯一的例外情况就是：能够把字节表达式赋给字变量。所以能够写语句：

```
DECLARE COUNT WORD;
COUNT=117;
```

为了便于简单地将同一个值赋给几个变量，我们可以把赋值语句写成：

变量 1，变量 2，…，变量 n＝表达式；

例如：

```
LEFT, RIGHT=INIT-1;
```

赋值语句还可以嵌套，即能够把赋值嵌套在某个赋值语句的里面（使用一个特殊的赋值号 :=），例如在下面的语句中：

```
VOLUME=HEIGHT*(AREA:=LENGTH* WIDTH);
```

下面这段程序是一个使用赋值语句的例子：

```
FACTORIAL:
  DO; compute 1!, 2!, 3!, 4! (计算 1!, 2!, 3!, 4!)
    DECLARE FACT1 BYTE;
    DECLARE FACT2 BYTE;
    DECLARE FACT3 BYTE;
    DECLARE FACT4 BYTE;
    FACT1=1;
    FACT2=2*FACT1;
    FACT3=3*FACT2;
    FACT4=4*FACT3;
  END FACTORIAL;
```

分别对变量 FACT1、FACT2、FACT3、FACT4 进行说明。其类型均为字节 (BYTE)

执行部分，运算结果为 FACT1=1!, FACT2=2!, FACT3=3!, FACT4=4!

下面这段程序的作用与上一个相同，区别仅在于它使用了嵌套的赋值，此外，它还用一个说明语句说明了所有的 4 个变量。

```
FACTORIAL;
  DO; compute 1!, 2!, 3!, 4!
    DECLARE (FACT1, FACT2, FACT3, FACT4) BYTE;
```

(它等价于上例的4个说明语句)

```
    FACT4=4*(FACT3:=3*(FACT2:=2 * (FACT1:=1)));
```

(它等价于上例的4个执行语句,运算结果也相同)

```
END FACTORIAL;
```

二、选择语句

假如说赋值语句是仅有的一种可执行语句,那末程序员很快就会感到厌烦了,为了使程序设计更有趣、更生动,引进了选择语句。在 PL/M-86 当中有两种选择语句——IF (如果)语句和 CASE (情况)语句.

IF 语句的形式是这样的:

```
  IF 表达式 THEN 语句;
```

例如:

```
    IF SPEED>55 THEN FINE=25;
```

IF 语句的意思为:如果表达式的值为真,则执行 THEN 后面的语句。人们很自然地会提出这个疑问:"如果表达式的值不是真,那又该做什么呢?"答案是它什么也不干,除非告诉它这时它应做的事情,也就是象在下面这个例子中一样,加上保留字 ELSE,这样在表达式的值为假时,就执行 ELSE 后面的语句:

```
    IF HEIGHT<6 THEN CLEARANCE=6-HEIGHT;
    ELSE CLEARANCE=0;
```

下面的程序利用 IF 语句来计算所得税:

```
TAX:
  DO;
    DECLARE (SALARY, TAX) INTEGER;
    DECLARE (AGE, EXEMPTION) BYTE;      } 说明语句
    SALARY=…;          insert salary here
    AGE=…;             insert age here
                                  (SALARY (工资)及 AGE (年龄)是两个未
                                  定值,可根据实际情况填入)
    EXEMPTION=1;
    IF AGE=65 THEN EXEMPTION=EXEMPTION+1;
                                  (若 AGE=65, 就做 THEN 后面的语句)
    SALARY=SALARY-750*EXEMPTION;
    IF SALARY<1000 THEN TAX=14*SALARY/100;
                                  (若工资不到 1000 元, 则所得税=14*工资额
                                  /100)
    ELSE TAX=140+20*(SALARY-1000)/100;
                                  (否则,即当工资达到 1000 元时,所得税=140
                                  +20*(工资额-1000)/100)
END TAX;
```

在 IF 语句中，THEN 之后只允许出现一个语句，但是用语句括号(以 DO 语句开头且以 END 语句结束)括起来的一组语句的表现形式，就好象是单个语句一样，这样的一个语句组，就叫做一个简单的 DO 块，它类似于其他语言 (如 PASCAL) 中的复合语句。DO 块的格式为：

```
DO;
  语句;
  语句;
  ;
  语句;
END;
```

相对讲，下面这个 IF 语句就比较复杂了：

```
IF MINUTES>=60 THEN
   DO;                          (计时程序，如何分数>=60 的话，就应将小时
      HOURS=HOURS+1;            数加 1，并且从分数中减去 60；否则，无需做任何
      MINUTES=MINUTES-60;       动作。)
   END;
```

由上可见，IF 语句，具有根据表达式的真或假，选择两个语句中的一个来执行的作用。而 CASE 语句是一个更为通用的选择语句，我们可以把它看作是 IF 语句的一般化。CASE 语句按照一个表达式的值，从一组语句中选出一个语句来执行，选取哪个语句依赖于写在"DO CASE" 后面的表达式(通常是算术表达式)的结果，如其结果得 0，执行 DO 块中的第一个语句；如其结果得 1，则执行第二个语句，依此类推。当然，DO 块中的语句也可以是空的(仅由一个分号组成)，表示对被选情况没有动作。CASE 语句的形式为：

```
DO CASE  表达式;
```

我们把以 CASE 语句开头且以配对的 END 结束的一组语句称为 DO-CASE 块。例如

```
DO CASE DAY S$ OF $CHRISTMAS;
  GO TO ERROR;                                                    第0天
  PATRIDGE $IN $PEAR $TREE=PATRIDGE $IN $PEAR $TREE+1;            第1天
  TURTLE $DOVES=TURTLE $DOVES+2;                                  第2天
  FRENCH $HENS=FRENCH $HENS+3;                                    第3天
  CALLING $BIRDS=CALLING $BIRDS+4;                                第4天
  GOLDEN $RINGS=GOLDEN $RINGS+5;                                  第5天
  GEESE $A $LAYING=GEESE $A$ LAYING+6;                            第6天
  SWANS $A $SWIMMING=SWANS $A $SWIMMING+7;                        第7天
  MAIDS $A$MILKING=MAIDS $A $MILKING+8;                           第8天
  DRUMMERS $DRUMMING=DRUMMERS $DRUMMING+9;                        第9天
  PIPERS $PIPING=PIPERS $PIPING+10;                               第10天
  LADIES $DANCING=LADIES $DANCING+11;                             第11天
  LORDS $A $LEAPING=LORDS $A $LEAPING+12;                         第12天
END;
```

在上面的例子中，如果 DAYSOFCHRISMAS 为7，那么该块中得到执行的语句只是：

```
SWANS$A$SWIMMING=SWANS$A$SWIMMING+7;
```

上面的这个 DO-CASE 块，等价于下面的 IF 语句的集合：

```
IF DAYS$OF$CHRISTMAS=0 THEN
      GO TO ERROR;
ELSE IF DAYS$OF$CHRISTMAS=1 THEN
      PATRIDGE$IN$PEAR$TREE=PARTRIDGE$IN$PEAR$TREE+1;
ELSE IF DAYS$OF$CHRISTMAS=2 THEN
      TURTLE$DOVES=TURTLE$DOVES+2;
ELSE IF DAYS$OF$CHRISTMAS=3 THEN
      FRENCE$HENS=FRENCE$HENS+3;
ELSE IF DAYS$OF$CHRISTMAS=4 THEN
      CALLING$BIRDS=CALLING$BIRDS+4;
ELSE IF DAY S$OF$CHRISTMAS=5 THEN
      GOLDEN$RINGS=GOLDEN$RINGS+5;
ELSE IF DAYS$OF$CHRISTMAS=6 THEN
      GEESE$A$LAYING=GEESE$A$LAYING+6;
ELSE IF DAYS$OF$CHRISTMAS=7 THEN
      SWANS$A$SWIMMING=SWANS$A$SWIMMING+7;
ELSE IF DAYS$OF$CHRISTMAS=8 THEN
      MAIDS$A$MILKING=MAIDS$A$MILKING+8;
ELSE IF DAYS$OF$CHRISTMAS=9 THEN
      DRUMMERS$DRUMMING=DRUMMERS$DRUMMING+9;
ELSE IF DAYS$OF$CHRISTMAS=10 THEN
      PIPERS$PIPING=PIPERS$PIPING+10;
ELSE IF DAYS$OF$CHRISTMAS=11 THEN
      LADIES$DANCING=LADIES$DANCING+11;
ELSE IF DAYS$OF$CHRISTMAS=12 THEN
      LORDS$A$LEAPING=LORDS$A$LEAPING+12;
```

由此可见，我们并非一定需要 CASE 语句，因为 CASE 语句的作用，总可以象上面这样用一组 IF 语句来实现，然而利用 CASE 语句，可以使程序更为简洁，CASE 语句的结构，也要比用 IF 语句的结构更为整齐。

三、重复语句

直到现在，我们所看到的就是如何编写一个顺序执行的程序，但从实用角度出发，我们还需要重复执行一个或多个语句的能力。为此 PL/M-86 提供了两种类型的重复语句，一种就是给定重复的次数(迭代 DO 语句)，另一种就是给定重复的条件 (DO-WHILE 语句)，即当条件满足时重复执行。有时，我们还可以用更为基本的 GO TO 语句，来实现这种重复功能。

GOTO 语句的形式是：

```
GO TO 标号;
```

实际上 GOTO 是一个保留字，但为了阅读方便，也可以把它分开来写，即写成 GO TO。下面这个例子说明了 GOTO 语句的用途：

```
JAIL;
      ⋮
    GO TO JAIL;
```

迭代 DO 语句的形式是这样的：

```
DO 变量=开始表达式 TO 停止表达式 BY 步长表达式;
```

以迭代 DO 语句开头，且以与之相对应的 END 结束的一段程序，叫做迭代 DO 块。当控制首次到达 DO 语句时，计算开始表达式的值且把它赋给循环控制"变量"，然后，将该变量与停止表达式比较，如果变量值超过停止表达式，控制就转向该 DO 块下面的语句，否则继续执行该迭代 DO 块，在该块的末尾，把步长表达式的值，加到变量上，并再一次与停止表达式比较，重复上述过程。例如，一个 10 元素的数组可以用下面这种方法置 0：

```
DO I=0 TO 9 BY 1;
    ARRAY(I)=0;
    END;
```

又例如：

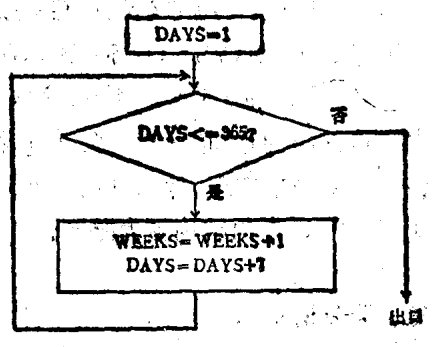
```
DO DAYS=1 TO 365 BY 7;
    WEEKS=WEEKS+1;
    END;
```

它计算的是每年有多少个星期。因为一年 365 天，每星期 7 天，故星期数 (WEEKS) 每加一次一，天数 (DAYS) 就加 7，直到天数大于 365。故最终 WEEKS 中就存放着待求的结果了。其执行过程为，把值 1，8，15，…，365 先后赋给 DAYS，在每次赋值完成之后，都要执行语句：

```
WEEKS=WEEKS+1;
```

它与下面这段程序等价：

```
    DAYS=1;
CYCLE;
    IF DAYS<=365 THEN
      DO;
        WEEKS=WEEKS+1;
        DAYS=DAYS+7;
        GOTO CYCLE;
      END;
```



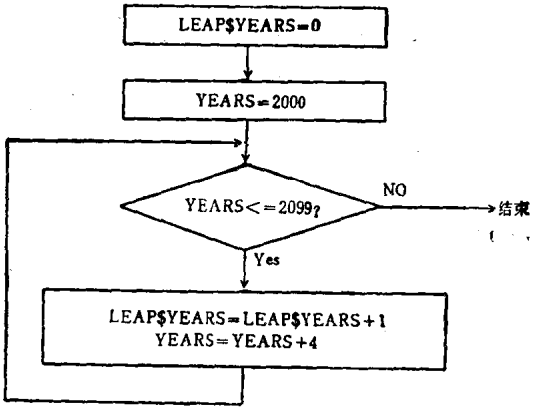
框 图

下面这个例子告诉我们，如何利用迭代 DO 语句来计算 21 世纪的闰年次数：

```
LEAPS;
  DO;
  DECLARE YEARS WORD;
```

```
    DECLARE LEAP$YEARS BYTE;
    LEAP$YEARS=0;
    DO YEARS=2000 TO 2099 BY 4;
        LEAP$YEARS=LEAP$YEARS+1;
    END;
  END LEAPS;
```

对应的流程图为：



根据闰年的确定规律，在21世纪这100年间，2000年为闰年，以后每4年闰年一次，故有上述算法。

在迭代 DO 语句中，变量(有时也称它为循环控制变量)通常都是每循环一次增1的，在这种情况下，可把“BY 1”省去。下面这段程序就说明了这一点，它计算头10个整数的和：

```
  ADD 10;
    DO;                    (计算：SUM=1+2+3+4+5+6+7+8+9+10=55)
    DECLARE I BYTE;
    DECLARE SUM BYTE;
    SUM=0;
      DO I=1 TO 10;        (循环变量 I 每次增1，即步长为1，故可省去 “BY 1”)
          SUM=SUM+I;
    END;
    END ADD 10;
```

因此，前面数组清0的例子也可以写为：

```
      DO I=0 TO 9;
        ARRAY(I)=0;
        END;
```

一般说来，当我们希望重复执行一个语句，而重复次数不依赖于循环中语句的结果，或我们事先已经能够确定重复次数时，那末利用这种迭代 DO 语句还是比较合适的，在这种语句中，赋给控制变量的值和重复执行的次数都十分明显，从而给阅读程序增加了方便，

但是，在重复执行的次数依赖于循环中语句的结果，或在我们事先不能确定重复的次数时，最好就要使用 DO-WHILE 语句，DO-WHILE 语句具有下面的形式：

```
    DO WHILE 表达式;
```

从 DO-WHILE 语句开始，直到与之配对的 END 中间的这个程序段，叫做 DO-WHILE 块。举个例子：

```
DO WHILE DEMAND>SUPPLY;
    PRICE=PRICE+1;
END;
```

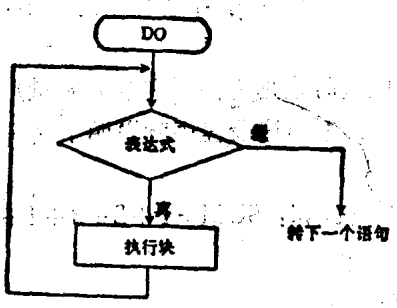
该程序完成的任务是：当"求"大于"供"时，就提高商品的价格。也就是说，在这段程序中，只要 DEMAND 大于 SUPPLY，就重复执行语句：

```
PRICE=PRICE+1;
```

它等价于下面这段程序：

```
CYCLE;
  IF DEMAND>SUPPLY THEN
     DO;
        PRICE=PRICE+1;
        GO TO CYCLE;
     END;
```

可见，我们可以把 DO-WHILE 块理解为：



即只要 WHILE 后面的表达式的值为"真"，就重复执行这些语句。DO-WHILE 常用于迭代 DO 语句不能做或不太容易做的地方，此如，在必须用非整数值控制重复的地方。需要注意的是，在 DO-WHILE 块中的语句，最终要改变条件的值，这一点很重要，因为否则的话它将无限制地重复执行下去。DO-WHILE 语句的重复次数为 0 到无穷多次。例：

```
/*DO WHILE*/
MORE=0; SPACE_OK=1;
DO WHILE (MORE AND SPACE_OK);          重复次数为 0
   ITEMS=ITEMS+1;        /*SKIPPED*/
   N_TRACKS=N_TRACKS+10;                跳过这些语句
                         /*SKIPPED*/
   IF N_TRACKS>=999      /*SKIPPED*/
      THEN SPACE_OK=0;
   END;
/*DO WHILE*/
CODE='A';
```

```
DO WHILE (CODE='A');
   TEMP=TEMP* STEP;                          (重复执行，直到
                                              TEMP>98.6)
   IF TEMP>98.6
      THEN CODE='B';
      N_STEPS=N_STEPS+1;
   END;
```

四、其他可执行语句

此外，还有几个可执行语句是：

```
HLAT;        (停机)
ENABLE;      (允许中断)
DISABLE;     (禁止中断)
```

它们对应的 8086 指令，分别为：(1)停止处理机；(2)允许中断；(3)禁止(屏蔽)中断。

§7.7 说 明 语 句

一、纯量

纯量类型的说明很简单，纯量说明是最简单的一种说明语句，它为某个变量定义一个由其所代表的数字值．例如：

```
DECLARE LITTLE$THINGS BYTE;          (8位无符号数)
DECLARE BIG$THINGS    WORD;          (16位无符号数)
DECLARE SIGNED$THINGS INTEGER;       (有符号数)
DECLARE FRACTIONAL$THINGS REAL;      (实数)
```

这样说明了之后，可给 LITTLE$THINGS 一个 0 到 255 之间的整数；可把 0 到 65535 之间的任何整数赋给 BIG$THINGS；SIGNED$THINGS 则可以从 −32768 到 +32767 之间取任一整数作值；FRACTIONAL$THINGS 则可以代表某个范围内的任何浮点数．这里给出了几个利用刚刚说明过的变量的例子：

```
LITTLE$THINGS=57;
BIG$THINGS=43195;
SIGNED$THINGS=-14216;
FRACTIONAL$THINGS=27.148;
```

你也许已经意识到必须对你所要用的变量进行说明，但是怎样才能知道应给它们什么类型呢？它们到底该是字节 (BYTE)、字 (WORD) 呢，还是该为整型 (INTEGER) 或实型 (REAL)？为了回答这个问题，就必须考虑在程序的运行过程中，各个变量的取值范围，根据它们的取值范围来确定它们的类型．例如，若某个变量既不可能为负，也不会取小数值，又不会超过 255，那么我们就应把它说明成一字节，象 NUMBEROFWIVES (实际上在一夫一妻制的今天，该变量的取值只有两种可能——"有"，"无") 和 BALL$SCORE (球赛得分) 这样一些变量，都可以取字节作为自身的类型．当这种变量有可能大于 255，但不会大于

65535时，例如书的页数 (PAGES $ IN $ BOOK)、中型公司的职员数 (NUMBER $ OF $ EMPLOYEES)，就把它说明成一个字。如果它有可能取负值，那就把它说明成整型。实型数的表示范围一般较大，可以用它们来表示现实世界中发生的某些事件。倘若一时确定不了某个变量的取值范围，那就可以考虑最坏的情况，把它当作实型数。假如某个变量仅是偶尔地超过255，哪怕只超过一点点，那也必须说明它是字。采取这样一些措施之后，程序将正确运行，尽管这样有可能使码子的数量比必要的多些，程序也可能运行得慢些，但是如果不采取这样一些措施，其程序就有可能完全死掉，更糟的是它有可能给出不正确的结果。

对于纯量说明，还可以给出这样几个例子：

```
DECLARE SWITCH BYTE;
DECLARE COUNT  WORD;
DECLARE INDEX  INTEGER;
DECLARE (NET, GROSS, TOTAL) REAL;     (3个变量)
```

二、关联项 (Related Items)

迄今我们遇到的变量说明，仅仅是纯量说明，其变量的类型都是简单类型，由纯量说明引进的变量，每次只能取一个值，在程序当中这样一些变量的值之间，也看不出有什么关系。但是这样一些值之间通常都是有联系的，而把这些有关的值集合起来，往往能够简化程序。例如，在下面这段程序中，程序读进10个人的年龄，并统计出其中超过40岁的人数：

```
OVER$40$THE$HARD$WAY;
   DO;
   DECLARE  AGE$0    BYTE;
   DECLARE  AGE$1    BYTE;
   DECLARE  AGE$2    BYTE;
   DECLARE  AGE$3    BYTE;
   DECLARE  AGE$4    BYTE;
   DECLARE  AGE$5    BYTE;
   DECLARE  AGE$6    BYTE;
   DECLARE  AGE$7    BYTE;
   DECLARE  AGE$8    BYTE;
   DECLARE  AGE$9    BYTE;
   DECLARE  OVER$40  BYTE;
      ⋮              /*read in the ages*/  (读进10个人的年龄)
   OVER$40=0;        /*initialize the count*/(开始置超过40岁的人数等于0)
   IF AGE$0>40  THEN OVER$40=OVER$40+1;
   IF AGE$1>40  THEN OVER$40=OVER$40+1;
   IF AGE$2>40  THEN OVER$40=OVER$40+1;
   IF AGE$3>40  THEN OVER$40=OVER$40+1;
   IF AGE$4>40  THEN OVER$40=OVER$40+1;
   IF AGE$5>40  THEN OVER$40=OVER$40+1;
```

```
IF AGE$6>40  THEN OVER$40=OVER$40+1;
IF AGE$7>40  THEN OVER$40=OVER$40+1;
IF AGE$8>40  THEN OVER$40=OVER$40+1;
IF AGE$9>40  THEN OVER$40=OVER$40+1;
   :                 /* do something with result*/
                     (对结果进行处理,如输出、打印等)
END OVER$40$THE$HARD$WAY;
```

很显然,变量 AGE$0、AGE$1、AGE$2、…、AGE$9 都表示年龄,从这个意义上讲,它们是彼此相关的。PL/M-86 允许将这些相关变量集合起来,用一个有 10 个字节值的变量来表示,例如,假设用 AGE 来代表该相关变量组,可把 AGE 说明成:

```
DECLARE  AGE (10)  BYTE;
```

这种具有多个值的变量称为数组。因此,我们可以把数组看作是变量的一个有序集合,其中所有的变量都具有同样的类型。数组中的元素,可以借助于下标来存取,在 PL/M-86 中,数组中的第一个元素的下标为 0,下标不一定要求是常数,也可以把变量和表达式用作下标。故数组 AGE 的各个分量(叫做元素),分别为 AGE (0)、AGE (1)、…、AGE (9)。利用数组的概念,就可以把上面那段程序改写成如下的形式:

```
OVER$40$THE$EASY$WAY;
   DO;
      DECLARE AGE(10) BYTE;
      DECLARE OVER$40 BYTE;
      :                 /* read in the ages*/   (读进 10 个人的年龄)
      OVER$40=0;        /* initialize the count*/ (初始化计数值为 0)
      DO I=0 TO 9;
      IF AGE(I)>40      THEN  OVER$40=OVER$40+1;
      END;
      :                 /* do something with result*/
      END OVER$40$THE$EASY$WAY;
```

除了类型 BYTE 之外,数组还可以具有其他类型。例如:

```
                                                    维数   类型
DECLARE LOTS$OF$LITTLE$THINGS       (100) BYTE;     100    字节
DECLARE LOTS$OF$BIG$THINGS           (25) WORD;      25    字
DECLARE LOTS$OF$SIGNED$THINGS        (50) INTEGER;   50    整型
DECLARE LOTS$OF$FRACTIONAL$THINGS    (10) REAL;      10    实型
```

除了数组之外,还有一种把相关变量集合在一起的方法,它就是所谓的结构 (structure),结构类似于 Pascal 中的记录类型。下面这个例子就是说明一个结构:

```
DECLARE RELATED$THINGS STRUCTURE
   (LITTLE$THING BYTE, BIG$THING WORD);
```

这个结构中的各个分量 (称为成员),可以用 RELATED$THINGS.LITTLE$THING 和 RELATED$THINGS.BIG$THING 来访问。结构与数组之间,存在着下面几个不同点:

1．数组的分量被称为元素，而结构的分量叫做成员。

2．数组的所有元素，都具有相同的类型，而结构的各个成员的类型可以不同。

3．数组的元素是通过其在数组中的位置（它可以是某个变量的值，我们也常把此变量叫做下标变量）来访问的，即通过下标访问数组的分量而结构中的成员却是通过名字（在程序当中，这些名字是固定的）访问的。

结构的分量可以是纯量，也可以是数组。例如：

```
DECLARE PERSON STRUCTURE (NAME (15) BYTE,
    AGE BYTE, HEIGHT REAL, WEIGHT REAL);
```

可以这样来访问该结构中的各个成员：PERSON.NAME(0)、PERSON.NAME(1)、…、PERSON.NAME(14)、PERSON.AGE、PERSON.HEIGHT 和 PERSON.WEIGHT。这种以数组作为结构的分量的例子很多，下面这两个例子都说明了这种用法：

```
DECLARE PAYCHECK   STRUCTURE (NAME (15) BYTE, SALARY WORD);
DECLARE AUTOMOBILE STRUCTURE (CHASSIS$NUMBER WORD,
    CYLINDERS BYTE, TIRE$PRESSURE (4) REAL);
```

在上面这些例子中，数组处于结构的里面，即数组是结构的分量。与此相反，结构也可以位于数组的内部，即结构可以作为数组的元素。例如：

```
DECLARE PLAYING$CARD (52) STRUCTURE (SUIT BYTE, VALUE BYTE);
```

这个结构数组的分量可以表示为 PLAYING$CARD(7).SUIT、PLAYING$CARD(25).VALUE 这种形式。下面这个例子还要复杂：

```
DECLARE PAYCHECK(100)STRUCTURE (NAME (15) BYTE, SALARY WORD);
```

这里的一些分量可表示为 PAYCHECK(38).NAME (7)、PAYCHECK (70).SALARY 等等。我们对此要感到奇怪，这样下去还有个完吗？答案是肯定的，所以不必为此烦恼。这是因为 PL/M-86 不允许结构内部包含结构，因此，我们所能够说明的最复杂的东西，就是一个包含数组的结构的数组了，它具有与上例一样的形式。

现在是看一个包含结构的例子的时候了。假设有这样一家公司，它以计算机文件的形式保存工资名单。在发薪的那一天，公司都要运行其工资管理程序，这个程序的主要作用，就是读进工资文件，并打印出工资单。但现在要加工资，假设公司决定给每个雇员增加 50 元工资，这样就要执行下面这段程序：

```
RAISES:
    DO;
        DECLARE PAYCHECK (100) STRUCTURE
          (NAME (15) BYTE, SALARY WORD);
        DECLARE I BYTE;
          ⋮                    /* read in the payroll file */  (读进工资文件)
          ⋮
        DO I=1 to 99;      /* increase everyone's salary*/ (为每人增加工资 50 元)
          PAYCHECK (I).SALARY=PAYCHECK (I).SALARY+50;
        END;
          ⋮                /* write out the updated file */ (打印出修改过的工资文件)
```

```
END RAISES;
```

三、存贮器单元

当我们在程序中写这样一个语句时，

```
DECLARE MY$SPECIAL$BYTE BYTE;
```

实际上就是告诉编译程序从存贮器中取出某个空闲的字节，把这个字节分配给MY$SPECIAL$BYTE，在一般情况下，我们不关心这样一个字节在存贮器中的位置，但是我们每个人往往都想成为机器的主人，最起码要使我们能够有这样的感受，为迎合人们的这种需要，PL/M-86允许我们规定单元的位置，例如：

```
DECLARE MY$SPECIAL$BYTE BYTE AT (3000H);
```

这样一种控制手段有时也是很有用的，因为有些单元往往具有非常特定的意义。例如，我们有可能想规定3000H单元不是代表一个存贮单元，而是代表一个输入转接口。在第四章我们已经知道这种应用被称为存贮器映象I/O。在这种情况下，保证变量 MY$SPECIAL$BYTE 指向单元 3000 H 而不指向任何别的地方，就显得非常重要了。

有时候，即使没有告诉编译程序在哪里分配一个变量，我们也有可能希望知道编译程序取的是哪个单元，这样的一个位置是个常量，甚至在程序的运行过程中，它也不发生改变，在程序当中也许要使用这个常量。PL/M-86 在不告诉我们该常量是什么的情况下，允许我们表达此常量，即当我们要访问单元 MY$SPECIAL$BYTE 的时候，就可以在程序中写出 @MY$SPECIAL$BYTE，我们把这样一些常量称为访问位置常量 (reference-location constants)。

利用访问位置常量，我们可能想用某个别的变量的位置来确定一个变量的位置，例如：

```
DECLARE FLOOR (20) WORD;
DECLARE LOBBY AT (@ FLOOR(0));
```

这样我们就既可以用 FLOOR (0)，也可以用 LOBBY 来访问 FLOOR (0) 了。

利用访问位置常量可做的第二件事情，是把该常量赋给别的某个变量。例如：

```
MY$SPECIAL$LOCATION = @MY$SPECIAL$BYTE;
```

但是在写这个语句之前，我们应该确保这里所做的不是把苹果赋给桔子。为了满足这一需要，PL/M-86 提供了一种特殊类型——指针 (POINTER) 类型，利用这种类型来访问存贮单元。访问位置常量的类型就是指针。请看下面的例子：

```
DECLARE OZ REAL;
DECLARE YELLOW$BRICK$ROAD POINTER;
YELLOW$BRICK$ROAD = @OZ;
DECLARE DATA$PNTR POINTER;
DATA$PNTR = 3A07H;
```

由上面的第二个例子可见，整数常量也可以是指针类型。

指针的使用是受到严格限制的，它们不可以与数值运算符或逻辑运算符一起使用，例如语句

```
DATA$PNTR = DATA$PNTR + 1;
```

是无效的。可用于指针的运算符只有关系运算符一种，因此下面这个语句是有效的：

```
IF DATA$PNTR = YELLOW$BRICK$ROAD THEN…;
```

如果说借助于指针，我们所能做的就是对它们进行比较或赋值的话，那么它们真的有存在的必要吗？它们真的有用吗？对这样一个问题的回答，基于下述事实：我们确实不想对指针再作更多的操作，但希望对指针指向的那些东西进行更多的操作，我们希望能够象使用变量一样使用指针指向的东西。在 PL/M-86 当中，可以用一个名字来代表指针所指的东西，例如：

```
DECLARE ITEM$PNTR POINTER;
DECLARE ITEM BASED ITEM$PNTR BYTE;
```

在这个例子当中，ITEM 就是 ITEM$PNTR 所指向的那个字节的名字，ITEM 的位置是不固定的，它随着 ITEM$PNTR 的改变而改变。ITEM 被称为有基变量，它是以 ITEM $ PNTR 为基的。

为了便于理解，我们准备举一个使用了有基变量的例子，它所做的工作是把数组中的最大值清 0。首先，不用有基变量来编制该程序，然后再用有基变量来解决此问题。由此，可对两者进行比较。

```
WITHOUT $BASED$ VARIABLES;                (不用有基变量)
  DO;
    DECLARE ITEM(50) WORD;                /* the array of words */
                                          (说明一个 50 维的字数组)
    DECLARE BIG $ITEM $INDEX BYTE;        /* index into array for biggest value */
                                          (指向数组中最大值的指针)
    DECLARE I BYTE;                       /* a running index into array */
                                          (下标变量)
        .                                 /* read in the 50 values */
        .
        .                                 (读进数组的 50 个值)
    BIG $ITEM $INDEX=0;                   /* initialize the index */
                                          (索引的初始化)
    DO I=1 TO 49;                         /* find the biggest item */
                                          (求出数组中的最大值，它由BIG $ITEM $
    IF ITEM(I)>ITEM(BIG $ITEM $INDEX)       INDEX 指向)
      THEN BIG $ITEM $INDEX=I;
    END;
    ITEM (BIG $ITEM $INDEX) =0;           /* zero out the biggest item */
    :                                     (将原最大者置零)
    END WITHOUT$BASED$VARIABLES;
WITH $BASED $ VARIABLES;                  (利用有基变量)
    DO;
    DECLARE ITEM (50) WORD;               /* the array of words */
    DECLARE BIG $ITEM $PNTR POINTER;      /* pointer to biggest value */
    DECLARE BIG$ITEM BASED BIG $ITEM $PNTR WORD;
                                          /* this is the biggest item */
```

```
DECLARE I BYTE;                          /* a running index into array */
  ⋮                                      /* read in 50 values */
BIG$ITEM$PNTR=@ITEM(0);                  /* initialize the pointer */
DO I=1 TO 49;
  IF ITEM(I)>BIG$ITEM                    /* find the biggest item */
                                         (求出数组中的最大项)
      THEN BIG$ITEM$PNTR=@ITEM(I);
END;
BIG$ITEM=0;                              /* zero out the biggest item */
                                         (最大项置0)
  ⋮                                      /* write out the 50 values */
END WITH$BASED$VARIABLES;
```

四、文字说明 (Literal Declarations)

为了方便起见，PL/M-86 允许我们用一个名字，来代替一个字符序列。例如：

```
DECLARE PI LITERALLY'3.14159';
```

有了这样一个语句之后，我们就能够在程序中，把 PI 用作为 3.14159 的代名词，即在原来要求写上 3.14159 的地方，只要写上 PI 即可。这种说明语句的另外一个用途就是说明常量，其中的这个常量，也许我们在今后某个时候(下星期、下个月)要对之进行修改，而在做修改时就无需修改多处。也就是说我们并不是在整个程序中，都用那个常量，而是给常量一个名字，如

```
DECLARE BUFFER$SIZE LITERALLY '32';
```

这样就在程序中出现常量 32 的地方，用 BUFFER$SIZE 来代替，显然若要改变这个常量，只要对相应的说明语句进行一次修改就可以了，从而大大方便了程序员，也提高了程序的正确性。

"聪明的"程序员大概已经发现：使用 LITERALLY 能够产生 PL/M-86 中的保留字的同义词，利用这种技巧，可以很容易地编写一些极难读的程序。象下面这个例子：

```
DECLARE LTL LITERALLY 'LITERALLY';
DECLARE  DCL LTL 'DECLARE';
DCL WRD LTL 'WORD';
DCL MQP WRD;                          (看不懂了吧)
```

假如你愿意的话，当然可以随意利用这种技巧。但值得注意的是，倘若你写出这样的语句：

```
bt dnt cm z me whn u gt n trbl
```

又有谁能够理解呢？我想即便是你本人，理解起来恐怕也不会很便当的。

§7.8 过　程

程序设计中一个非常重要的概念就是子程序或曰过程。利用过程，我们就能够在程序的不同地方多次执行某个代码段，而不需在所有这些地方重复写出它们的代码。下面以一个美

元兑换的例子作为说明：

```
MAKING $ CHANGE;
  DO;
    DECLARE COINS(8) BYTE;          /* this is the result */
                                    (保存结果的数组)
    DECLARE CHANGE BYTE;            /* number to be converted */
                                    (该找回的钱数)
    DECLARE I BYTE;                 /* index into COINS array */
                                    (数组 COINS 的下标)
    CHANGE=100-…;                   /* write the cost here */
                                    (在此填上开销，假设你给店员一美元，而
                                    买的东西不值这么多)
    I=0;                            /* initialize the index */
    DO WHILE CHANGE>=50;            /* half dollars */      (半美元)
      COINS(I)=50;
      I=I+1;
      CHANGE=CHANGE-50;
    END;
    DO WHILE CHANGE>=25;            /* quarters */  (2角5分)
      COINS(I)=25;
      I=I+1;
      CHANGE=CHANGE-25;
    END;
    DO WHILE CHANGE>=10;            /* dimes */  (1角银币)
      COINS(I)=10;
      I=I+1;
      CHANGE=CHANGE-10;
    END;
    DO WHILE CHANGE>=5;             /* nickle */  (5分镍币)
      COINS(I)=5;
      I=I+1;
      CHANGE=CHANGE-5;
    END;
    DO WHILE CHANGE>=1;             /* pennies */  (便士)
      COINS(I)=1;
      I=I+1;
      CHANGE=CHANGE-1;
    END;
    DO WHILE I<8;                   /* zero out rest of coins */
```

```
    COINS(I)=0;
    I=I+1;
  END;
END MAKING$CHANGE;
```

对于 X 的不同值，下面这个代码序列在程序中的几个地方出现过：

```
    COIN(I)=X;
    I=I+1;
    CHANGE=CHANGE-X;
```

倘若我们能够仅在一个地方给出这段代码，然后根据需要，在程序的其他地方调用它，无疑这将简化上面的程序。在 PL/M-80 中，只要把这段代码当作一个过程，即可达到这个目的。

利用过程之后，上面的程序就成为：

```
MAKING$CHANGE$WITH$PROCEDURES;
  DO;
    DECLARE COINS(8) BYTE;          /* this is the result */
    DECLARE CHANGE BYTE;            /* number to be converted */
    DECLARE I BYTE;                 /* index into COINS array */
    NEXT$COIN;                      /* this is a procedure declaration */
                                    (过程说明)
      PROCEDURE (X);                /* X is specified when procedure is called */
                                    (X 是在调用过程时定值的)
        DECLARE X BYTE;
        COINS(I)=X;
        I=I+1;
        CHANGE=CHANGE-X;
      END NEXT$COIN;
    CHANGE=100-…;                   /* write the cost here */
    I=0;                            /* initialize the index */
    DO WHILE CHANGE>=50;            /* half dollars */
      CALL NEXT$COIN(50);
    END;
    DO WHILE CHANGE>=25;            /* quarters */
      CALL NEXT$COIN(25);
    END;
    DO WHILE CHANGE>=10;            /* dimes */
      CALL NEXT$COIN(10);
    END;
    DO WHILE CHANGE>=5;             /* nickles */
      CALL NEXT$COIN(5);
    END;
```

```
    DO WHILE CHANGE>=1;          /* pennies */
      CALL NEXT$COIN(1);
    END;
    DO WHILE I<8;                /* zero out rest of coins */
      CALL NEXT$COIN(0);
    END;
  END MAKING$CHANGE$WITH$PROCEDURES;
```

上面的例子说明了这样一个事实：过程是一个代码段，它仅被说明但不执行，它与程序中的其它说明语句一起出现，这一次序是不可避免的，因为我们一般必须在使用一个实体之前先定义它。过程被说明了之后，就可以在程序的其他地方利用 CALL 语句来调用它，也只有在用 CALL 语句调用它的时候，该过程才真正投入运行。

一、信息传递 (Passing Information)

在许多实际应用当中，我们都要发给过程一些输入信息，并要从过程接收一些输出信息(结果)。给过程发送信息的最简便的方法，就是将要传递的信息在调用过程之前，送到某个(或某些)特殊变量当中，在每次调用该过程时，都使用相同的变量，过程也能以同样的方式返回结果。在下面这个例子中，信息传递就是通过特殊变元实现的。

```
        说明                              调用
UP$COUNT;                          CALL UP$COUNT
  PROCEDURE;
    COUNT=COUNT+1;
  END UP$COUNT;
```

在这个例子当中，COUNT 就是一个特殊变元，利用它给过程发送信息，也利用它从过程接收信息。

给过程发送信息的另一种方法，就是在每次调用过程时，都要给出这些信息，通常也把以这种方式确定的信息叫做参数。在下例中，50 就是一个参数：

```
CALL NEXT$COIN(50);
```

在过程当中，对应于每个参数都存在着一个变量，每次调用过程时，都把参数值放到对应的变量当中去。使用参数的一个例子如下：

```
        说明                              调用
CHECK$SIZE;                        DECLARE MAX BYTE;
  PROCEDURE(I,J);                  DECLARE MIN BYTE;
    DECLARE I BYTE;                      ⋮
    DECLARE J BYTE;                CALL CHECK$SIZE (MAX, MIN);
    IF I<J THEN COUNT=COUNT+1;
  END CHECK$SIZE;
```

在这个例子当中，MAX 和 MIN 的值(不是地址)被送到了 CHECK$SIZE，并分别成为其参数 I 和 J 的值，这里，CHECK$SIZE 根本不知道 MAX 和 MIN 的位置，因而它也就无法改变 MAX 和 MIN 的值了。

但是，如果参数中所包含的是某个值的地址，而不再是值本身，那末过程就既能够从该单元中取得一个值，也能够将某个结果放到该单元当中去。这一点将在下面的例子中得到说明。

说明

```
SWITCH:
  PROCEDURE(I,J);
    DECLARE I POINTER;
    DECLARE J POINTER;
    DECLARE VAL$I BASED I BYTE;
    DECLARE VAL$J BASED J BYTE;
    DECLARE TEMP BYTE;
    TEMP=VAL$I;
    VAL$I=VAL$J;
    VAL$J=TEMP;
  END SWITCH;
```

调用

```
DECLARE FIRST BYTE;
DECLARE LAST BYTE;
        :
CALL SWITCH(@FIRST, @LAST)
```

例中送给过程的已不再是 FIRST 和 LAST 的值，而是它们的地址，即 @FIRST 和 @LAST，因此 I 和 J 的值也分别成为 FIRST 和 LAST 的地址，这样，在过程里面，就需要有对应于 FIRST 和 LAST 的变量，实际上 VAL$I 与 VAL$J 就是起这种作用的两个变元，也就是说，在过程内部，我们把 VAL$I 和 VAL$J 当作 FIRST 和 LAST。

从过程返回结果的方法也不止这些，但在介绍最后一种结果返回方法之前，我们必须先引进 RETURN (返回) 语句。迄今为止，我们总是假定只有在碰到过程结束语句 (END) 时，过程才返回，事实上我们可以在此 END 语句的紧前面加上一个 RETURN 语句，从而使这种返回方式表现得更为明显。举个例子：

```
UPCOUNT:
  PROCEDURE;
  COUNT=COUNT+1;
  RETURN;                    /* this statement is optional */ (该语句也可不加)
END UPCOUNT;
```

这样一个 RETURN 语句，并不是必需的，因为即使没有它，编译程序在到达过程的终点时，也知道必须返回，然而有些过程有可能在到达过程的终点之前就要返回，这时就需要 RETURN 语句了，例如下面这个过程：

```
UPCOUNT:
  PROCEDURE;
  IF COUNT=10 THEN RETURN;
  COUNT=COUNT+1;
  RETURN;                    /* this statement is optional */
END UPCOUNT;
```

现在我们可以介绍从过程获得结果的最后一种方法了，即以过程的名字得到结果。这时，我们不再是用 CALL语句来调用程序，而是把过程名当作表达式的一个操作数来实现对过程的调用。请看下例：

说明：

```
PHONE$BILL:
  PROCEDURE (NUMBER$OF$CALLS) WORD;
    DECLARE NUMBER$OF$CALLS BYTE;
    RETURN 500+5*NUMBER$OF$CALLS;
  END PHONE$BILL;
```

调用：

```
EXPENSES=PHONE$BILL(78)+ELECTRIC$BILL(113)+…;
```

这种以过程名本身，作为返回值的过程与其他过程的不同点，反映在下面两个方面，第一点 RETURN 语句确定了返回值，所以这时 END 语句前面的 RETURN 语句不再是可有可无了，它是必须出现的；第二，过程本身决定了待返回的结果的类型，例如上例中结果的类型为字 (WORD)。因此也称这些过程为有类型过程。

综上所述，我们已经讨论了给过程发送信息的三种方法，也讨论了从过程接收信息的三种方法。概括如下：

发送信息到过程	从过程接收信息
特定变量	特定变量
值参数	地址参数
地址参数	有类型过程

二、中断过程

第三章已经比较详细地讨论过 8086 的中断机构，概括起来讲，中断机构大致是这样的：外部设备给处理器发送一个中断信号，以及一个 0 到 255 之间的数(中断序号)，然后中断处理器，即处理器响应中断，转去执行相应的中断处理程序。在 PL/M-86 中，可以说明(定义)包含中断号的过程，也就是说，我们可以规定中断子程序。与习惯上的过程不同，中断过程不是用 CALL 语句调用的，而是在处理器响应某个中断时自动地调用、运行适当的中断过程。例如下面这个过程，它只有当 75 号中断发生时才被调用：

```
KEY$PRESS:
  PROCEDURE INTERRUPT 75;
    CHARACTER=INPUT(1);
  END KEY$PRESS;
```

三、重入过程 (Reentrant Procedures)

重入过程，就是自己可以调用自己的那些过程。尽管这种情况不会经常出现，但有时它确实是很有用的。作为例子，假设我们要写一段计算阶乘的程序，比如说要计算 7!，计算 7! 的一种方法为计算 6! 与 7 的乘积。也就是说，对于一个计算 X 的阶乘的过程来说，它要先调用计算 (X-1) 的阶乘的过程，然后再将 (X-1) 的阶乘与 X 相乘获得结果。但是，如果我们不是十分小心的话，这一过程也许永远不会结束，因此为了保证过程调用序列能够终止，规定在求 1 的阶乘时，该阶乘过程就返回结果 1，而不再调用任何其他过程。PL/M-86 的求阶乘的过程是这样的：

```
FACTORIAL:
  PROCEDURE (X) WORD REENTRANT;
    DECLARE X BYTE;
    IF X=1 THEN RETURN 1;
    RETURN X*FACTORIAL (X-1);
  END FACTORIAL;
```

在 FACTORIAL 过程的说明中，包含了一个我们从未见过的关键字 REENTRANT，它告诉编译程序这个过程是可重入的，即在得到所要求的结果之前，可能要多次进入该过程。编译程序也必须知道这一点，只有这样它才能保存与初始输入有关的信息(如例中的 X)。倘若不是这样，又会出现什么情况呢？当用语句

```
ANSWER=1+FACTORIAL(7);
```

调用过程FACTORIAL时，变量X的值为7，接着过程FACTORIAL就将调用FACTORIAL(7-1)，这时将重新进入 FACTORIAL 过程，且X的值变为6，而X的初始值7就被丢失了。由于X值的丢失，FACTORIAL(7) 也就不可能返回一个正确的结果值 X*FACTORIAL(6)。解决这个问题的方法就是加上 REENTRANT，有了它，编译程序就在每次进入该过程时，为X分配一个不同的存贮单元，保存好X的原始值。

过程调用自身，仅是过程重入的一种方法，重入过程的另一种方法就是：某个过程调用第二个过程，而第二个过程反过来又调用第一个(原始)过程，这种形式的重入，也被称做递归。如果在执行某个过程时，中断发生了，这样就必须转去进行中断处理，完成后再调用原来正运行的过程，可见这种过程也能重入。总之，任何在返回之前，有可能进入多次的过程都应做上标记 REENTRANT，以使它们能够正确运行。倘若你不清楚它们是否可重入，那你总可以把它们全都标记成 REENTRANT，这样得出的结果将是正确的。

四、间接过程调用

假设我们想要调用一个过程，但又不知道究竟要调用哪一个，这个过程要到执行程序时才能确定。例如，我们想把50转换成一个数字序列，但转换成何种类型，则依赖于它们出现的先后次序：

```
DECLARE A$CONVERSION$ROUTING POINTER;
          ⋮
A$CONVERSION$ROUTING=@CONVERT$TO$BINARY;
GO TO COMMON$PLACE;
          ⋮
A$CONVERSION$ROUTING=@CONVERT$TO$OCTAL;
GO TO COMMON$PLACE;
          ⋮
A$CONVERSION$ROUTING=@CONVERT$TO$HEXADECIMAL;
GO TO COMMON$PLACE;
          ⋮
A$CONVERSION$ROUTING=@CONVERT$TO$NUMERALS;
```

```
    GO TO COMMON$PLACE;
                ⋮
COMMON$PLACE:
    CALL A$CONVERSION$ROUTING (50);
```

在程序的不同地方，我们把转换子程序的地址赋给 A$CONVERSION$ROUTING，然后，当到达 COMMON$PLACE 时，就能够间接调用某个转换子程序，甚至还可以传递一个参数。

§7.9 块结构和作用域

我们已经知道，如何在程序的一个部分中说明目标(变量、过程等等)，但是我们还没讲在这些目标被说明之后，究竟在程序的哪些地方可以使用这些目标。对于每个目标来讲，它都有一个作用域，所谓作用域，就是指程序中能够识别该目标名的那些部分。

在讨论作用域之前，必须先引进块的概念。块就是以 DO 或 PROCEDURE 开头，且以相应的 END 结束的一个语句序列。一个完整的 PL/M-86 程序，也是一个块，此外，PL/M-86 中还有这样一些块：

1. 过程说明；
2. 简单 DO 块；
3. DO-WHILE 块；
4. 迭代 DO 块；
5. DO-CASE 块。

我们已经看到，在过程说明的开头能够说明目标，目标还可以在任何简单 DO 块的开头被说明。

现在能够定义目标的作用域了，作用域是由下面的等式定义的：

作用域＝说明目标的那个块＋所有嵌套块－重新说明过同一个标识符的那些块。

一般来讲，我们应遵守这样一个限制，即在使用目标之前，必须先说明目标。但是由于标号也是一种目标，因此，必须弄清标号说明的意义。标号被看成是在包含它的最小块的头上说明的。下面我们准备用例子来说明这些作用域规则。

例 1. 作用域包括说明目标的那个块。

```
    DO;
      DECLARE X BYTE;
      X=X+1;                    /* of course this is within the scope of X */
                                (该X在其作用域内)
    END;
```

例 2：作用域也包括嵌套块

```
DO;
  DECLARE X BYTE;
  DO;
    DECLARE Y BYTE;
```

```
    X=Y+1;                          /* this is also within the scope of X*/
                                    (这也属于X的作用域)
  END;
END;
```

例3：作用域不包括重新说明了相同标识符的那些嵌套块.

```
DO;
  DECLARE X BYTE;
  DO;
    DECLARE X(5) BYTE;
    X=X+1;                          (出错. 因为这里已重新把X说明成数组，不再
                                    是在标量X的作用域之内了)
    X(3)=X(2)+1;                    (该语句有效，因为它是在数组X的作用域之
                                    内)
    END;
  X=X+1;                            (在纯量X的作用域之内)
END;
```

例4：作用域不包括外层块

```
DO;
  DO;
    DECLARE X BYTE;
  END;
  X=X+1;                            (出错.因为它不属于X的作用域)
END;
```

例5：目标必须在使用之前说明

```
DO;
  A:
    PROCEDURE;
    X=X+1;                          (出错.这是因为X还没被说明)
  END A;
  DECLARE X BYTE;
END;
```

例6：标号能被向前引用

```
DO;
    ⋮                               (把标号L看作是在此说明的)
    GOTO L;                         (根据约定,L已在块头说明,故语句正确)
    L:                              (这并不是L的说明)
END;
```

例7：标号的作用域还包括内层块

```
DO;
```

```
    ⋮                                    (把L看作是在此说明的)
  DO;
    GO TO L;
  END;
  L:
END;
```

例 8：重入过程能够在说明之前调用

```
DO;
  A:
    PROCEDURE REENTRANT;
      CALL B;
    END A;
  B:
    PROCEDURE REENTRANT;
      CALL A;
    END B;
END;
```

在这种情况下，因为每个过程都要调用另一个，因而不可能都满足先定义、后调用的要求，也就是说 REENTRANT 过程是“说明——使用”规则的一个例外。

§7.10 输 入/输 出

众所周知，对一个完整的程序设计语言来讲，它应该包括输入(数据送到程序)输出(送出答案)手段。在前面的例子当中，曾经使用过语句

```
SUM=SUM+INPUT(3);
```

来输入数据，并使用下面这个语句输出结果

```
OUTPUT(3)=SUM;
```

一般说来，把 INPUT(i) 用作表达式中的一个操作数，就能够从输入转接口 i 读进一个数据字节；而通过在赋值语句的左边使用 OUTPUT(j)，就能够把一个数据字节写到任意输出转接口 j。进一步讲，利用 INWORD(i) 或 OUTWORD(j) 就能从转接口读进一个数据字，或把一个数据字写到输出转接口。在下面这个例子当中，程序从头 100 个输入转接口读进 16 位数据，并且把它们写回到各自的输出转接口。

```
IN $ ONE $ PORT $ AND $ OUT $ THE $ OTHER;
  DO;
    DECLARE I BYTE;
    DO I=0 TO 99;
      OUTWORD(I)=INWORD(I);
    END;
  END IN $ ONE $ PORT $ AND $ OUT $ THE $ OTHER;
```

§7.11 模块程序设计

人们在不断为现代计算机系统的巨大功能及惊人速度而赞叹不已的同时，也对计算机软件系统本身存在的严重问题表示深深的关注，“软件危机”至今仍然存在。所谓软件危机它包含两个方面的内容：一方面，投入开发软件的资源日益增加，软件成本越来越高，甚至占整个系统价格的一半以上；另一方面，软件系统的可靠性得不到保证，几乎没有不存在错误的软件系统。世界上最精心设计，并花费了巨额投资的美国阿波罗登月飞行系统的软件程序也没有避免错误，阿波罗 14 号在 10 天的飞行中，出现了 18 个软件错误。

我们用巨额投资得到的却是不可靠的软件，这样一个严酷的事实妨碍着计算机的进一步发展。而由于大型程序的发展，程序不可靠性问题变得日益严重，Dijkstra 认为这是因为在程序中缺乏良好的结构，从而使程序的编写、调试、维护都变得十分困难，为了解决这些问题，他首先提出了“结构程序设计”的概念，即自上而下的逐步精细化设计程序，使程序结构化、模块化。实践证明，采用结构程序设计的思想所编制的程序，无论在编写速度还是在软件质量上，都比非结构程序设计好得多。下面我们就来看看 PL/M-86 是如何支持程序的结构化设计的。

迄今，我们一直把以 “NAME:DO;” 开头，而以 “END NAME;” 结尾的那个代码块称为一个程序。严格讲，它仅仅是一个模块，而一个程序是由一个或多个模块组成的。每个模块的编译都独立于别的模块。这样一来，我们就能够把一个程序分给几个程序员去编制，同时每个程序员又都可以把它的程序分成较小的、易于理解的一些片段。

现在来温习一下模块的结构，它有这样的形式：

```
名字：
  DO;
    语句；
    语句；
    ⋮
    语句；
  END 名字；
```

其中的语句既可以是说明语句，也可以是可执行语句，但说明语句总是在先。任何语句本身都可以是一个块（过程块、DO-WHILE 块等等），块中包含了其他一些语句。PL/M-86 也是分层结构的，而这种层次结构，则是通过使用块得到的。这样一来，我们就有必要把上面显式列出的语句，与包含在这些语句内部的那些语句区别开来，我们将使用术语最外层的语句，来代表上面显式列出的语句。

组成程序的那个模块叫做主程序，实际上叫主模块可能更好些，可惜的是人们已经习惯用主程序这个术语了。主程序由最外层的可执行语句和说明语句（可能有）组成，事实上，主程序本身就可能是个完整的程序。

现在就可以使用模块设计技术了。我们可以把一个程序分成几个任务，比如说一个任务负责从某个复杂数据结构中（例如飞机订票系统）读写数据，另一个任务负责对这些读写数据进行处理。因此，读写数据的一些过程说明与数据本身的说明一起可以组成一个模块，而把对

数据的实际操作放到主程序当中去，这样一来，就可以由不同的程序员去分别编写各程序块了。

前面我们已经提到过：说明语句为编译程序提供了一些信息，从而使编译程序知道，应为可执行语句产生哪种代码。例如：

```
DECLARE THIS$HERE$THING WORD;
```

就使编译程序知道在它碰到

```
THIS$HERE$THING=0;
```

时，必须产生一个把0送到两个相邻的存贮器字节中去，而不是送到一个字节当中，此外，DECLARE 语句还使编译程序为 THIS$HERE$THING 分配两个连续的存贮器字节。因此在编译程序碰到该赋值语句时，它也就知道了该将哪两个字节清0。

但是，如果 THIS$HERE$THING 的说明在某个别的模块当中，当编译程序看到一个包括 THIS$HERE$THING 的赋值语句时，编译程序又该怎么办呢？它可能在不知道 THIS$HERE$THING 的地址的情况下，产生一个置0的代码，并做上标记，以表示要求某人在后面填上正确的单元地址，然而，除非编译程序知道 THIS$HERE$THING 的类型(BYTE 或 WORD)，否则它就不能产生任何代码。

为了帮助编译程序正确地实现程序的编译，任何使用到 THIS$HERE$THING 而没有说明它的模块，都至少必须告诉编译程序该单元的类型。在 PL/M-86 当中，这被写成：

```
DECLARE THIS$HERE$THING WORD EXTERNAL;
```

其意思为：THIS$HERE$THING 是一个字，其说明在本模块的外面。尽管这个语句很象一个说明，然而事实上它不是，它仅仅规定了 THIS$HERE$THING 的类型，但并未为它保留任何存贮器空间。这样，编译程序就能够产生类型正确的赋值语句代码了，只是它还不知道在码子当中，应放上什么存贮器地址。

在真正说明 THIS$HERE$THING 的模块当中，应该告诉编译程序其他还有一些模块要使用该变元 THIS$HERE$THING，然后，编译程序就记住 THIS$HERE$THING 的地址，以便将其填入其他模块的某些码子中去。因此，我们把该说明写成：

```
DECLARE THIS$HERE$THING WORD PUBLIC;
```

由此编译程序就意识到这个说明是公共的，其他模块也可以使用该信息。

在编译完了所有的模块之后，必须检查一遍由编译程序留下的所有标记，这些标记都附在各模块的码字中，它们确定了"必须把 THIS$HERE$THING 的地址写到码字的哪些地方，或(2) THIS$HERE$THING 的地址是什么。然后就能够把 THIS$HERE$THING 的地址写到码字的适当位置去了。这样一个过程，可看作是将各种模块链到一起的过程，进行这种链接工作的人或程序，就被称为链接者或链接程序。当然，你也不必为此惊慌，我们不一定要你也非成为链接者不可，因为当你接到 PL/M-86 编译程序时，你也就同时收到了一个链接程序。

下面的例子，说明了模块的用法，其中第一个模块为主程序，它使用了 SUCCESSOR 和 COUNT。

```
M1:
  DO;                                   /* first module */   (第一个模块)
    DECLARE COUNT BYTE PUBLIC;                               (说明部分)
```

```
    SUCCESSOR:
    PROCEDURE (X) BYTE PUBLIC;
      DECLARE X BYTE;
      RETURN X+1;
    END SUCCESSOR;
  END M1;
M2:
  DO;                                  /* second module */(第二个模块)
    DECLARE ARG BYTE;                  /* this is a declaration */  (说明)
    DECLARE COUNT BYTE EXTERNAL;                                    (不是说明)
    SUCCESSOR:
    PROCEDURE (X) BYTE EXTERNAL;                                    (这也不是说明)
      DECLARE X BYTE;
    END SUCCESSOR;
    ARG=3;
    COUNT=SUCCESSOR (ARG);
  END M2;
```

§7.12 交通灯管理程序

现在我们来结束对交通灯问题的讨论，在第四章，我们已经为它设计了一个控制交通灯的系统，现在的任务就是为这一系统配上软件，即用 PL/M-86 来编制一个交通灯的管理程序。

假设交通灯位于一条主要的交通干线(主线)，与一条小的交叉街道的交叉口。在一般情况下，控制主线的交通灯为绿，而控制交叉街道的信号灯为红。只有在交叉街道上有多于5辆汽车等待过马路时，主线上的灯才变为红，交叉街道上的灯闪烁黄色，直到交叉街道上不剩汽车时，才改变信号灯的这种状态。

在第四章给出的系统中，交通灯是作为存贮器映象输出口使用的，其存贮器地址为1000H，我们假定交通灯中的各个灯泡与 1000H 单元各位的对应关系如下：

(最左位) 7 主线上的红灯
6 主线上的黄灯
5 主线上的绿灯
4 主线上的左转灯
3 交叉街道上的红灯
2 交叉街道上的黄灯
1 交叉街道上的绿灯
(最右位) 0 交叉街道上的左转灯

在交通灯控制系统中，还有一个输入转接口，它告诉处理机有多少汽车在交叉街道上等着，这个转接口是从埋在地下的感应器与一些计数电路中获得信息的，我们假定它是50号输入

转接口。

根据上面这些要求、条件，我们用 PL/M-86 编制了下面这段程序：

```
TRAFFIC$LIGHT:
  DO;
    DECLARE LIGHTS BYTE AT(1000H);          /* memory-mapped output */
                                            (存贮器映象输出转接口)
    DECLARE CAR$COUNT LITERALLY 'INPUT(50)'; /* input */
    DECLARE MAIN$RED LITERALLY '80H';  /* names for individual bulbs */
                                            (各灯泡的名字)
    DECLARE MAIN$YELLOW LITERALLY '40'H;
    DECLARE MAIN$GREEN LITERALLY '20H';
    DECLARE MAIN$LEFT$TURN LITERALLY '10H';
    DECLARE CROSS$RED LITERALLY '08H';
    DECLARE CROSS$YELLOW LITERALLY '04H';
    DECLARE CROSS$GREEN LITERALLY '02H';
    DECLARE CROSS$LEFT$TURN LITERALLY '01H';
  DELAY:
    PROCEDURE (X);                          /* causes an X second delay*/
                                            (延迟X秒)
      DECLARE X BYTE;
        :
    END DELAY;
  START:
    LIGHT=MAIN$GREEN+CROSS$RED;             /* normal setting */
                                            (在一般情况下交通灯的状态)
    IF CAR$COUNT>5 THEN                     /* too many cars waiting */
                                            (若交叉街道上等待的车子太多，就
                                            让它们通过)
      DO;                                   /* let them go through */
        LIGHTS=MAIN$YELLOW+CROSS$RED; /* stop highway */
                                            (禁止主线上的车子通行)
      CALL DELAY(3);
        DO WHILE CAR$COUNT>0;               /* start cross street */
                                            (开始通过交叉街道)
        LIGHTS=MAIN$RED;
        CALL DELAY(1);
        LIGHTS=MAIN$RED+CROSS$YELLOW;
        CALL DELAY(1);
      END;
```

```
    END;
    GO TO START;                          /* and repeat the cycle */
  END;                                    /* of program */
```

现在我们让系统更复杂一些，要求当发出火警时，将城里的所有交通灯都置为红灯。假设从失火点发出两种信号给每个信号灯，一个信号表示发出警报，另一个信号表示没事了。

我们把警报信号接到处理机的中断 (INTR) 引脚，同时还加上一些电路，以便在适当的时候传送对应的中断类型(例如它是 10) 给处理机，所有的出事/安全信号线都连到 51 号输入转接口，以使在事件过去(进入安全状态)之后，从该输入转接口读到的是全 1 (真)；否则，即还存在危险时，从 51 号输入转接口读到的为全 0 (假)。为了适合这种需求，在程序中我们又加进了这样一个过程说明：

```
FIRE$ALARM:
  PROCEDURE INTERRUPT 10;
    DECLARE SAVED$LIGHTS BYTE;
    DECLARE ALL$CLEAR LITERALLY 'INPUT(51)';
    SAVED$LIGHTS=LIGHTS;                  /* we need to restore these later */
                                          (以便火警解除后，恢复交通灯的状
                                          态)
    DO WHILE NOT ALL$CLEAR;               /* blinking red */
                                          (若出现火警，红灯闪烁)
      LIGHTS=0;
      CALL DELAY (1);
      LIGHTS=MAIN$RED+CROSS$RED;
      CALL DELAY(1);
    END;
    LIGHTS=SAVED$LIGHTS;                  /* restore old settings */
                                          (恢复原交通灯的状态)
  END FIRE$ALARM;
```

在此，我们不想再继续设计下去了，当然你也许有兴趣进一步修改、扩充上述的程序，以使其更实用，处理更为复杂的问题。这无疑是个很好的尝试。

本章并非是 PL/M-86 的所有特色及规则的汇编，而是以易于消化的形式介绍了语言的大部分特色，通过本章的学习，使你能够编些有意义的程序。尽管有好多细节问题相当重要，然而由于篇幅的限制，我们也没在这里进行讨论。如果你感兴趣的话，可以从《英特尔 PL/M-86 程序设计手册》(Intel PL/M-86 Programming Mannual) 中了解到 PL/M-86 的所有细节。

习　　题

用 PL/M-86 写出下列程序

7.1 求下列级数之和

$1+x+x^2/2!+x^3/3!+\cdots+x^n/n!$

7.2 编写程序，计算 $T=1^3+2^3+3^3+\cdots+N^3$，直到T的值等于或大于 10^4 为止。

7.3 写一个过程，用以求数组元素的平方值，并把这些值放在一个一维数组中。例如，设有 1 * 4 的 A 数组与 B 数组，求：

$$D(I)=A(I)^2-B(I)^2$$
$$x=A(1)^2+A(2)^2+A(3)^2+A(4)^2$$
$$y=B(1)^2+B(2)^2+B(3)^2+B(4)^2$$

提示：主程序模块可两次调用该过程，并把 A 数组元素的平方值放在 AA(4) 的数组中，把 B 数组元素的平方值放在 BB(4) 的数组中，然后分别求出 D 数组的各元素以及 x、y 的值。

7.4 如果某小组 5 个人的工资分别为 T_1、T_2、T_3、T_4、T_5。为便于发放，问需要领回拾元、伍元、贰元、一元、一角一张的人民币各多少张，以及一分钱的硬币多少枚？

7.5 设有 10 个学生的成绩分别为：76, 69, 84, 90, 73, 88, 99, 63, 100, 80。分别写出解决以下三个问题的 PL/M-86 程序。

① 输出最优秀的成绩；

② 把学生成绩按优劣排序(分数高的放在前面；)

③ 用下标变量 S(6) 统计 60～69 分之间的人数，S(7) 统计 70～79 分之间的人数，S(8) 统计 80～89 分之间的人数，S(9) 统计 90～99 分之间的人数，S(10) 统计 100 分的人数。

7.6 设计一个模拟家庭关系的数据结构。每一个人用一个记录表示，该记录包含他或她的名字，和指向双亲、配偶和子女的指针。写一个过程在该结构中插入一个新的人，并写几个过程建立新的人和现有家庭成员的关系，例如：

offspring (parent, child)
marry (wife, husband)

附 录 A

8086 指令系统摘要

一、数据传送

1. MOV：传送

	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
寄存器/存贮器与寄存器之间交换数据	1 0 0 0 1 0 d w	mod reg r/m		
立即数传送给寄存器/存贮器	1 1 0 0 0 1 1 w	mod 0 0 0 r/m	数 据	若 w=1为数据
立即数传送给寄存器	1 0 1 1 w reg	数 据	若 w=1为数据	
由存贮器传送给累加器	1 0 1 0 0 0 0 w	偏移低字节	偏移高字节	
由累加器传送给存贮器	1 0 1 0 0 0 1 w	偏移低字节	偏移高字节	
寄存器/存贮器传送给段寄存器	1 0 0 0 1 1 1 0	mod 0 reg r/m		
段寄存器传送给寄存器/存贮器	1 0 0 0 1 1 0 0	mod 0 reg r/m		

2 PUSH：压入堆栈

寄存器/存贮器	1 1 1 1 1 1 1 1	mod 1 1 0 r/m
寄存器	0 1 0 1 0 reg	
段寄存器	0 0 0 reg 1 1 0	

3. POP：弹出堆栈

寄存器/存贮器	1 0 0 0 1 1 1 1	mod 0 0 0 r/m
寄存器	0 1 0 1 1 reg	
段寄存器	0 0 0 reg 1 1 1	

4. XCHG：交换

	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
寄存器/存贮器与寄存器交换	1 0 0 0 0 1 1 w	mod reg r/m
寄存器与累加器交换	1 0 0 1 0 reg	

5. IN：输入到 AL/AX

从固定转接口输入到 AL/AX	1 1 1 0 0 1 0 w	转 接 口
从可变转接口输入到 AL/AX	1 1 1 0 1 1 0 w	

6. OUT：从 AL/AX 输出到：

从 AL/AX 输出到固定转接口	1 1 1 0 0 1 1 w	转 接 口

从 AL/AX 输出到可变转接口	1 1 1 0 1 1 1 w
7. XLAT：把翻译字节传送给 AL	1 1 0 1 0 1 1 1
8. LEA：把 EA 装入寄存器	1 0 0 0 1 1 0 1 | mod reg r/m
9. LDS：把指示器装入 DS	1 1 0 0 0 1 0 1 | mod reg r/m
10. LES：把指示器装入 ES	1 1 0 0 0 1 0 0 | mod reg r/m
11. LAHF：把标志装入 AH	1 0 0 1 1 1 1 1
12. SAHF：把 AH 存入标志	1 0 0 1 1 1 1 0
13. PUSHF：压入标志	1 0 0 1 1 1 0 0
14. POPF：弹出标志	1 0 0 1 1 1 0 1

二、算术指令

1. ADD：加法

	7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0
寄存器/存贮器与寄存器相加，结果送回其一	0 0 0 0 0 0 d w | mod reg r/m
立即数与寄存器/存贮器相加	1 0 0 0 0 0 s w | mod 0 0 0 r/m | 数 据 | 若 s,w=0,1 为数据
立即数与累加器相加	0 0 0 0 0 1 0 w | 数 据 | 若 w=1 为数据

2. ADC：带进位的加法

寄存器/存贮器与寄存器相加，结果送回其一，	0 0 0 1 0 0 d w | mod reg r/m
立即数与寄存器/存贮器相加	1 0 0 0 0 0 s w | mod 0 1 0 r/m | 数 据 | 若 s,w=0,1 为数据
立即数加到累加器	0 0 0 1 0 1 0 w | 数 据 | 若 w=1 为数据

3. INC：递加

寄存器/存贮器递加	1 1 1 1 1 1 1 w | mod 0 0 0 r/m
寄存器递加	0 1 0 0 0 reg

4. AAA：加法的 ASCII 调整 — 0 0 1 1 0 1 1 1

5. DAA：加法的 10 进制调整 — 0 0 1 0 0 1 1 1

6. SUB：减法

寄存器/存贮器和寄存器相减，结果送回两者之一	0 0 1 0 1 0 d w | mod reg r/m
由寄存器/存贮器减去立即数	1 0 0 0 0 0 s w | mod 1 0 1 r/m | 数 据 | 若 s,w=0,1 为数据
由累加器减立即数	0 0 1 0 1 1 0 w | 数 据 | 若 w=1 为数据

7. SBB：带借位的减法

寄存器/存贮器和寄存器相减，结果送回两者之一	0 0 0 1 1 0 d w | mod reg r/m

寄存器/存贮器减立即数

1 0 0 0 0 0 s w	mod 0 1 1 r/m	数　据	若s,w=0,1为数据

累加器减立即数

0 0 0 1 1 1 0 w	数　据	若 w=1 为数据

8. DEC：递减

寄存器/存贮器递减

1 1 1 1 1 1 1 w	mod 0 0 1 r/m

寄存器递减

0 1 0 0 1 reg

9. NEG：改变符号

1 1 1 1 0 1 1 w	mod 0 1 1 r/m

10. CMP：比较

寄存器/存贮器和寄存器比较：

0 0 1 1 1 0 d w	mod reg r/m

立即数与寄存器/存贮器比较：

1 0 0 0 0 0 s w	mod 1 1 1 r/m	数　据	若s,w=0,1为数据

立即数与累加器比较

0 0 1 1 1 1 0 w	数　据	若 w=1 为数据

11. AAS：减法的 ASCII 调整

0 0 1 1 1 1 1 1

12. DAS：减法的 10 进制调整

0 0 1 0 1 1 1 1

13. MUL：乘法(无符号)

1 1 1 1 0 1 1 w	mod 1 0 0 r/m

14. IMUL：整数乘法(带符号)

1 1 1 1 0 1 1 w	mod 1 0 1 r/m

15. AAM：乘法的 ASCII 调整

1 1 0 1 0 1 0 0	0 0 0 0 1 0 1 0

16. DIV：除法(无符号)

1 1 1 1 0 1 1 w	mod 1 1 0 r/m

17. IDIV：整数除法(带符号)

1 1 1 1 0 1 1 w	mod 1 1 1 r/m

18. AAD：除法的 ASCII 调整

1 1 0 1 0 1 0 1	0 0 0 0 1 0 1 0

19. CBW：将字节转换为字

1 0 0 1 1 0 0 0

20. CWD：将字转换为双字

1 0 0 1 1 0 0 1

三　逻辑运算

1. NOT："非"

1 1 1 1 0 1 1 w	mod 0 1 0 r/m

2. SHL/SAL：逻辑/算术左移

1 1 0 1 0 0 v w	mod 1 0 0 r/m

3. SHR 逻辑右移

1 1 0 1 0 0 v w	mod 1 0 1 r/m

4. SAR 算术右移

1 1 0 1 0 0 v w	mod 1 1 1 r/m

5. ROL：左环移

1 1 0 1 0 0 v w	mod 0 0 0 r/m

6. ROR：右环移

1 1 0 1 0 0 v w	mod 0 0 1 r/m

7. RCL：带进位左环移

1 1 0 1 0 0 v w	mod 0 1 0 r/m

8. RCR：带进位右环移

1 1 0 1 0 0 v w	mod 0 1 1 r/m

AND：与

9. 寄存器/存贮器和寄存器进行"与"操作，结果送回两者之一

0 0 1 0 0 0 d w	mod reg r/m

10. 立即数"与"寄存器/存贮器

1 0 0 0 0 0 0 w	mod 1 0 0 r/m	数 据	若 w=1 为数据

11. 立即数"与"累加器

0 0 1 0 0 1 0 w	数 据	若 w=1 为数据

TEST：对操作数"与"操作，不取结果

12. 寄存器/存贮器"与"寄存器

1 0 0 0 0 1 0 w	mod reg r/m

13. 立即数"与"寄存器/存贮器

1 1 1 1 0 1 1 w	mod 0 0 0 r/m	数 据	若 w=1 为数据

14. 立即数"与"累加器

1 0 1 0 1 0 0 w	数 据	若 w=1 为数据

OR：或

15. 寄存器/存贮器"或"寄存器，结果送回两者之一

0 0 0 0 1 0 d w	mod reg r/m

16. 立即数"或"寄存器/存贮器

1 0 0 0 0 0 0 w	mod 0 0 1 r/m	数 据	若 w=1 为数据

17. 立即数"或"累加器

0 0 0 0 1 1 0 w	数 据	若 w=1 为数据

XOR：异或

18. 寄存器/存贮器"异或"寄存器，结果送回到两者之一

0 0 1 1 0 0 d w	mod reg r/m

19. 立即数"异或"寄存器/存贮器

1 0 0 0 0 0 0 w	mod 1 1 0 r/m	数 据	若 w=1 为数据

20. 立即数"异或"累加器

0 0 1 1 0 1 0 w	数 据	若 w=1 为数据

四、字符串操作

1. REP：重复

1 1 1 1 0 0 1 Z

2. MOVS：传送字节/字

1 0 1 0 0 1 0 w

3. CMPS：比较字节/字

1 0 1 0 0 1 1 w

4. SCAS：扫描字节/字

1 0 1 0 1 1 1 w

5. LODS：把字节/字装入 LODS

1 0 1 0 1 1 0 w

6. STOS：把 AL/AX 的字节/字存贮起来

1 0 1 0 1 0 1 w

五、控制传移

CALL：调用

1. 段内直接调用：

1 1 1 0 1 0 0 0	位移低字节	位移高字节

2. 段内间接调用：

1 1 1 1 1 1 1 1	mod 0 1 0 r/m

3. 段间直接调用：

1 0 0 1 1 0 1 0	偏移低字节	偏移高字节
	段低字节	段高字节

4. 段间间接调用：

1 1 1 1 1 1 1 1	mod 0 1 1 r/m

六、JMP：无条件跳转

1. 段内直接跳转

11101001	位移低字节	位移高字节

2. 短段内直跳转

11101011	位　移

3. 段内间接跳转

11111111	mod 100 r/m

4. 段间直接跳转

11101010	偏移低字节	偏移高字节
	段低字节	段高字节

5. 段间间接跳转

11111111	mod 101 r/m

七、RET：由 CALL 返主

1. 段内返主

11000011

2. 段内立即数加于 SP

11000010	数据低字节	数据高字节

3. 段间返主

11001011

4. 段间立即数加于 SP

11001010	数据低字节	数据高字节

5. JE/JZ：相等/为0跳转

01110100	位　移

6. JL/JNGE：小于/不大于等于跳转

01111100	位　移

7. JLE/JNG：小于等于/不大于跳转

01111110	位　移

8. JB/JNAE：低于/不高于等于跳转

01110010	位　移

9. JBE/JNA：低于等于/不高于跳转

01110110	位　移

10. JP/JPE：奇偶性为偶/校验为偶跳转

01111010	位　移

11. JO：溢出跳转

01110000	位　移

12. JS：有符号跳转

01111000	位　移

13. JNE/JNZ：不相等/不为0跳转

01110101	位　移

14. JNL/JGE：不小于/大于等于跳转

01111101	位　移

15. JNLE/JG：不小于等于/大于跳转

01111111	位　移

16. JNB/JAE：不低于/高于等于跳转

01110011	位　移

17. JNBE/JA：不低于等于/高于跳转

01110111	位　移

18. JNP/JPO：奇偶性为奇/奇偶校验为奇跳转

01111011	位　移

19. JNO：不溢出跳转

01110001	位　移

20. JNS：无符号跳转

01111001	位　移

21. LOOP：循环 CX 次 | 11100010 | 位　移 |

22. LOOPZ/LOOPE：当为 0/相等时循环 | 11100001 | 位 转 移 |

23. LOOPNZ/LOOPNE：当不为 0/不相等时循环 | 11100000 | 位　移 |

24. JCXZ：CX 为 0 跳转 | 11100011 | 位　移 |

INT：中断

1. 指定类型 | 11001101 | 类　型 |

2 类型 3： | 11001100 |

3. INTO：溢出中断 | 11001110 |

4 IRET：中断返回 | 11001111 |

八、处理机控制

1. CLC：清除进位 | 11111000 |

2 CMC：进位求反 | 11110101 |

3. STC：设置进位 | 11111001 |

4. CLD：清除 | 11111100 |

5. STD：设置方向 | 11111101 |

6. CLI：清除中断 | 11111010 |

7. STI：设置中断 | 11111011 |

8 HLT：停机 | 11110100 |

9. WAIT：等待 | 10011011 |

10. ESC：交权 | 11011 xxx | mod xxx r/m |

11. LOCK：总线封锁前缀 | 11110000 |

脚注：

AL=8 位累加器

AX=16 位累加器

CX=计数寄存器

DS=数据段寄存器

ES=附加段寄存器

高于/低于系指无符号值

大于/小于系指带符号值

若 d=1，则为“进入”寄存器，若 d=0，则为“来自”寄存器

若 w=1，则为字指令；若 w=0，则为字节指令

若 mod=11，则 r/m 作为 REG 字段

若 mod=00，则位移=0*，没有位移低字节和位移高字节

若 mod=01，则位移=位移低字节，有符号扩展到16位；没有位移高字节

若 mod=10，则位移=位移高字节，位移低字节

若 r/m=000，则 EA=(BX)+(SI)+DISP

若 r/m=001，则 EA=(BX)+(DI)+DISP

若 r/m=010，则 EA=(BP)+(SI)+DISP

若 r/m=011，则 EA=(BP)+(DI)+DISP

若 r/m=100，则 EA=(SI)+位移

若 r/m=101，则 EA=(DI)+位移

若 r/m=110，则 EA=(BP)+位移*

若 r/m=111，则 EA=(BX)+位移

位移紧跟指令的第2字节(如果需要的话，可放在数据之前)

若 s: w=01，则形成16位立即数操作数

若 s:w=11，则一个立即数据字节有符号扩展而成16位的操作数。

若 v=0，则"计数"=1；若 v=1，则"计数"输入(CL)。

x=任意

z 用于串基本指令(Primitive)，与 ZF 标志作比较。

段超越前缀：

0 0 1	reg	1 1 0

REG 是根据下表分配的：

16位(w=1)	8位(w=0)	段
0 0 0 AX	0 0 0 AL	0 0 ES
0 0 1 CX	0 0 1 CL	0 1 CS
0 1 0 DX	0 1 0 DL	1 0 SS
0 1 1 BX	0 1 1 BL	1 1 DS
1 0 0 SP	1 0 0 AH	
1 0 1 BP	1 0 1 CH	
1 1 0 SI	1 1 0 DH	
1 1 1 DI	1 1 1 BH	

访问标志寄存器的指令作为16位目标，用符号 FLAGS 表示：

FLAGS=X:X:X:X:(OF):(DF):(IF):(TF):(SF):(ZF):X:(AF):X:(PF):X:(CF)

附 录 B

8086 机器指令译码指南

第一字节		第二字节	第3、4、5、6字节	ASM-86 指令格式	
HEX	BINARY				
00	0000 0000	MOD REG R/M	(DISP-LO), (DISP-HI)	ADD	REG8/MEM8, REG8
01	0000 0001	MOD REG R/M	(DISP-LO), (DISP-HI)	ADD	REG16/MEM16, REG16
02	0000 0010	MOD REG R/M	(DISP-LO), (DISP-HI)	ADD	REG8, REG8/MEM8
03	0000 0011	MOD REG R/M	(DISP-LO), (DISP-HI)	ADD	REG16, REG16/MEM16
04	0000 0100	DATA-8		ADD	AL, IMMED8
05	0000 0101	DATA-LO	DATA-HI	ADD	AX, IMMED16
06	0000 0110			PUSH	ES
07	0000 0111			POP	ES
08	0000 1000	MOD REG R/M	(DISP-LO), (DISP-HI)	OR	REG8/MEM8, REG8
09	0000 1001	MOD REG R/M	(DISP-LO), (DISP-HI)	OR	REG16/MEM16, REG16
0A	0000 1010	MOD REG R/M	(DISP-LO), (DISP-HI)	OR	REG8, REG8/MEM8
0B	0000 1011	MOD REG R/M	(DISP-LO), (DISP-HI)	OR	REG16, REG16/MEM16
0C	0000 1100	DATA-8		OR	AL, IMMED8
0D	0000 1101	DATA-LO	DATA-HI	OR	AX, IMMED16
0E	0000 1110			PUSH	CS
0F	0000 1111			(not used)	
10	0001 0000	MOD REG R/M	(DISP-LO), (DISP-HI)	ADC	REG8/MEM8, REG8
11	0001 0001	MOD REG R/M	(DISP-LO), (DISP-HI)	ADC	REG16/MEM16, REG16
12	0001 0010	MOD REG R/M	(DISP-LO), (DISP-HI)	ADC	REG8, REG8/MEM8
13	0001 0011	MOD REG R/M	(DISP-LO), (DISP-HI)	ADC	REG16, REG16/MEM16
14	0001 0100	DATA-8		ADC	AL, IMMED8
15	0001 0101	DATA-LO	DATA-HI	ADC	AX, IMMED16
16	0001 0110			PUSH	SS
17	0001 0111			POP	SS
18	0001 1000	MOD REG R/M	(DISP-LO), (DISP-HI)	SBB	REG8/MEM8, REG8
19	0001 1001	MOD REG R/M	(DISP-LO), (DISP-HI)	SBB	REG16/MEM16, REG16
1A	0001 1010	MOD REG R/M	(DISP-LO), (DISP-HI)	SBB	REG8, REG8/MEM8
1B	0001 1011	MOD REG R/M	(DISP-LO), (DISP-HI)	SBB	REG16, REG16/MEM16
1C	0001 1100	DATA-8		SBB	AL, IMMED8
1D	0001 1101	DATA-LO	DATA-HI	SBB	AX, IMMED16
1E	0001 1110			PUSH	DS
1F	0001 1111			POP	DS
20	0010 0000	MOD REG R/M	(DISP-LO), (DISP-HI)	AND	REG8/MEM8, REG8
21	0010 0001	MOD REG R/M	(DISP-LO), (DISP-HI)	AND	REG16/MEM16, REG16
22	0010 0010	MOD REG R/M	(DISP-LO), (DISP-HI)	AND	REG8, REG8/MEM8
23	0010 0011	MOD REG R/M	(DISP-LO), (DISP-HI)	AND	REG16, REG16/MEM16
24	0010 0100	DATA-8		AND	AL, IMMED8
25	0010 0101	DATA-LO	DATA-HI	AND	AX, IMMED16

(续)

第一字节 HEX	第一字节 BINARY	第二字节	第3、4、5、6字节	ASM-86 指令格式	
26	0010 0110			ES:	(segment override prefix)
27	0010 0111			DAA	
28	0010 1000	MOD REG R/M	(DISP-LO), (DISP-HI)	SUB	REG8/MEM8, REG8
29	0010 1001	MOD REG R/M	(DISP-LO), (DISP-HI)	SUB	REG16/MEM16, REG16
2A	0010 1010	MOD REG R/M	(DISP-LO), (DISP-HI)	SUB	REG8, REG8/MEM8
2B	0010 1011	MOD REG R/M	(DISP-LO), (DISP-HI)	SUB	REG16, REG16/MEM16
2C	0010 1100	DATA-8		SUB	AL, IMMED8
2D	0010 1101	DATA-LO	DATA-HI	SUB	AX, IMMED16
2E	0010 1110			CS:	(segment override prefix)
2F	0010 1111			DAS	
30	0011 0000	MOD REG R/M	(DISP-LO), (DISP-HI)	XOR	REG8/MEM8, REG8
31	0011 0001	MOD REG R/M	(DISP-LO), (DISP-HI)	XOR	REG16/MEM16, REG16
32	0011 0010	MOD REG R/M	(DISP-LO), (DISP-HI)	XOR	REG8, REG8/MEM8
33	0011 0011	MOD REG R/M	(DISP-LO), (DISP-HI)	XOR	REG16, REG16/MEM16
34	0011 0100	DATA-8		XOR	AL, IMMED8
35	0011 0101	DATA-LO	DATA-HI	XOR	AX, IMMED16
36	0011 0110			SS:	(segment override prefix)
37	0011 0110			AAA	
38	0011 1000	MOD REG R/M	(DISP-LO), (DISP-HI)	CMP	REG8/MEM8, REG8
39	0011 1001	MOD REG R/M	(DISP-LO), (DISP-HI)	CMP	REG16/MEM16, REG16
3A	0011 1010	MOD REG R/M	(DISP-LO), (DISP-HI)	CMP	REG8, REG8/MEM8
3B	0011 1011	MOD REG R/M	(DISP-LO), (DISP-HI)	CMP	REG16, REG16/MEM16
3C	0011 1100	DATA-8		CMP	AL, IMMED8
3D	0011 1101	DATA-LO	DATA-HI	CMP	AX, IMMED16
3E	0011 1110			DS:	(segment override prefix)
3F	0011 1111			AAS	
40	0100 0000			INC	AX
41	0100 0001			INC	CX
42	0100 0010			INC	DX
43	0100 0011			INC	BX
44	0100 0100			INC	SP
45	0100 0101			INC	BP
46	0100 0110			INC	SI
47	0100 0111			INC	DI
48	0100 1000			DEC	AX
49	0100 1001			DEC	CX
4A	0100 1010			DEC	DX
4B	0100 1011			DEC	BX
4C	0100 1100			DEC	SP
4D	0100 1101			DEC	BP
4E	0100 1110			DEC	SI
4F	0100 1111			DEC	DI
50	0101 0000			PUSH	AX
51	0101 0001			PUSH	CX
52	0101 0010			PUSH	DX
53	0101 0011			PUSH	BX

(续)

第一字节 HEX	第一字节 BINARY	第二字节	第3、4、5、6字节	ASM-86 指令格式
54	0101 0100			PUSH SP
55	0101 0101			PUSH BP
56	0101 0110			PUSH SI
57	0101 0111			PUSH DI
58	0101 1000			POP AX
59	0101 1001			POP CX
5A	0101 1010			POP DX
5B	0101 1011			POP BX
5C	0101 1100			POP SP
5D	0101 1101			POP BP
5E	0101 1110			POP SI
5F	0101 1111			POP DI
60	0110 0000			(not used)
61	0110 0001			(not used)
62	0110 0010			(not used)
63	0110 0011			(not used)
64	0110 0100			(not used)
65	0110 0101			(not used)
66	0110 0110			(not used)
67	0110 0111			(not used)
68	0110 1000			(not used)
69	0110 1001			(not used)
6A	0110 1010			(not used)
6B	0110 1011			(not used)
6C	0110 1100			(not used)
6D	0110 1101			(not used)
6E	0110 1110			(not used)
6F	0110 1111			(not used)
70	0111 0000	IP-INC8		JO SHORT-LABEL
71	0111 0001	IP-INC8		JNO SHORT-LABEL
72	0111 0010	IP-INC8		JB/JNAE/ SHORT-LABELJC
73	0111 0011	IP-INC8		JNB/JAE/ SHORT-LABELJNC
74	0111 0100	IP-INC8		JE/JZ SHORT-LABEL
75	0111 0101	IP-INC8		JNE/JNZ SHORT-LABEL
76	0111 0110	IP-INC8		JBE/JNA SHORT-LABEL
77	0111 0111	IP-INC8		JNBE/JA SHORT-LABEL
78	0111 1000	IP-INC8		JS SHORT-LABEL
79	0111 1001	IP-INC8		JNS SHORT-LABEL
7A	0111 1010	IP-INC8		JP/JPE SHORT-LABEL
7B	0111 1011	IP-INC8		JNP/JPO SHORT-LABEL
7C	0111 1100	IP-INC8		JL/JNGE SHORT-LABEL
7D	0111 1101	IP-INC8		JNL/JGE SHORT-LABEL
7E	0111 1110	IP-INC8		JLE/JNG SHORT-LABEL
7F	0111 1111	IP-INC8		JNLE/JC SHORT-LABEL
80	1000 0000	MOD 000 R/M	(DISP-LO), (DISP-HI), DATA-8	ADD REG8/MEM8, IMMED8

第一字节		第二字节	第3、4、5、6字节	ASM-68 指令格式	
HEX	BINARY				
80	1000 0000	MOD 001 R/M	(DISP-LO), (DISP-HI) DATA-8	OR	REG8/MEM8, IMMED8
80	1000 0000	MOD 010 R/M	(DISP-LO), (DISP-HI) DATA-8	ADC	REG8/MEM8, IMMED8
80	1000 0000	MOD 011 R/M	(DISP-LO), (DISP-HI) DATA-8	SBB	REG8/MEM8, IMMED8
80	1000 0000	MOD 100 R/M	(DISP-LO), (DISP-HI) DATA-8	AND	REG8/MEM8, IMMED8
80	1000 0000	MOD 101 R/M	(DISP-LO), (DISP-HI) DATA-8	SUB	REG8/MEM8, IMMED8
80	1000 0000	MOD 110 R/M	(DISP-LO), (DISP-HI) DATA-8	XOR	REG8/MEM8, IMMED8
80	1000 0000	MOD 111 R/M	(DISP-LO), (DISP-HI) DATA-8	CMP	REG8/MEM8, IMMED8
81	1000 0001	MOD 000 R/M	(DISP-LO), (DISP-HI) DATA-LO, DATA-HI	ADD	REG16/MEM16, IMMED16
81	1000 0001	MOD 001 R/M	(DISP-LO), (DISP-HI) DATA-LO, DATA-HI	OR	REG16/MEM16, IMMED16
81	1000 0001	MOD 010 R/M	(DISP-LO), (DISP-HI) DATA-LO, DATA-HI	ADC	REG16/MEM16, IMMED16
81	1000 0001	MOD 011 R/M	(DISP-LO), (DISP-HI) DATA-LO, DATA-HI	SBB	REG16/MEM16, IMMED16
81	1000 0001	MOD 100 R/M	(DISP-LO), (DISP-HI) DATA-LO, DATA-HI	AND	REG16/MEM16, IMMED16
81	1000 0001	MOD 101 R/M	(DISP-LO), (DISP-HI) DATA-LO, DATA-HI	SUB	REG16/MEM16, IMMED16
81	1000 0001	MOD 110 R/M	(DISP-LO), (DISP-HI), DATA-LO, DATA-HI	XOR	REG16/MEM16, IMMED16
81	1000 0001	MOD 111 R/M	(DISP-LO), (DISP-HI), DATA-LO, DATA-HI	CMP	REG16/MEM16, IMMED16
82	1000 0001	MOD 000 R/M	(DISP-LO), (DISP-HI), DATA-8	ADD	REG8/MEM8, IMMED8
82	1000 0010	MOD 001 R/M		(not used)	
82	1000 0010	MOD 010 R/M	(DISP-LO), (DISP-HI) DATA-8	ADC	REG8/MEM8, IMMED8
82	1000 0010	MOD 011 R/M	(DISP-LO), (DISP-HI), DATA-8	SBB	REG8/MEM8, IMMED8
82	1000 0010	MOD 100 R/M		(not used)	
82	1000 0010	MOD 101 R/M	(DISP-LO), (DISP-HI), DATA-8	SUB	REG8/MEM8, IMMED8
82	1000 0010	MOD 110 R/M		(not used)	
82	1000 0010	MOD 111 R/M	(DISP-LO), (DISP-HI) DATA-8	CMP	REG8/MEM8, IMMED8
83	1000 0011	MOD 000 R/M	(DISP-LO), (DISP-HI), DATA-SX	ADD	REG16/MEM16, IMMED8
83	1000 0011	MOD 001 R/M		(not used)	
83	1000 0011	MOD 010 R/M	(DISP-LO), (DISP-HI) DATA-SX	ADC	REG16/MEM16, IMMED8
83	1000 0011	MOD 011 R/M	(DISP-LO), (DISP-HI), DATA-SX	SBB	REG16/MEM16, IMMED8
83	1000 0011	MOD 100 R/M		(not used)	
83	1000 0011	MOD 101 R/M	(DISP-LO), (DISP-HI), DATA-SX	SUB	REG16/MEM16, IMMED8
83	1000 0011	MOD 110 R/M		(not used)	

(续)

第一字节		第二字节	第3、4、5、6字节	ASM-86 指令格式	
HEX	BINARY				
83	1000 0011	MOD 111 R/M	(DISP-LO), (DISP-HI) DATA-SX	CMP	REG16/MEM16, IMMED8
84	1000 0100	MOD REG R/M	(DISP-LO), (DISP-HI)	TEST	PEG8/MEM8, REG8
85	1000 0101	MOD REG R/M	(DISP-LO), (DISP-HI)	TEST	REG16/MEM16, REG16
86	1000 0110	MOD REG R/M	(DISP-LO), (DISP-HI)	XCHG	REG8, REG8/MEM8
87	1000 0111	MOD REG R/M	(DISP-LO), (DISP-HI)	XCHG	REG16, REG16/MEM16
88	1000 1000	MOD REG R/M	(DISP-LO), (DISP-HI)	MOV	REG8/MEM8, REG8
89	1000 1001	MOD REG R/M	(DISP-LO), (DISP-HI)	MOV	REG16/MEM16/REG16
8A	1000 1010	MOD REG R/M	(DISP-LO), (DISP-HI)	MOV	REG8, REG8/MEM8
8B	1000 1011	MOD REG R/M	(DISP-LO), (DISP-HI)	MOV	REG16, REG16/MEM16
8C	1000 1100	MOD 0SR R/M	(DISP-LO), (DISP-HI)	MOV	REG16/MEM16, SEGREG
8C	1000 1100	MOD 1-R/M		(not used)	
8D	1000 1101	MOD REG R/M	(DISP-LO), (DISP-HI)	LEA	REG16, MEM16
8E	1000 1110	MOD 0SR R/M	(DISP-LO), (DISP-HI)	MOV	SEGREG, REG16/MEM16
8E	1000 1110	MOD 1-R/M		(not used)	
8F	1000 1111	MOD 000 R/M	(DISP-LO), (DISP-HI)	POP	REG16/MEM16
8F	1000 1111	MOD 001 R/M		(not used)	
8F	1000 1111	MOD 010 R/M		(not used)	
8F	1000 1111	MOD 011 R/M		(not used)	
8F	1000 1111	MOD 100 R/M		(not used)	
8F	1000 1111	MOD 101 R/M		(not used)	
8F	1000 1111	MOD 110 R/M		(not used)	
8F	1000 1111	MOD 111 R/M		(not used)	
90	1001 0000			NOP	(exchange AX, AX)
91	1001 0001			XCHG	AX, CX
92	1001 0010			XCHG	AX, DX
93	1001 0011			XCHG	AX, BX
94	1001 0100			XCHG	AX, SP
95	1001 0101			XCHG	AX, BP
96	1001 0110			XCHG	AX, SI
97	1001 0111			XCHG	AX, DI
98	1001 1000			CBW	
99	1001 1001			CWD	
9A	1001 1010	DISP-LO	DISP-HI, SEG-LO, SEG-HI	CALL	FAR PROC
9B	1001 1011			WAIT	
9C	1001 1100			PUSHF	
9D	1001 1101			POPF	
9E	1001 1110			SAHF	
9F	1001 1111			LAHF	
A0	1010 0000	ADDR-LO	ADDR-HI	MOV	AL, MEM8
A1	1010 0001	ADDR-LO	ADDR-HI	MOV	AX, MEM16
A2	1010 0010	ADDR-LO	ADDR-HI	MOV	MEM8, AL
A3	1010 0011	ADDR-LO	ADDR-HI	MOV	MEM16, AL
A4	1010 0100			MOVS	DEST-STR8, SRC-STR8
A5	1010 0101			MOVS	DEST-STR16, SRC-STR16

第一字节 HEX	第一字节 BINARY	第二字节	第3,4,5,6字节	ASM-86 指令格式	
A6	1010 0110			CMPS	DEST-STR8, SRC-STR8
A7	1010 0111			CMPS	DEST-STR16, SRC-STR16
A8	1010 1000	DATA-8		TEST	AL, IMMED8
A9	1010 1001	DATA-LO	DATA-HI	TEST	AX, IMMED16
AA	1010 1010			STOS	DEST-STR8
AB	1010 1011			STOS	DEST-STR16
AC	1010 1100			LODS	SRC-STR8
AD	1010 1101			LODS	SRC-STR16
AE	1010 1110			SCAS	DEST-STR8
AF	1010 1111			SCAS	DEST-STR16
B0	1011 0000	DATA-8		MOV	AL, IMMED8
B1	1011 0001	DATA-8		MOV	CL, IMMED8
B2	1011 0010	DATA-8		MOV	DL, IMMED8
B3	1011 0011	DATA-8		MOV	BL, IMMED8
B4	1011 0100	DATA-8		MOV	AH, IMMED8
B5	1011 0101	DATA-8		MOV	CH, IMMED8
B6	1011 0110	DATA-8		MOV	DH, IMMED8
B7	1011 0111	DATA-8		MOV	BH, IMMED8
B8	1011 1000	DATA-LO	DATA-HI	MOV	AX, IMMED16
B9	1011 1001	DATA-LO	DATA-HI	MOV	CX, IMMED16
BA	1011 1010	DATA-LO	DATA-HI	MOV	DX, IMMED16
BB	1011 1011	DATA-LO	DATA-HI	MOV	BX, IMMED16
BC	1011 1100	DATA-LO	DATA-HI	MOV	SP, IMMED16
BD	1011 1101	DATA-LO	DATA-HI	MOV	BP, IMMED16
BE	1011 1110	DATA-LO	DATA-HI	MOV	SI, IMMED16
BF	1011 1111	DATA-LO	DATA-HI	MOV	DI, IMMED16
C0	1100 0000			(not used)	
C1	1100 0001			(not used)	
C2	1100 0010	DATA-LO	DATA-HI	RET	IMMED16 (interseg)
C3	1100 0011			RET	(intersegment)
C4	1100 0100	MOD REG R/M	(DISP-LO), (DISP-HI)	LES	REG16, MEM16
C5	1100 0101	MOD REG R/M	(DISP-LO), (DISP-HI)	LDS	REG16, MEM16
C6	1100 0110	MOD 000 R/M	(DISP-LO), (DISP-HI), DATA-8	MOV	MEM8, IMMED8
C6	1100 0110	MOD 001 R/M		(not used)	
C6	1100 0110	MOD 010 R/M		(not used)	
C6	1100 0110	MOD 011 R/M		(not used)	
C6	1100 0110	MOD 100 R/M		(not used)	
C6	1100 0110	MOD 101 R/M		(not used)	
C6	1100 0110	MOD 110 R/M		(not used)	
C6	1100 0110	MOD 111 R/M		(not used)	
C7	1100 0111	MOD 000 R/M	(DISP-LO), (DISP-HI), DATA-LO, DATA-HI	MOV	MEM16, IMMED16
C7	1100 0111	MOD 001 R/M		(not used)	
C7	1100 0111	MOD 010 R/M		(not used)	
C7	1100 0111	MOD 011 R/M		(not used)	
C7	1100 0111	MOD 100 R/M		(not used)	

第一字节 HEX	第一字节 BINARY	第二字节	第3、4、5、6字节	ASM-86 指令格式	
C7	1100 0111	MOD 101 R/M		(not used)	
C7	1100 0111	MOD 110 R/M		(not used)	
C7	1100 0111	MOD 111 R/M		(not used)	
C8	1100 1000			(not used)	
C9	1100 1001			(not used)	
CA	1100 1010	DATA-LO	DATA-HI	RET	IMMED16 (intersegment)
CB	1100 1011			RET	(intersegment)
CC	1100 1100			INT	3
CD	1100 1101	DATA-8		INT	IMMED8
CE	1100 1110			INTO	
CF	1100 1111			IRET	
D0	1101 0000	MOD 000 R/M	(DISP-LO), (DISP-HI)	ROL	REG8/MEM8,1
D0	1101 0000	MOD 001 R/M	(DISP-LO), (DISP-HI)	ROR	REG8/MEM8,1
D0	1101 0000	MOD 010 R/M	(DISP-LO), (DISP-HI)	RCL	REG8/MEM8,1
D0	1101 0000	MOD 011 R/M	(DISP-LO), (DISP-HI)	RCR	REG8/MEM8,1
D0	1101 0000	MOD 100 R/M	(DISP-LO), (DISP-HI)	SAL/SHL	REG8/MEM8,1
D0	1101 0000	MOD 101 R/M	(DISP-LO), (DISP-HI)	SHR	REG8/MEM8,1
D0	1101 0000	MOD 110 R/M		(not used)	
D0	1101 0000	MOD 111 R/M	(DISP-LO), (DISP-HI)	SAR	REG8/MEM8,1
D1	1101 0001	MOD 000 R/M	(DISP-LO), (DISP-HI)	ROL	REG16/MEM16,1
D1	1101 0001	MOD 001 R/M	(DISP-LO), (DISP-HI)	ROR	REG16/MEM16,1
D1	1101 0001	MOD 010 R/M	(DISP-LO), (DISP-HI)	RCL	REG16/MEM16,1
D1	1101 0001	MOD 011 R/M	(DISP-LO), (DISP-HI)	RCR	REG16/MEM16,1
D1	1101 0001	MOD 100 R/M	(DISP-LO), (DISP-HI)	SAL/SHL	REG16/MEM16,1
D1	1101 0001	MOD 101 R/M	(DISP-LO), (DISP-HI)	SHR	REG16/MEM16,1
D1	1101 0001	MOD 110 R/M		(not used)	
D1	1101 0001	MOD 111 R/M	(DISP-LO), (DISP-HI)	SAR	REG16/MEM16,1
D2	1101 0010	MOD 000 R/M	(DISP-LO), (DISP-HI)	ROL	REG8/MEM8,CL
D2	1101 0010	MOD 001 R/M	(DISP-LO), (DISP-HI)	ROR	REG8/MEM8,CL
D2	1101 0010	MOD 010 R/M	(DISP-LO), (DISP-HI)	RCL	REG8/MEM8,CL
D2	1101 0010	MOD 011 R/M	(DISP-LO), (DISP-HI)	RCR	REG8/MEM8,CL
D2	1101 0010	MOD 100 R/M	(DISP-LO), (DISP-HI)	SAL/SHL	REG8/MEM8,CL
D2	1101 0010	MOD 101 R/M	(DISP-LO), (DISP-HI)	SHR	REG8/MEM8,CL
D2	1101 0010	MOD 110 R/M		(not used)	
D2	1101 0010	MOD 111 R/M	(DISP-LO), (DISP-HI)	SAR	REG8/MEM8,CL
D3	1101 0011	MOD 000 R/M	(DISP-LO), (DISP-HI)	ROL	REG16/MEM16,CL
D3	1101 0011	MOD 001 R/M	(DISP-LO), (DISP-HI)	ROR	REG16/MEM16,CL
D3	1101 0011	MOD 010 R/M	(DISP-LO), (DISP-HI)	RCL	REG16/MEM16,CL
D3	1101 0011	MOD 011 R/M	(DISP-LO), (DISP-HI)	RCR	REG16/MEM16,CL
D3	1101 0011	MOD 100 R/M	(DISP-LO), (DISP-HI)	SAL/SHL	REG16/MEM16,CL
D3	1101 0011	MOD 101 R/M	(DISP-LO), (DISP-HI)	SHR	REG16/MEM16,CL
D3	1101 0011	MOD 110 R/M		(not used)	
D3	1101 0011	MOD 111 R/M	(DISP-LO), (DISP-HI)	SAR	REG16/MEM16,CL
D4	1101 0100	00001010		AAM	
D5	1101 0101	00001010		AAD	
D6	1101 0110			(not used)	

(续)

第一字节 HEX	第一字节 BINARY	第二字节	第3、4、5、6字节	ASM-86 指令格式
D7	1101 0111			XLAT SOURCE-TABLE
D8	1101 1000	MOD 000 R/M		
	1XXX	MODYYY R/M	(DISP-LO), (DISP-HI)	ESC OPCODE, SOURCE
DF	1101 1111	MOD 111 R/M		
E0	1110 0000	IP-INC-8		LOOPNE/ LOOPNZ SHORT-LABEL
E1	1110 0001	IP-INC-8		LOOPE/ LOOPZ SHORT-LABEL
E2	1110 0010	IP-INC-8		LOOP SHORT-LABEL
E3	1110 0011	IP-INC-8		JCXZ SHORT-LABEL
E4	1110 0100	DATA-8		IN AL, IMMED8
E5	1110 0101	DATA-8		IN AX, IMMED8
E6	1110 0110	DATA-8		OUT AL, IMMED8
E7	1110 0111	DATA-8		OUT AX, IMMED8
E8	1110 1000	IP-INC-LO	IP-INC-HI	CALL NEAR-PROC
E9	1110 1001	IP-INC-LO	IP-INC-HI	JMP NEAR-LABEL
EA	1110 1010	IP-LO	IP-H1, CS-LO, CS-HI	JMP FAR-LABEL
EB	1110 1011	IP-INC8		JMP SHORT-LABEL
EC	1110 1100			IN AL, DX
ED	1110 1101			IN AX, DX
EE	1110 1110			OUT AL, DX
EF	1110 1111			OUT AX, DX
F0	1111 0000			LOCK (prefix)
F1	1111 0001			(not used)
F2	1111 0010			REPNE/REPNZ
F3	1111 0011			REP/REPE/REPZ
F4	1111 0100			HLT
F5	1111 0101			CMC
F6	1111 0110	MOD 000 R/M	(DISP-LO), (DISP-HI) DATA-8	TES REG8/MEM8, IMMED8
F6	1111 0110	MOD 001 R/M		(not used)
F6	1111 0110	MOD 010 R/M	(DISP-LO), (DISP-HI)	NOT REG8/MEM8
F6	1111 0110	MOD 011 R/M	(DISP-LO), (DISP-HI)	NEG REG8/MEM8
F6	1111 0110	MOD 100 R/M	(DISP-LO), (DISP-HI)	MUL REG8/MEM8
F6	1111 0110	MOD 101 R/M	(DISP-LO), (DISP-HI)	IMUL REG8/MEM8
F6	1111 0110	MOD 110 R/M	(DISP-LO), (DISP-HI)	DIV REG8/MEM8
F6	1111 0110	MOD 111 R/M	(DISP-LO), (DISP-HI)	IDIV REG8/MEM8
F7	1111 0111	MOD 000 R/M	(DISP-LO), (DISP-HI) DATA-LO, DATA-HI	TEST REG16/MEM16, IMMED16
F7	1111 0111	MOD 001 R/M		(not used)
F7	1111 0111	MOD 010 R/M	(DISP-LO), (DISP-HI)	NOT REG16/MEM16
F7	1111 0111	MOD 011 R/M	(DISP-LO), (DISP-HI)	NEG REG16/MEM16
F7	1111 0111	MOD 100 R/M	(DISP-LO), (DISP-HI)	MUL REG16/MEM16
F7	1111 0111	MOD 101 R/M	(DISP-LO), (DISP-HI)	IMUL REG16/MEM16
F7	1111 0111	MOD 110 R/M	(DISP-LO), (DISP-HI)	DIV REG16/MEM16
F7	1111 0111	MOD 111 R/M	(DISP-LO), (DISP-HI)	IDIV REG16/MEM16
F8	1111 1000			CLC

第一字节		第二字节	第3、4、5、6字节	ASM-86 指令格式	
HEX	BINARY				
F9	1111 1001			STC	
FA	1111 1010			CLI	
FB	1111 1011			STI	
FC	1111 1100			CLD	
FD	1111 1101			STD	
FE	1111 1110	MOD 000 R/M	(DISP-LO), (DISP-HI)	INC	REG8/MEM8
FE	1111 1110	MOD 001 R/M	(DISP-LO), (DISP-HI)	DEC	REG8/MEM8
FE	1111 1110	MOD 010 R/M		(not used)	
FE	1111 1110	MOD 011 R/M		(not used)	
FE	1111 1110	MOD 100 R/M		(not used)	
FE	1111 1110	MOD 101 R/M		(not used)	
FE	1111 1110	MOD 110 R/M		(not used)	
FE	1111 1110	MOD 111 R/M		(not used)	
FF	1111 1111	MOD 000 R/M	(DISP-LO), (DISP-HI)	INC	MEM16
FF	1111 1111	MOD 001 R/M	(DISP-LO), (DISP-HI)	DEC	MEM16
FF	1111 1111	MOD 010 R/M	(DISP-LO), (DISP-HI)	CALL	REG16/MEM16 (inter)
FF	1111 1111	MOD 011 R/M	(DISP-LO), (DISP-HI)	CALL	MEM16 (intersegment)
FF	1111 1111	MOD 100 R/M	(DISP-LO), (DISP-HI)	JMP	REG16/MEM16 (inter)
FF	1111 1111	MOD 101 R/M	(DISP-LO), (DISP-HI)	JMP	MEM16 (intersegment)
FF	1111 1111	MOD 110 R/M	(DISP-LO), (DISP-HI)	PUSH	MEM16
FF	1111 1111	MOD 111 R/M		(not used)	

附 录 C

ASCII 代 码

1. 非打印 ASCII 字符

hex	abrev	intent	hex	abrev	intent
00	NUL	null or time fill	10	DLE	data line escape
01	SOH	start of heading	11	DC1	device control 1 (X ON)
02	STX	start of text	12	DC2	device control 2 (TAPE)
03	ETX	end of text	13	DC3	device control 3 (X-OFF)
04	EOT	end of transmission	14	DC4	device control 4 (TAPE)
05	ENQ	enquiry	15	NAK	negative acknowledge
06	ACK	acknowledge	16	SYN	synchronous idle
07	BEL	bell	17	ETB	end of transmission blocks
08	BS	backspace	18	CAN	cancel
09	HT	horizontal tabulation	19	EM	end of medium
0A	LF	line feed	1A	SUB	substitute
0B	VT	vertical tabulation	1B	ESC	escape
0C	FF	form feed	1C	FS	file separator
0D	CR	carriage return	1D	GS	group separator
0E	SO	shift out	1E	RS	record separator
0F	SI	shift in	1F	US	unit separator
			7F	DEL	delete

2. 可打印 ASCII 字符

hex	char	hex	char	hex	char	hex	char	hex	char	hex	char
20		30	0	40	@	50	P	60	`	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	/	6C	l	7C	|
2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o		