

# 《8086/8088 汇编语言程序设计》

编著：唐宁九、李征、张钢、郑成明、周群彪

XXX 出版社

X年X月X日

序言.....	5
第1章 引言.....	8
1.1 二进制编码与计算机系统.....	8
1.2 汇编语言基本概念与术语.....	9
1.3 汇编语言在计算机科学中的位置.....	10
习题1.....	12
第2章 数与编码.....	13
2.1 进位记数制.....	13
2.2 不同进位记数制间的转换.....	15
2.2.1 十进制与R进制间相互转换.....	15
2.2.2 二进制与八进制、十六进制间相互转换.....	20
2.3 编码.....	22
2.3.1 无符号数.....	23
2.3.2 原码.....	24
2.3.3 补码.....	26
2.3.4 ASCII码.....	32
2.3.5 BCD码.....	34
习题2.....	36
第3章 计算机系统模型.....	37
3.1 计算机系统的基本结构.....	37
3.1.1 中央处理器.....	38
3.1.2 内部存储器.....	40
3.1.3 系统总线.....	40
3.1.4 设备接口.....	41
3.1.5 外部设备.....	43
3.2 计算机系统存储单元.....	43
3.3 控制信号与时序过程.....	46
3.3.1 控制信号.....	47
3.3.2 时序过程.....	49
3.4 机器指令系统.....	50
习题3.....	52
第4章 8086/8088 CPU.....	53
4.1 8086/8088CPU 基本结构与工作原理.....	53
4.2 8086/8088CPU 的寄存器组.....	58
4.2.1 数据寄存器组.....	58
4.2.2 段寄存器组.....	59
4.2.3 地址指针寄存器组.....	60
4.2.4 控制寄存器.....	61
4.2.5 寄存器的隐含使用与特定使用.....	65
习题4.....	66
第5章 8086/8088 基本指令系统.....	67
5.1 汇编指令的基本格式.....	67
5.2 寻址方式.....	69
5.2.1 寄存器寻址方式.....	69

5. 2. 2 立即数寻址方式.....	69
5. 2. 3 存储器寻址方式.....	70
5. 2. 4 其它寻址方式.....	74
5. 3 基本指令系统.....	75
5. 3. 1 传送类指令.....	75
5. 3. 2 算术运算类指令.....	82
5. 3. 3 位操作类指令.....	90
5. 3. 4 处理器控制类指令.....	102
习题 5.....	106
第 6 章 汇编语言源程序组织.....	110
6.1 汇编语言的语句种类和格式.....	110
6.1.1 指令语句.....	110
6.1.2 伪指令语句.....	112
6.1.3 标识符.....	112
6.2 常量与变量.....	113
6.2.1 常量.....	113
6.2.2 简单变量定义.....	115
6.2.3 标号和内存变量的属性及属性操作符.....	119
6.3 汇编语言的表达式.....	124
6.3.1 数值表达式.....	124
6.3.2 地址表达式.....	127
6.4 段定义伪指令与源程序框架.....	128
6.4.1 段定义伪指令.....	129
6.4.2 段声明伪指令的段初值.....	131
6.4.3 IP 和 SP 的初值.....	132
6.4.4 源程序的基本框架.....	132
6.5 编制汇编语言程序的完整过程.....	134
6.5.1 编程工具及经典过程.....	134
6.5.2 用 UltraEdit 设置简易的汇编语言编程环境.....	135
6.5.3 DEBUG 常用命令简介.....	137
习题 6.....	142
第 7 章 分支与循环程序设计.....	146
7. 1 无条件转移指令.....	146
7. 2 条件转移指令.....	149
7. 3 分支程序设计.....	153
7. 4 循环控制指令.....	157
7. 5 循环程序设计.....	158
习题 7.....	162
第 8 章 子程序设计与系统调用.....	164
8. 1 子程序调用与返回指令.....	164
8. 2 子程序设计.....	170
8. 2. 1 子程序设计的一般规范.....	170
8. 2. 2 子程序设计示例.....	172
8. 3 系统调用.....	182

8.3.1 系统调用的概念.....	182
8.3.2 常用的系统调用.....	183
8.3.3 系统调用示例.....	186
习题 8.....	190
第 9 章 数值运算程序设计.....	191
9.1 二进制乘除法运算指令.....	191
9.1.1 乘法运算指令.....	191
9.1.2 除法运算指令.....	193
9.2 BCD 码加减法指令.....	195
9.2.1 BCD (Binary-Coded Decimal) 码.....	195
9.2.2 BCD 码加减法指令.....	196
9.3 BCD 码加减法程序设计原理与实现.....	200
9.4 BCD 码乘除法调整指令.....	201
9.4.1 BCD 码乘法调整指令 AAM.....	201
9.4.2 BCD 码除法调整指令 AAD.....	202
习题 9.....	203
第 10 章 非数值处理程序设计.....	204
10.1 串操作指令.....	205
10.1.1 串操作指令的特点.....	205
10.1.2 串操作指令.....	207
10.2 串操作指令的应用.....	212
10.2.1 串操作指令在程序中的使用要点.....	212
10.2.2 程序设计举例.....	214
10.3 其它非数值处理程序设计实例.....	216
习题 10.....	221
第 11 章 输入/输出程序设计.....	222
11.1 输入/输出指令.....	222
11.1.1 I/O 端口的编址方法.....	222
11.1.2 输入/输出指令.....	223
11.1.3 I/O 端口的寻址方式.....	223
11.2 主机与外设数据传送的方式.....	224
11.2.1 无条件传送方式.....	224
11.2.2 程序查询传送方式.....	224
11.2.3 中断传送方式.....	224
11.2.4 直接存储器传送方式(DMA).....	225
11.3 中断.....	225
11.3.1 中断的一般概念.....	225
11.3.2 中断源及中断类型码.....	225
11.3.3 中断向量表.....	227
11.3.4 中断优先级.....	227
11.3.5 中断过程.....	228
11.4 几个常用 BIOS 中断调用.....	229
11.4.1 键盘中断调用 (INT 16H).....	229
11.4.2 显示中断调用 (INT 10H).....	230

11.4.3 打印中断调用 (INT 17H) .....	231
11.4.4 串行通信中断调用 (INT 14H) .....	232
11.4.5 时间中断调用 (INT 1AH) .....	233
11.5 几个常用的 DOS 系统功能调用 (INT 21H) .....	233
11.6 应用举例.....	236
习题 11.....	243

内部资料，不得外传！

# 序言

随着计算机科学的发展，各类程序设计语言也得到不断的充实和更新；随着计算机应用的广阔拓展，各类程序设计语言的应用领域和前景也得到相应深化和发展。汇编语言，作为最接近计算机底层的程序设计语言，在计算机理论与应用不断更新和提升时，并没有被发展的浪潮所淹没，而且其理论与应用也在不断得到升华。汇编语言作为深入理解计算机系统原理的一种原理性程序设计语言，在计算机学科内仍然保持其存在的必要性，并且，因为它作为衔接硬件与软件的中介，其存在的必要性将长期持续下去；在各类对速度要求较苛刻的计算机应用系统中，核心代码仍然大多采用汇编语言编写，以提高其执行效率，部分针对硬件采取特殊操作的应用程序也是采用汇编语言编写，充分说明了汇编语言在应用中仍然存在的必要性。汇编语言在计算机科学的理论与应用中都有存在的必要性，同时需要强调，汇编语言存在还具有相当的重要性。汇编语言是深入理解计算机底层结构的基础理论，如果对它感兴趣，那么计算机底层世界的大门将会敞开，如果惧怕甚至反感它，那么它将成为初学者深入计算机底层世界永远的绊脚石。

现代计算机所使用的信号都是数字信号，都是“数字计算机”。通常情况下，数字计算机能直接识别、执行或处理的指令或数据，是 0 和 1 的二进制编码。查阅 Intel、AMD 或任意一款处理器手册，我们会看到在描述每一条指令时，必然会介绍该指令的二进制代码，换言之，如果期望计算机执行某条指令的功能，唯一的方法就是将该指令对应的二进制代码送入处理器让它被解释、被执行。这种处理器指令的二进制代码就是机器指令，是计算机唯一能直接解释、执行的指令，机器指令的全集即是机器语言，而汇编语言正是它的助记版本。用机器指令编写的程序是计算机唯一能够直接识别并执行的程序，而其他语言编写的程序必须转换成机器语言程序才能被执行。因此，机器语言程序又被称为目标程序；更重要的推论是，计算机的一切命令和高级语言的函数、过程等，都是由机器语言组成的。任何语言都是交流的工具；计算机编程语言，无论是 C/C++，Delphi，JAVA，还是汇编语言，都是人和计算机交流的工具，即人向计算机传达意图的工具，用于描述我们期望计算机怎么做，预置应对变化的对策等。我们会看到，计算机能够解释、执行的指令数量是非常有限的，似乎很难和现代电脑的强大及精彩有什么联系。但正如无数乐曲都是由 7 个音符组成的一样，所有的程序都是机器指令的不同组合。

理论上，我们完全可以使用机器指令编程，业界前辈在上世纪 40 年代计算机问世时正

是这样做的；然而二进制代码编写的程序尽管是计算机可直接识别和执行的，但人使用起来却非常吃力：每条 0 和 1 的代码是那么相似，几乎每读一条指令都需要查指令对照表，或者将对照表全部背诵下来。在现代程序设计中，这种担心已经没有必要了，因为人们很容易就会想到，应该用符号来代替不易识别的二进制代码，最自然的符号就是人类语言中的字或词，8086/8088 汇编语言采用的是英文单词缩写。这些代替机器指令的“符号”被称为“助记符”。以助记符替代机器指令为主体的编程语言，就是汇编语言。由于是用助记符逐条替代机器指令，即指令助记符和机器指令是简单的一一对应的关系，所以汇编语言又被称为“符号化的机器语言”；因为直接使用机器指令编程是极不实用的，所以汇编语言才是具有实用价值的最古老的计算机编程语言。设计程序时，先使用助记符来编写程序，编写完成后，再使用对应的二进制代码替换助记符。显然，用这种方式编程的效率和程序的可读性会提高很多。

从计算机诞生到上世纪 80 年代初期，由于汇编语言是面向机器的语言，其应用几乎是所向披靡的，特别是其不可比拟的时空效率，使汇编语言在编程工具中有着不可撼动的王者地位；高级语言是面向问题的语言，方便人的使用，但编译必须考虑问题的一般性导致代码相对庞大。然而随着计算机硬件平台性能的不不断提升，汇编语言时空效率的优势变得越来越不重要了；随着计算机应用的飞速发展，面向不同应用的编程语言具有更高的编程效率和更好的可读性，到今天，汇编语言仿佛已经不是一种主流的编程工具了。那么，这里存在一个不可回避的问题是：为什么还要深入理解汇编语言？

第一，汇编语言是现代计算机原理的重要组成部分，是计算机硬件和软件衔接点，是深入计算机底层世界的基础。学习汇编语言，能使我们对计算机组成原理、编译原理等计算机的基础理论有准确的认识；第二，对涉及系统和底层的开发和编程有极其重要和有益的帮助；第三，尽管汇编语言已经不是主流的编程工具，但在一些关键环节，汇编语言的优势还是非常重要的。例如在重要软件的核心环节，由于使用频度高，如果代码是用汇编语言编写的，会有效地提高运行效率；第四，使用汇编语言进行程序设计，会使系统对硬件平台性能的要求降低，即在有效节约成本方面也具有独到的作用。

和其他编程语言不同，汇编语言（或机器语言）和某种或某系列的 CPU 是密切相关的，不同 CPU 的汇编语言程序是不能直接换用的，这也是汇编语言和其他编程语言不能相比的重要缺点。但是，对于汇编语言的学习过程，我们只能选学某种汇编语言，在其中去了解和体验汇编语言中最普遍的东西。相对于现代处理器而言，8086/8088 CPU 结构简单，指令系统功能也相对简单，但是麻雀虽小，五脏俱全。例如，8086/8088 CPU 中具有存储器的分段管理模式，能够体现逻辑地址与物理地址的关系；具有标志寄存器，能够体现分支或循环结

构程序设计的硬件基础；具有堆栈管理，能够体现子程序调用、返回或中断调用、返回的硬件机制基础；具有能够同时完成无符号数、补码运算的加减指令，能够体现一条指令具有多种含义的特点；具有 BCD 码运算调整指令，能够体现底层数据的不同编码机制。这些特点都正是各种计算机系统、包括现代的 CPU 中具有的最普遍、最本质的原理，并且在学习和掌握 8086/8088 汇编语言的过程中所提炼的学习和思维方法，同样适用于其它 CPU 及其汇编语言。掌握了这样一些基本原理和基本学习方法后，自学其它更为复杂的汇编语言，将会成为不困难的事情。并且，由于桌面 PC 非常普及，Windows 操作系统通常自带 DEBUG 等 8086/8088 指令系统的模拟调试程序，各种版本的 8086/8088 汇编器、连接器通常都能在 PC 上运行，使用方便，所以介绍 8086/8088 汇编语言也是非常有利于上机实践的。

本教材的内容主要基于 8086/8088 指令系统，试图以全新的诠释方式来讲解这种汇编语言。就本教材涉及的教学内容而言，可作为计算机专业本科生的汇编语言程序设计教材。本教材的特点在于以下几点。第一，由于在初学者普遍感觉指令记忆很困难，而且枯燥，为了避免初学阶段记忆太多指令而抹杀了学习的乐趣，将指令系统中的指令分散到各个章节中去，使初学者在学完指令系统的基础部分后即可进行编程实践，使初学者能够循序渐进，保持知识积累与学习兴趣的并行提升；第二，增强有关 CPU 结构及工作原理的说明，但同时避免讨论其细节，使初学者能够尽快的为自己搭建一个基本合理的模型计算机，并借助它加深对汇编语言的理解，并且，在掌握汇编语言之后，借助此模型机深化对计算机系统原理的理解；第三，结合实际的教学经验，在总结初学者容易产生的疑问与容易出现的错误理解后，举例加以说明；第四，突出硬件与软件的衔接，充分说明 CPU 中关键器件与指令系统中核心指令与程序设计的关系，使初学者能够真正深入理解软件结构是如何由底层硬件结构支撑起来的；第五，教材与上机实践密切结合，在详细说明编写、汇编、调试程序方法的基础上，教材中所列举的例子都可以直接上机实现调试。

在这里，我们期望这本以新方式诠释的教材能够为汇编语言初学者提供一点微薄的帮助。由于编者水平有限，时间比较仓促，疏漏之处在所难免，欢迎广大读者提出宝贵意见！

唐宁九

2007 年 9 月 日



# 第 1 章 引言

## 1.1 二进制编码与计算机系统

在种类繁多的计算机应用中，我们能够见到丰富多彩的信息表达形式，也能够使用功能各异的应用程序。例如，我们可以使用各种媒体播放软件观赏众多的视频和音频文件，能够欣赏到五颜六色的画面和悦耳的音乐；可以使用各种字处理软件编辑文档或源程序，从而制作由不同文字组成的文本，这些软件提供的编辑、查找功能使我们能够方便的对文本进行修改、浏览；使用数据库管理软件使我们能够定义各种数据库，在数据库中存放种类丰富的数据。但是，计算机应用领域中这些形式各异的软件和数据形式在计算机底层的描述却具有共通性。这种共通性则是指，无论两个应用程序的功能或性质有着多么大的不同，无论两种数据形式的性质有着多么大的不同，它们都是使用由“0”和“1”两种数字信号的排列组合来表达的。一般而言，现代计算机硬件系统只能鉴别“0”和“1”这两种信号，看起来十分单一，但是通过不同的排列组合，这两种信号组成的序列可以表达丰富多彩的信息，而我们平时在计算机应用中见到的不同信息表达在底层正是这样表述的。这种在计算机系统中，由“0”、“1”两种信号排列组合而成的序列，我们称之为编码。

在计算机系统里的二进制编码是需要人来解释的，只有当人们按照某种既定的编码规则来解释其含义时，二进制编码才是有意义的。例如，在任意一个存储单元中，保存了“00110101”这样一个二进制编码，对它的解释可以有很多种，它可以是一帧图像中某个像素位置的亮度表示；可以是音频数据某个采样点上的音量幅度；可以是坐标系中的一个坐标值；可以是一个字母的编码；可以是另一数据的地址；也可以是 CPU（中央处理器）执行的一条机器指令。那么，初学者通常会问，它究竟具有什么含义呢？这个问题恰好忽略了提问者本身，软件的设计一定是由人来完成的，因此，系统中信息的解释也就是由人来定义的。如果我们把上述那个编码解释为像素的亮度值，我们会将它送到显示接口卡的显示存储器中，使它能够在屏幕上显示出来；如果我们把它解释为音频数据中的音量幅度，我们会将它传送到音频接口卡，使它能够在音箱中播放出来；如果我们认为它是一个坐标值，我们或许会将它用于坐标变换；如果我们把它看作是一个字符编码，我们会将它用于字符模板的查找，并将字符模板送到显示接口卡，最终将字体形状显示在屏幕上；如果我们认为它是程序中的一条指令，我们会将它送到 CPU 去解释并执行。

经过上面的阐述，我们认识到计算机系统底层中的信息是需要人来解释的。事实上，如果站在较高层面上，操作系统、应用程序这样软件也会帮助人们来解释这些信息，但归根结底信息的解释是由人来完成的。这个问题看起来似乎很容易理解？或许我们以前早就理解了？但是，当我们面对加法指令同时完成无符号数和补码两类加法运算、同时影响两类标志位时，我们可能就会忘记这样的问题，因此不能解释指令使用的究竟是什么编码，从而陷入困惑。在正式进入汇编语言的学习殿堂之前，应当建立起人解释底层编码含义的观念，为解决将来学习过程中的困惑做好准备。

## 1.2 汇编语言基本概念与术语

CPU 能够直接执行的程序，是由机器指令序列组成的程序，CPU 从内部存储器中逐条读取机器指令，并加以解释、执行，并且 CPU 会将这个过程不断持续下去。上一节已经阐述过，使用二进制编码表示的指令即是**机器指令**。例如，我们可以使用代码“001”表示加法操作，使用代码“010”表示减法操作，使用代码“011”表示逻辑与操作等等。机器指令是可执行程序最小组成单位，它具有原子性，CPU 要么完整的执行一条机器指令，要么完全不执行它。

首先，机器指令实际上是最底层的软件元素，那么它与汇编语言有什么关联呢？汇编语言中的**汇编指令**即是机器指令的助记版本，在汇编语言中使用英文单词缩写来表达指令功能，而不是使用二进制代码。例如，我们可以使用“add”表示加法操作，使用“sub”表示减法操作，使用“and”表示逻辑与操作等等。正因为如此，人们记忆汇编指令会比记忆机器指令容易得多。对于一种特定的 CPU，其机器指令的全集构成**机器指令系统**，其对应的汇编指令全集也就构成了**汇编指令系统**。原则上来说，机器指令与汇编指令是一一对应的关系，但是也不排除个别例外的情况。这种例外的情况在 8086/8088 指令系统中也是存在的，在后面的章节中会给出相应的说明。另外，应当注意，不同型号的 CPU 具有不同的指令系统。其次，汇编指令（或机器指令）和高级语言中的语句有什么区别和联系呢？我们可以通过如下的例子来感受一下：

**例 1.2.1** 如下的 C 语言语句与汇编语言程序片段具有相同的功能。

C 语言语句：

```
w = x+y-z;          /*x 变量加 y 变量，再减 z 变量，运算结果保存到 w 变量中*/
```

相应的汇编语言程序片段：

```
mov  al, x          ; 将变量 x 的内容传送到 al 寄存器
add  al, y          ; 将变量 y 的内容与 al 寄存器的内容相加，结果保存到 al 寄存器
sub  al, z          ; al 寄存器的内容与变量 z 的内容相减，结果保存到 al 寄存器
mov  w, al          ; al 寄存器的内容传送到变量 w 保存
```

从上面的例子可以看出，在 C 语言中一个赋值语句，如果使用 8086/8088CPU 的汇编指令系统表出，需要至少 4 条指令。这里可以得出这样的结论：高级语言中的语句是面向问题描述的，其表述形式更接近于人类描述问题的方式，语句虽相对复杂，但可读性、功能性均较强；汇编指令则不同，它们是面向机器操作的，其表述形式更接近于人类操作机器的方式，每条指令相对简单，功能性较弱，但对于不熟悉指令系统的初学者，程序片段的可读性并不高。如果程序较长，那么阅读和理解汇编语言程序比阅读和理解高级语言程序要困难很多。

无论是使用高级语言还是汇编语言所编写的程序，最初都是一种文本格式的程序，概念上称为**源程序**，而源程序是不能被执行的。使用汇编语言书写的源程序通常称为**汇编语言程序**。对应的，由机器指令组成的，能够被 CPU 直接解释、执行的程序称为**目标程序**（或目标代码）。源程序必须先经过翻译过程，转换为目标程序后，才能被 CPU 执行。对于高级语言，其语句与机器指令并不是一一对应的，因此这样的翻译过程比较复杂，称为编译过程，完成编译过程的应用软件称为**编译器**。对于汇编语言，其指令语句与机器指令基本上是一一对应的，因此翻译过程相对简单，称为**汇编过程**，完成汇编过程的应用软件称为**汇编器**（或**汇编程序**）。如果是将内部存储器中的机器指令序列翻译为汇编指令，以达到分析程序的目的，那么这样的过程称为**反汇编过程**。一个完整的汇编语言程序除了包括汇编指令系统中的指令外，还包括其它一些语法要素，而这些语法要素是与具体的机器指令系统无关的。例如，汇编语言中的**伪指令**，它们能够完成变量定义与空间分配、定义标号、定义常量、条件汇编、重复汇编等功能，但它们的解释与执行是由汇编器来完成的，而不是 CPU 直接完成的，并且，伪指令的执行阶段是在源程序的**汇编阶段**，而不是目标代码的**执行阶段**。因此，**汇编语言**是一种低级语言，它由某种特定 CPU 型号对应的汇编指令系统与其它一些与指令系统无关的语法要素、语法规则构成。

## 1.3 汇编语言在计算机科学中的位置

从上一小节的阐述中，我们明白了汇编语言与高级语言的区别，高级语言是面向问题描述的，其源程序容易被人们读懂，程序编写也相对容易，而汇编语言则是面向机器操作的，

其源程序阅读和理解相对困难，程序设计也相对繁琐。那么，我们为什么还要花费那么多精力学习它，花费那么多时间来做一件貌似事倍功半的事情呢？这与汇编语言在计算机科学中的特殊位置是紧密联系在一起的。下面将从理论和应用两个方面来阐述汇编语言在计算机科学中的位置。

首先，CPU 从产生至今，其核心原理中始终保持了周而复始的读取机器指令、解释机器指令、执行机器指令这样一个基本机制。只要这种机制保持存在，机器指令系统就会始终存在，因此，汇编语言也会随之继续存在下去。照目前的发展趋势来看，CPU 的这种基本机制会在相当长的时间内不会出现本质的改变。由于汇编指令实际上是机器指令的助记版本，因此学习汇编语言就是在主要学习机器指令系统。那么学习指令系统的必要性和重要性是什么呢？

总体上来说，计算机科学可以分为硬件和软件两个大的分支，而汇编语言正是衔接这两个分支的重要关节点。可以这样说，汇编语言既是深入学习计算机硬件工作原理的必要基石，也是理解硬件结构对各类软件实现支撑的关键原理所在。例如，计算机组成原理课程是一门深入剖析 CPU 内部结构及原理、深入阐述指令系统设计原理、深入分析指令微操作的课程，那么，如果不理解指令系统以及相关的计算机系统基本原理，学习这门课程的基础是不充分的。对于微机原理与接口技术这样的课程中，指令系统已被作为基本工具使用。换言之，如果没有掌握汇编语言，那么深入学习计算机系统的硬件工作原理几乎是不可能的。对于软件相关的课程，汇编语言与它们的学习没有直接的关联，但是如果没有掌握汇编语言，将影响对底层软件工作原理的理解，影响对软件优化的理解。

在现代计算机的应用领域，汇编语言的应用仍然是比较广泛的。首先，如果有必要跟踪或监控正在运行的应用程序，例如针对某些病毒程序的跟踪或分析，但是该应用程序又不是自己编写的，没有源代码，那么通常所使用的程序跟踪工具都是将机器指令反汇编为汇编指令。在这种情况下，只能通过阅读汇编语言程序来分析程序的功能。因此，汇编语言是调试、分析系统底层程序的基本工具。并且，目前一些对响应速度要求较高的系统中，其软件的核心部分都是使用汇编语言来编写的，因为相对于高级语言，汇编语言程序产生的机器指令序列要短小很多，执行效率也高出很多。因此，由于汇编语言程序无以伦比的时间、空间效率优势，在计算机应用中，它仍然是对高级语言程序的一种必要补充。

## 习题 1

1. 试简述汇编语言与高级语言的区别与联系。
2. 为什么说汇编语言是最接近计算机系统底层的语言？请结合自己的理解加以叙述。
3. 编译器与汇编器的概念是什么？它们有什么区别？
4. 简述机器指令、汇编指令的概念，并说明它们的区别与联系。

内部资料，不得外传！

## 第 2 章 数与编码

### 2.1 进位记数制

记数的概念是伴随着人们对数量概念的认识而产生的。人们最早在对物品记数时，为了区分物品数量的具体差异，可能会使用小石子的数量来代表体积较大的物品数量，这就是最原始的**记数概念**，但是数量太大以后，小石子可能越来越多，这样也就无法清晰的展示物品的数量。那么，人们可能很快想到用较大的石子来代替一组小石子，这样记数又重新清晰起来，这种代替操作就是原始的**进位概念**。使用进位来帮助记数，会使得记数结果非常清晰，**进位记数制**的概念由此产生。在小学阶段，我们学习基本的四则运算时，做加法总是逢十进一，那么我们实际上使用的是进位记数制中最普遍的一种，即是十进制，基于十进制的数字符号我们称为十进制数。我们在学习经典数学理论时，在日常生活中，所说、所见、所写的数字通常都是十进制数。那么，有没有其它的进位记数制呢？回答是肯定的。计算机系统内部使用的就是二进制编码，而二进制编码是建立在二进制数基础之上的。

**数与编码**是两个不同的概念，数是指理论上的数值，进位记数制正是数在理论上的符号表达形式，而编码则是数在数字系统中的表达形式。二者的区别将在 2.3 节讨论编码时给出详细的阐述。这里讨论进位记数制，是因为计算机系统所使用的进位记数制与人们通常使用的十进制不同，是为了清晰的展现二者间的区别与联系。使已经牢牢接受了十进制数的人们能够站在更高的层面上、从整体理论角度来重新认识进位记数制，使人们也能够接受二进制数，从而也才能深刻理解系统底层的编码理论、指令功能。在此基础上，也才能进一步接受在程序设计中常用的八进制、十六进制数。在讨论进位记数制的本质前，我们先来观察一下十进制数的多项式形式，如下例所示：

**例 2.1** 十进制数 5769.32 的多项式表达如下式所示。

$$5769.32 = 5 \times 10^3 + 7 \times 10^2 + 6 \times 10^1 + 9 \times 10^0 + 3 \times 10^{-1} + 2 \times 10^{-2}$$

从上例的多项式表达中，我们能够看出什么规律，抽象出怎样的一般概念呢？首先，按照多项式表达形式，我们可以抽象出“**系数**”、“**数位**”、“**权值**”、“**基数**”的概念。例中的 5、7、6、9、3、2 都是多项式中的系数，它们各自所在的“千位”、“百位”、“十位”、“个位”、“十

分位”、“百分位”即是各系数所在的数位。例中的  $10^3$ 、 $10^2$ 、 $10^1$ 、 $10^0$ 、 $10^{-1}$ 、 $10^{-2}$  则是数位对应的权值，它们是与数位对应起来的。可以观察到，权值表示为指数形式，其中“10”为“基数”，此 10 与“十进制”中的“十”是对应的，并非偶然。该数值完整的十进制表达为“5769.32”，我们可以容易的得出一般性的结论，十进制数表达形式其实是系数按照所在数位对应的权值大小，依降序排列而成的系数序列。系数同时也解释为表达数字的基本符号，那么在十进制中，这样的系数总共有多少呢？我们都很清楚，在十进制中，系数从 0 到 9，总共有 10 个系数符号，系数符号的数量等于基数。那么，我们会问，这些规律能够一般化吗？如果是 R 进制数，它的系数符号有多少个呢？它可以表达为这种多项式形式吗？我们先来看一个例子。

**例 2.2** 几种进位记数制的系数列举如下。

十进制数的系数：0, 1, 2, 3, 4, 5, 6, 7, 8, 9

二进制数的系数：0, 1

三进制数的系数：0, 1, 2

四进制数的系数：0, 1, 2, 3

八进制数的系数：0, 1, 2, 3, 4, 5, 6, 7

十六进制的系数：0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

我们可以得出一般性的推论，R 进制数的系数共有 R 个，从 0 到 R-1，但是要注意，单个系数必须是单个符号，例如十进制的“12”，在十进制中它不是单个系数，但在十六进制中它却是单个系数，用“C”表示。而数值 R 即为 R 进制数的基数。R 进制数的标准书写方法为：(系数序列)<sub>R</sub>。任意的 R 进制数都可以采用多项式来表出，如下例所示。

**例 2.3** 整数部分 n 位、小数部分 m 位的 R 进制数，可使用多项式表达形式如下。

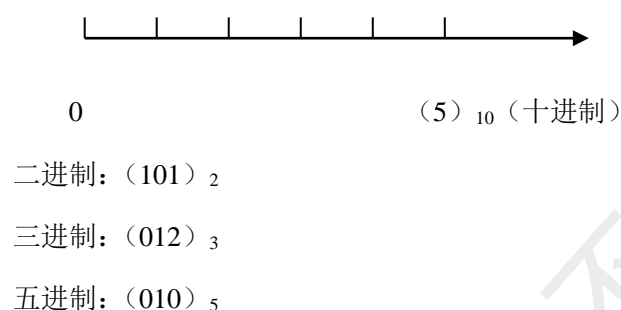
$$(K_{n-1}K_{n-2}\dots K_1K_0.K_{-1}K_{-2}\dots K_{-m})_R = K_{n-1}R^{n-1} + K_{n-2}R^{n-2} + \dots + K_1R^1 + K_0R^0 + K_{-1}R^{-1} + K_{-2}R^{-2} + \dots + K_{-m}R^{-m}$$

其中  $K_i$  为 R 进制系数，变化范围从 0 到 R-1。

## 2.2 不同进位记数制间的转换

在上一小节中，我们明确了进位记数制的概念，接下来，我们要讨论是不同进位记数制间的转换问题，这对于初学者理解二进制数是有益处的，并且，这样的数制转换在程序设计中也会经常使用到。首先，我们会提出这样的问题，理论上同一个数值是否可以用不同的进位记数制来表示？不同的进位记数制表示相同数值时得到的不同表达形式，其本质含义是什么？下面我们先看一个例子。

**例 2.4** 考虑十进制数“5”的在数轴上的位置，以及在其它进位记数制中，它的表达形式。



从上面的例子，我们可以看到，在数轴上距离原点 5 个单位距离的同样一个点，可以使用不同进位记数制表达出不同的形式。这种情况正如一个人可以为自己取很多个名字一样，但是在正规场合被认可的名字只有一个，就是身份证上的姓名；对于数也是一样，在计算机系统中，能够被认可的只能是二进制表达形式。那么，我们熟悉的十进制如何才能转换为计算机系统认可的二进制数呢？反过来，如果我们不习惯观察计算机系统提供的二进制数，那么怎样才能将它转换为十进制数呢？更一般的说，不同进位记数制间有无统一的相互转换方法呢？由于经常进行的转换都是将十进制数转换为  $R$  进制，或者将  $R$  进制数转换为十进制，而且任意两种进位记数制间的转换也能借助这两种基本转换来间接实现，因此，我们重点讨论这两种转换方法。

### 2.2.1 十进制与 $R$ 进制间相互转换

将十进制数转换为  $R$  进制的一般方法如下例所示。



**例 2.5** 将十进制数转换为 R 进制，可以考虑 R 进制的多项式表达形式（如例 2.3）。

$$(K_{n-1}K_{n-2}\dots K_1K_0.K_{-1}K_{-2}\dots K_{-m})_R = K_{n-1}R^{n-1} + K_{n-2}R^{n-2} + \dots + K_1R^1 + K_0R^0 + K_{-1}R^{-1} + K_{-2}R^{-2} + \dots + K_{-m}R^{-m}$$

对于整数部分，R 进制的系数可以按照由低位到高位顺序，通过逐次除以 R 得到，每次除法后得到的商作为下次除法的被除数，余数则是当前数位的系数：

$$(K_{n-1}R^{n-1} + \dots + K_0R^0)/R \quad \text{商:}(K_{n-1} \times R^{n-2} + \dots + K_1R^0) \quad \text{余数:}K_0$$

$$(K_{n-1}R^{n-2} + \dots + K_1R^0)_{10}/(R)_{10} \quad \text{商:}(K_{n-1}R^{n-3} + \dots + K_2R^0)_{10} \quad \text{余数:}K_1$$

.....

除法一直持续到商为 0 为止，最后一次的余数为最高位系数  $K_{n-1}$ 。这些系数是十进制的运算结果，同时在多项式形式的基础上，又可理解为 R 进制的系数，将系数由高到低排列起来，就得到 R 进制整数部分  $(K_{n-1}K_{n-2}\dots K_1K_0)_R$ 。

对于小数部分，R 进制的系数可以按照由高位到低位的顺序，通过逐次乘以 R 得到，每次乘法后得到的小数部分用作下一次乘法，整数部分则是当前数位的系数：

$$(K_{-1}R^{-1} + \dots + K_{-m}R^{-m})R \quad \text{整数:}K_{-1} \quad \text{小数:}(K_{-2}R^{-1} + \dots + K_{-m}R^{-m+1})$$

$$(K_{-2}R^{-1} + \dots + K_{-m}R^{-m+1})R \quad \text{整数:}K_{-2} \quad \text{小数:}(K_{-3}R^{-1} + \dots + K_{-m}R^{-m+2})$$

.....

乘法一直持续到整数部分为 0 或者小数部分陷入无限循环为止。这里需要注意，十进制中的有限小数在 R 进制中不一定有限，有可能是无限循环小数。也就是说，m 可能为无限大，对于无限循环的问题，大家可以作一个练习：将  $(0.2)_{10}$  转换为三进制表达形式。经过乘法所得的系数，既是十进制的乘法结果，也是 R 进制的系数，对于有限位小数的情况，最终我们能够得到 R 进制小数部分  $(0.K_{-1}K_{-2}\dots K_{-m})_R$ 。

注意，例中实现的运算都是十进制数运算，所得运算结果解释为 R 进制系数。不同进制的数不能混合在一起直接进行运算。对于上例的反转换过程，即将 R 进制数转换为十进制的原理，我们同样以例题的形式给出。

**例 2.6** 将 R 进制数转换十进制，考虑将 R 进制的多项式表达形式转换为十进制的多项式表达形式，然后采用熟悉的十进制运算将多项式合并起来。

$$\begin{aligned}
 (K_{n-1}K_{n-2}\dots\dots K_1K_0\cdot K_{-1}K_{-2}\dots\dots K_{-m})_R = & \\
 K_{n-1}R^{n-1} + K_{n-2}R^{n-2} + \dots\dots + K_1R^1 + K_0R^0 + K_{-1}R^{-1} + K_{-2}R^{-2} + \dots\dots + K_{-m}R^{-m} = & \\
 (K_{n-1})_{10}(R)_{10}^{n-1} + (K_{n-2})_{10}(R)_{10}^{n-2} + \dots\dots & \\
 + (K_1)_{10}(R)_{10}^1 + (K_0)_{10}(R)_{10}^0 + (K_{-1})_{10}(R)_{10}^{-1} + (K_{-2})_{10}(R)_{10}^{-2} + \dots\dots + (K_{-m})_{10}(R)_{10}^{-m} &
 \end{aligned}$$

此方法的关键在于将 R 进制多项式中的系数与基数先转换为十进制表达形式（通常情况下，为了转换方便，多项式中基数可直接写为十进制形式），然后再用十进制的乘法与加法将多项式合并起来，最终得到我们需要的十进制表达形式。这样，我们可以使用熟悉的十进制运算来完成转换。下面我们看两个实际例子，从具体实例中体会转换过程。

**例 2.7** 将十进制数  $(57.2)_{10}$  转换为二进制、八进制，然后分别重新转换为十进制形式。

(1) 由十进制转换为二进制

整数部分：

$$\begin{array}{r}
 2 \overline{)57} \quad 1 \\
 2 \overline{)28} \quad 0 \\
 2 \overline{)14} \quad 0 \\
 2 \overline{)7} \quad 1 \\
 2 \overline{)3} \quad 1 \\
 1 \quad \quad \quad
 \end{array}$$

使用短除法格式，自下向上将余数连接起来，就得到结果： $(111001)_2$ 。

小数部分：

$0.2 \times 2$  整数:0 小数:0.4

$0.4 \times 2$  整数:0 小数:0.8

$0.8 \times 2$  整数:1 小数:0.6

$0.6 \times 2$  整数:1 小数:0.2

已出现无限循环，仅保留目前所得的高位系数，将所得整数部分自上向下连接起来，得到转

换结果：(0011)<sub>2</sub>。

结合整数和小数转换结果，整体转换结果为：(111001.0011)<sub>2</sub>。

(2) 由所得的二进制转换结果转换回十进制

$$\begin{aligned} (111001.0011)_2 &= \\ 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 1 \times 2^{-4} &= \\ (1)_{10} \times (2)_{10}^5 + (1)_{10} \times (2)_{10}^4 + (1)_{10} \times (2)_{10}^3 + (0)_{10} \times (2)_{10}^2 + (0)_{10} \times (2)_{10}^1 + \\ (1)_{10} \times (2)_{10}^0 + (0)_{10} \times (2)_{10}^{-1} + (0)_{10} \times (2)_{10}^{-2} + (1)_{10} \times (2)_{10}^{-3} + (1)_{10} \times (2)_{10}^{-4} &= \\ (32)_{10} + (16)_{10} + (8)_{10} + (1)_{10} + (0.125)_{10} + (0.0625)_{10} &= (57.1875)_{10} \end{aligned}$$

转换回十进制后，小数部分出现了误差，是因为在上一步由十进制转换为二进制时，遇到了小数无限循环的情况，二进制转换结果中只保留了有限位小数的缘故。

(3) 由十进制转换为八进制

整数部分：

$$\begin{array}{r} 8 \overline{) 57} \quad 1 \\ 8 \overline{) 7} \quad 7 \\ 0 \end{array}$$

整数部分转换结果为：(71)<sub>8</sub>。

小数部分：

$$0.2 \times 8 \quad \text{整数:1} \quad \text{小数:0.6}$$

$$0.6 \times 8 \quad \text{整数:4} \quad \text{小数:0.8}$$

$$0.8 \times 8 \quad \text{整数:6} \quad \text{小数:0.4}$$

$$0.4 \times 8 \quad \text{整数:3} \quad \text{小数:0.2}$$

已陷入无限循环，仅保留有限位小数：(0.1463)<sub>8</sub>。

整体转换结果为：(71.1463)<sub>8</sub>。

(4) 由八进制结果转换回十进制

$$\begin{aligned}
(71.1463)_8 &= 7 \times 8^1 + 1 \times 8^0 + 1 \times 8^{-1} + 4 \times 8^{-2} + 6 \times 8^{-3} + 3 \times 8^{-4} = \\
&= (7)_{10} \times (8)_{10}^1 + (1)_{10} \times (8)_{10}^0 + (1)_{10} \times (8)_{10}^{-1} + (4)_{10} \times (8)_{10}^{-2} + (6)_{10} \times (8)_{10}^{-3} + (3)_{10} \times (8)_{10}^{-4} = \\
&= (56)_{10} + (1)_{10} + (0.125)_{10} + (0.0625)_{10} + (0.0117)_{10} + (0.0007)_{10} = \\
&= (57.1999)_{10}
\end{aligned}$$

转换误差来源于前一步，即十进制转换为八进制的结果。

**例 2.8** 将十进制数  $(28)_{10}$  转换为十六进制，并将转换结果重新转换为十进制。

(1) 由十进制转换为十六进制

该数只有整数部分，转换过程如下：

$$\begin{array}{r}
16 \overline{) 28} \quad C \\
16 \underline{) 1} \quad 1 \\
0
\end{array}$$

转换结果为  $(1C)_{16}$ 。在基数大于 10 的记数制中，注意单个系数必须是单个符号。

(2) 由十六进制转换结果转换回十进制

$$\begin{aligned}
(1C)_{16} &= 1 \times 16^1 + C \times 16^0 = (1)_{10} \times (16)_{10}^1 + (12)_{10} \times (16)_{10}^0 = \\
&= (16)_{10} + (12)_{10} = (28)_{10}
\end{aligned}$$

由基数大于 10 的进位记数制向十进制转换，需要注意单个系数转换为十进制可能出现多位的情况，这里的“C”转换为“12”就是一例。

通过上面两个例子，我们理解了十进制与 R 进制表达形式的相互转换方法，也通过这些例子接受了除十进制外的其它进制数的表达形式，特别是二进制。我们必须深刻理解，计算机系统各种编码都是基于二进制数的。同时，我们也应该知道，在汇编语言源程序或高级语言源程序中，我们可以使用十进制表达形式（本质上是十进制的字符串，是由字符编码构成的），经过汇编器汇编或编译器编译后得到的目标代码中，它们都被转换为二进制编码形式。在源程序中，八进制和十六进制表达形式也是经常使用的，特别是十六进制数，它常用于表达逻辑地址。二进制、八进制和十六进制间的联系比它们与十进制间的联系更紧密，转换过程也更简单，下面我们将阐明二进制、八进制和十六进制间的相互转换方法。在本小节结束前，我们指出，在以后的讨论中，进位记数制均使用下标指出，如果没有特别指明数的进位记数制，则表示该数为十进制。

## 2.2.2 二进制与八进制、十六进制间相互转换

在这一小节中，我们将讨论二进制与八进制、十六进制间的相互转换方法。在具体讨论转换方法前，让我们先来分析一下不同进位记数制中系数数量的对应关系。例如在十进制数中，一个系数的变化范围从 0 到 9，总共 10 个系数符号，那么需要多少个连续数位上的二进制系数才能表达这个变化范围呢？我们知道，如果使用 3 个连续数位的二进制系数，能够表达的不同数值为  $2^3=8$  个，不够表示十进制的 10 个系数符号；如果使用 4 个连续数位的二进制数，能够表达的不同数值为  $2^4=16$  个，能够表示十进制的 10 个系数符号，但又出现 6 个多余的符号。因此，十进制系数与二进制系数间的数量对应关系不是整数关系，这两种进位记数制间的转换没有简便的方法，只能按照 2.2.1 节中提供的方法进行转换。接下来，我们来观察一下八进制、十六进制系数与二进制系数间的数量对应关系。一个八进制系数所能表达的符号数量为 8 个，这正好与三位连续的二进制系数所能表达的系数数量  $2^3=8$  一致，因此，我们可以说一个八进制系数正好对应三位连续数位的二进制系数。与此相似，一个十六进制系数则对应四位连续数位的二进制系数。如果两种不同进位记数制间存在着整倍的系数数量对应关系，那么它们之间的转换将是便捷的。二进制与八进制间的转换，可以参照下面的例子。

**例 2.9** 将二进制数  $(01011010)_2$  转换为八进制数，并完成逆转换。

(1) 将二进制数  $(01011010)_2$  转换为八进制数

将二进制数按照由低位到高位顺序，连续三位编为一组，若高位编组不足三位，则在高位添零。分组完毕后直接将二进制系数编组转换为八进制系数即可。

二进制三位编组：001    011    010

八进制系数：        1        3        2

转换得到的八进制数为  $(132)_8$ 。

为了清楚地理解转换原理的正确性，可以考察二进制数的多项式形式：

$$\begin{aligned} & 0 \times 2^8 + 0 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = \\ & (0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0) \times 2^6 + (0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0) \times 2^3 + (0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0) \times 2^0 = \\ & 1 \times 8^2 + 3 \times 8^1 + 2 \times 8^0 \end{aligned}$$

## (2) 完成逆转换

将八进制数中的各个系数转换为二进制系数分组,最后将二进制系数分组拼接在一起即可完成转换。

八进制系数: 1 3 2

二进制编组: 001 011 010

很容易得到二进制转换结果  $(001011010)_2$ 。

二进制与十六进制间的转换,可以参考下例。

**例 2.10** 将二进制数  $(11101001)_2$  转换为十六进制数,并完成逆转换。

### (1) 二进制转换为十六进制

转换过程与二进制转换为八进制类似,只是将二进制系数编为 4 为一组。

二进制 4 位编组: 1110 1001

十六进制系数: E 9

转换得到的十六进制数为  $(E9)_{16}$ 。

### (2) 逆过程

与八进制转换为二进制类似,将系数转换为二进制分组,拼接分组即可完成转换。

十六进制系数: E 9

二进制编组: 1110 1001

容易得到转换结果  $(11101001)_2$ 。

二进制编码是计算机系统唯一能够识别的编码,而二进制编码是建立在二进制数基础上的,且由于十进制是人类习惯的表达方式,掌握二进制数与十进制数的相互转换是理解二进制数与编码的基础。在汇编语言源程序或汇编语言程序的调试工具中,由于二进制编码冗长而不便于表达,通常将数据表达为八进制、十六进制,而地址则通常被表示为十六进制,因为它们很容易与二进制形式相互转换。十进制数在源程序中也经常使用,但八进制、十六进

制也经常被使用，特别是在调试工具中，往往仅使用十六进制。因此，熟练掌握八进制、十六进制形式，特别是十六进制形式，是熟练掌握汇编语言程序设计的基础。

## 2.3 编码

计算机或其它数字设备中均使用数字信号来表达各种信息，然而这种表达形式并不是任意的，而是遵循一定规则的。一般而言，我们称按照一定规则组织的信号表达形式为编码。这里我们应当注意，数与编码这两个概念有着紧密的联系，但它们是不同的两个概念，数是指理论上我们所定义和使用的数字符号及其运算规则，而编码则是数在机器中的具体表示形式。换言之，编码与数在外观形式和使用规则上都可能出现一定的差别。首先，二者在形式上可能有差别，数是面向人类语言的，是使用人们习惯的符号来表达的，其表达手段很丰富，而在数字设备中的编码只能按照有限信号形式来表达；其次，二者的运算规则也会因形式不同而出现差异。但二者也具有紧密的联系，一般而言，数和编码可以理解为同一事物的两个不同的方面，数是本质，而编码是形式，虽然数与编码的具体表示形式不同，运算规则也不同，但通过良好的编码设计，可以使二者在形式和运算上都严密的对应起来。只有当二者严密对应时，编码才能作为数的表达形式在数字设备中被正确的使用。

在后面的小节中，我们将首先分析无符号数编码，将它作为编码与数无差别的特例加以介绍。然后，我们将分析原码、补码的概念，它们是针对带符号数的两种不同类型编码，在这里，我们可以体会到数与编码的区别，并且，我们能够较的深入理解编码设计的质量对计算机系统结构、程序设计等带来的影响。最后，我们会给出 BCD 码、ASCII 码的概念与相关分析，以加深对编码的认识。而以上这些编码，都是最基本的，也是在汇编语言程序设计中经常使用的，读者应在将来的学习中贯穿数与编码的概念，以便在其它课程中掌握更加复杂、更多种类的编码。在以后的讨论中我们称编码所表达的数为**真值**。并且我们指出，在本教材中，为了突出汇编语言与机器底层中的核心概念，我们仅讨论整数的编码，不讨论实数编码，有关实数编码的理论，请读者参阅计算机组成原理相关的教材。换言之，我们讨论的数集为**整数集**。在汇编语言语法中，我们以**编码末尾带“H”**来表示十六进制编码，以**编码末尾带“B”**来表示二进制编码，以**编码末尾带“O”**来表示八进制编码，末尾不带任何符号或带**“D”**来表示十进制编码。

## 2. 3. 1 无符号数

在这一小节中，我们讨论无符号数及其编码。如果我们仅考虑**正整数集**（包括零）的编码，那么最便捷的编码形式即是**无符号数编码**。从表达形式而言，由于正整数不具备符号，在编码中也不需要符号，因此，无符号数编码与真值的形式是完全一致的，这是编码中较简单也较特殊的情况，也是由于此原因，通常在对无符号数与无符号数编码并没有严格的进行区分。尽管无符号数编码相对简单，但是应当注意几个客观事实，第一，尽管数的进位记数制有多种，但在现今的绝大多数计算机系统中，编码却只能是二进制的；第二，由于受到机器字长的限制，使用编码表达的真值范围一定是有限的，编码所表达的整数集仅仅是整数全集中的一个子集；第三，无符号数编码可以实施加减运算，但在实施减法时，若能使用较小的数作为被减数，将产生负数结果，这是一种超出编码表示范围的现象，同样，实施加法时也可能得到超出表示范围的结果，对于运算结果超出表示范围的情况，在编码运算中称为**溢出**，当然这里是无符号溢出。对于乘除法，同样可能出现运算结果溢出的情况。并且，应当注意，在其它编码运算中也有类似的问题值得注意。下面我们通过示例来理解无符号数的运算。

**例 2.3.1** 假定机器字长为 8 个二进制位，试说明无符号数编码的表达范围，完成无符号数编码 01000010 与 10100001 的加法与减法，并说明有无溢出发生。

由于每个二进制位有 0 和 1 两种取值，8 个二进制位所能表达的编码个数为  $2^8=256$  个。对应最小真值的编码为 00000000，对应最大真值的编码为 11111111，因此，表达范围为 0 到 255。

加法：

$$\begin{array}{r} 01000010 \\ + \\ 10100001 \\ \hline 10100001 \end{array}$$

由于最高位在加法中没有产生进位，运算结果使用 8 位二进制仍能表示，因此没有溢出产生。

减法：



```

  01000010
- 10100001
-----
  10100001

```

由于最高位在减法中产生了强行借位，运算结果是错误的，因为其真正使用的被减数为 101000010，在第 8 位（最低位为第 0 位）比真实的被减数多了一个 1，在计算机中，运算正是这样被执行的。在这个减法中，由于运算结果超出表示范围而产生了错误，有溢出发生。

在汇编语言程序设计中，通常使用无符号数编码表示的信息除了运算中的正整数外，内存单元地址是使用无符号数编码来表示的，各类地址分量中，多数采用无符号数编码表示，但不排除使用带符号数编码的情况。在程序设计中，设计人员清晰的区分无符号数编码与带符号数编码是至关重要的，同时，这对初学者学习相关运算指令也带来了相当的难度。

## 2. 3. 2 原码

在这一小结中，我们将讨论一种带符号数的编码，原码。对于带符号数，首先我们应考虑如何使用二进制数字信号来表达正负符号。在二进制数的表达中，我们可以使用符号“+”和“-”来代表数的正负符号，但在机器中，我们仅有“0”和“1”两种数字信号，没有理论上使用的正负符号。因此，在机器中我们只能使用“0”和“1”来表达编码的正负符号。这也就产生了原码的概念。在原码中，使用“0”表示正号，“1”表示负号，并且固定在编码的最高位。带符号数与原码的相互转换如下例所示。

**例 2.3.2** 假设机器字长为 8 位，将 7 位带符号数 +1000010 与 -1010100 转换为原码

转换过程简单而直接，将正负符号分别转换为“0”和“1”即可，转换过程如下。

```

+1000010 ⇒ 01000010
-1010100 ⇒ 11010010

```

转换的逆过程仅需将原码最高位的“0”和“1”分别替换为“+”和“-”符号即可。

原码的设计是建立在无符号数编码基础之上的，因为我们可以容易的将原码切分为符号部分与数值部分，即最高位为符号，其余低位为数值部分，数值部分与其真值的数值部分没有任何区别。并且，我们应当注意，与无符号数编码类似，受机器字长的限制，原码也是有

表示范围的。若字长为 8 位，最小的编码为 11111111，即-127，最大的编码为 01111111，即+127。因此，原码的表示范围为-127 至+127。我们还应当注意，零的编码在原码中不唯一，它有两个编码，即 00000000 与 10000000，+0 与-0。就运算性质而言，原码并不是一种性质良好的编码，因为其符号位不能参加运算。机器在实施原码运算是，必须先将符号位从编码中分离，并单独进行处理，仅数值部分能够参加运算。那么，这会带来什么问题呢？我们通过下例来分析原码运算带来的问题。

**例 2.3.3** 假设机器字长为 8 位，试完成原码 01010001 与 10001101 的加法与减法运算，并判断溢出情况。

首先，应将原码的符号与数值部分分离：

0	1010001
正	数值
1	0001101
负	数值

(1) 加法：

1) 应先判断两个加数的符号，确定究竟是实施加法还是减法。本例中的加法为“正数+负数”的情况，由于数值部分的运算为无符号数运算，所以实际上应当实施减法，即 1010001-0001101。

2) 实施减法操作前，应当判断被减数与减数的大小关系，在此基础上判断运算结果的符号，并且如果被减数数值部分小于减数，应将二者位置交换，保持较大的数值作为被减数，防止由无符号数运算带来的错误（见例 2.3.1）。在本例中，被减数数值大于减数，不需要交换，且运算结果符号为正。

3) 对数值部分实施加法或减法运算，在本例中为减法运算，并将第 2 步判断的运算结果符号使用“0”或“1”表示后添加在运算结果的最高位，在本例中为“0”。

	1010001
-	
	0001101
	<hr/>
	1000100

添加符号位后，运算结果为 01000100。当被减数大于减数时，无符号运算不会出现溢出的

情况，运算结果正确。

## (2) 减法:

1) 与加法类似，应先判断被减数与减数的符号，确定究竟是实施加法还是减法。在本例中，为“正数-负数”的情况，实际上应当实施加法。

2) 实施加法操作前，不需要判断两个加数的大小关系，但当加法结果超出字长时，加法操作溢出。

$$\begin{array}{r} 1010001 \\ + \quad 0001101 \\ \hline 1011110 \end{array}$$

加法过程中，最高位没有进位，运算结果没有超出字长，因此，没有溢出发生。

对于上例的运算过程，读者还可以举出很多其它的例子。应当注意到，原码的符号与数值是分开处理的，符号位的判断对于运算过程而言必不可少，在计算机系统中，需要专门的逻辑电路对符号位进行分离和判断；就数值部分的运算而言，并不能简单的根据指令所提供的加法或减法属性来执行相应操作，而应当根据符号位判断的结果来执行，而且加法和减法操作都是必要的，在计算机系统中需要提供分离的加法和减法器件来完成各自的运算；对于减法，还需要判断被减数和减数的大小关系，这又增加了额外的器件。因此，原码对于机器操作而言，并不是一种很好的带符号数编码。在下一小节中，我们将介绍带符号数的另一种编码，即补码，相对于原码，它能够在很大程度上减少计算机系统所需要的器件，简化系统设计，我们将深刻体会到这一点。

## 2. 3. 3 补码

在本小节中，我们将讨论另一种带符号数的编码，补码。补码是现代计算机中普遍使用的带符号数编码，也是 8086/8088CPU 所使用的带符号数编码。假设机器字长为  $n$ ，则补码的定义如下。

$$[x]_{\text{补}} = 2^n + x \quad -2^{n-1} + 1 \leq x \leq 2^{n-1} \quad (\text{式 2.3.1})$$

应当注意，在式 2.3.1 中，转换结果必须附加字长限制，对超过字长限制那一位应在转换结果中去掉。按照式 2.3.1 的定义，正数的补码就是其真值本身，而负数的补码则是使用  $2^n$  来

减去其真值的数值部分，因而在形式上，负数的补码与其真值间会相差很大。式 2.3.1 不但给出了补码的定义，而且给出了由数的真值转换得到补码的基本方法，而由补码到真值的逆向转换过程由下式定义。

$$\begin{cases} x = [x]_{\text{补}} & [x]_{\text{补码最高位}} = 0 \\ x = -(2^n - [x]_{\text{补}}) & [x]_{\text{补码最高位}} = 1 \end{cases} \quad -2^{n-1} + 1 \leq x \leq 2^{n-1} \quad (\text{式 2.3.2})$$

从式 2.3.2 定义的逆转换可知，当补码最高位为 0 时，其真值符号为正，而正数的补码等于真值本身；当补码最高位为 1 时，其真值符号为负，其补码的数值部分由  $2^n$  减去补码得到，数值部分添加负号后得到真值。下面我们通过示例来观察正数和负数补码的形式。

**例 2.3.4** 将数+01000011 和-00010110 转换为补码形式，再将补码形式转换为真值形式。（注意，在转换过程中，均附加了字长限制。）

由真值得到补码：

$$[+01000011]_{\text{补}} = 100000000 + 01000011 = 01000011$$

$$[-00010110]_{\text{补}} = 100000000 - 00010110 = 11101010$$

由补码得到真值：

$$[01000011]_{\text{真值}} = +01000011$$

$$[11101010]_{\text{真值}} = -(100000000 - 11101010) = -00010110$$

下面我们来分析一下补码的加法和减法运算性质。首先，补码的加法和减法运算是否闭合？即两个数的补码被执行加或减操作后，得到的补码是否为正确的“和”或“差”补码？我们通过下面的示例来加以说明。

**例 2.3.5** 将数+01000010 与数-00110110 转换为补码，并完成加法和减法运算，并将运算结果转换为真值，以分析运算的闭合性。

(1) 将所给二进制数转换为补码：

$$[+01000010]_{\text{补}} = 01000010$$

$$[-00110110]_{\text{补}} = 100000000 - 00110110 = 11001010$$

(2) 补码的加法:

$$\begin{array}{r}
 01000010 \\
 + 11001010 \\
 \hline
 00001100
 \end{array}$$

真值的加法:

$$\begin{array}{r}
 01000010 \\
 - 00110110 \\
 \hline
 00001100
 \end{array}$$

补码运算结果转换为真值:  $[00001100]_{\text{真值}} = +00001100$ 

与真值运算结果比较, 可知二者是一致的。

(3) 补码的减法:

$$\begin{array}{r}
 01000010 \\
 - 11001010 \\
 \hline
 01111000
 \end{array}$$

真值的减法:

$$\begin{array}{r}
 01000010 \\
 + 00110110 \\
 \hline
 01111000
 \end{array}$$

补码运算结果转换为真值:  $[01111000]_{\text{真值}} = +01111000$ 

与真值运算结果比较, 可知二者是一致的。

由上例的分析可知 (这里不给出严格证明), 补码的加减法能够与真值的加减法对应起来, 满足**运算闭合性**, 即“**补码的和与差**”等价于“**和与差的补码**”。换言之, 使用补码加减法能够在计算机系统中完成带符号数的加减法。同时, 在例 2.3.5 中, 我们可以体会到, 虽然补码的符号位是编码的最高位, 但却与原码的符号位不同, **补码的符号位与数值是一体的**, 不能截然分开, 在运算中, 补码的符号位同样要参加运算。其根本原因在于补码运算实质上是一种带符号运算, 而原码运算实质上却是一种附加符号判断的无符号运算。因此, 由于这个根本区别, 我们对补码运算可以按照指令中指定的加法或减法操作直接执行运算, 而不需要预先判断两个操作数符号位间的关系。相对于原码, **符号位分离与判断的逻辑电路可以在使用补码的计算机系统中删去**, 简化了系统设计。并且, 由于补码的运算结果是带符号的, 因此在实行减法运算前, 不需要判断被减数与减数的大小关系, 在原码计算机中需要的、**用于实现被减数与减数关系判断的逻辑电路在补码计算机中也不再需要了**, 这又进一步简化了系统设计。

下面我们将给出数与补码间转换的简便方法, 此方法非常重要, 它不既可以简化数与补码的转换, 而且能够用于实现相反数补码间的相互转换, 在此基础上, 还能够帮助我们清晰分析现代计算机使用补码加法来实现补码减法的基本原理。从式 2.3.1 我们知道了数与补码

相互转换的基本方法，那么有没有更简便、更利于机器实现的转换方法呢？答案是肯定的，正数与自己的补码相等，相互间不需要转换，负数转换为补码或负数补码转换为负数真值，都可以通过“变反加 1”的方法来实现。我们通过示例来说明这种简便的转换方法。

**例 2.3.6** 通过变反加 1 法将负数-00110001 转换为补码，并使用该方法实现逆转换。

(1) 负数转换为补码：

数值部分变反加 1： $[-00110001]_{\text{补}} = 11001110 + 1 = 11001111$

基本转换法： $[-00110001]_{\text{补}} = 100000000 - 00110001 = 11001111$

比较可知，变反加 1 法与基本转换法所得结果一致。

(2) 负数补码转换为真值：

补码变反加 1 并添加负号： $[11001111]_{\text{真值}} = -(00110000 + 1) = -00110001$

基本转换法： $[11001111]_{\text{真值}} = -(100000000 - 11001111) = -00110001$

比较可知，变反加 1 法与基本转换法所得结果一致。

“变反加 1”不仅能够运用在负数与其补码间的相互转换中，并且也能用在相反数补码的相互转换中，或已知一正数补码或负数补码，我们能够将其变反加 1 后得到其相反数的补码，这一点对于在计算机系统中使用补码加法来实现补码减法是至关重要的。为了清晰的说明这一性质，并突出其相对于原码的优越性，我们将深入进行分析。从直观感觉而言，补码是一种正数编码与负数编码相互“耦合”或“互补”的编码，正数编码与负数编码的关系正像齿轮的凸出部与凹陷部一样，如果是相反数补码相加，会相互抵消。下面我们通过示例来观察这种现象。

**例 2.3.7** 分析数+00110101 即其相反数的补码的加法过程，说明正负数补码相互抵消的原理。

由正数求补码（补码与真值相等）： $[+00110101]_{\text{补}} = 00110101$

由正数补码求相反数补码，变反加 1 可得：11001011

两个相反数补码相加：

$$\begin{array}{r}
 00110101 \\
 + \quad 11001011 \\
 \hline
 1 \ 00000000
 \end{array}$$

附加 8 位的字长限制后，最高位进位丢失，运算结果为 0 的补码。我们可以将负数补码拆为变反加 1 的形式后分析此次加法，能够更清晰的理解相反数补码相加的原理：

$$\begin{aligned}
 00110101 + 11001011 &= 00110101 + (11001010 + 1) = \\
 (00110101 + 11001010) + 1 &= 11111111 + 1 = 100000000
 \end{aligned}$$

由于相反数补码间是变反加 1 的关系，因此，相加后必定结果为  $2^n$ ，附加字长限制后结果为 0 的补码。

通过上例，我们可以体会到正数与负数补码间的抵消或耦合关系。并且，由于变反加 1 法能够方便的将一个补码转换为其相反数的补码，那么两个补码减法就可以通过减法来实现。其原理可以由下式说明：

$$[x]_{\text{补}} - [y]_{\text{补}} = [x]_{\text{补}} + [-y]_{\text{补}} \quad (\text{式 2.3.3})$$

上式可以通过补码运算的闭合性容易的得到证明，这里我们不给出证明过程，读者可以通过实例加以验证。补码的这一性质使得使用加法运算实现减法运算成为可能，换言之，如果要执行两个补码的减法运算，那么只需要预先使用变反加 1 法将减数补码变为其相反数的补码，再执行加法运算即可。因此，在补码计算机中，仅需要加法器就可以完成带符号数的加法和减法两种运算，而不像原码计算机那样，既需要加法器，也需要减法器，更加简化了系统设计。

最后，我们讨论补码的表达范围以及补码运算的溢出问题。与无符号数编码、原码相似，在机器字长的限制下，补码也有表达范围，如果运算结果超出这个范围，也会产生溢出现象。在式 2.3.1 给出的补码定义中，已经给出了补码的表示范围。应当注意，补码的表达范围中，最小的负数补码与最大的正数补码并不对称，就 8 位编码而言，最小的负数补码为 10000000，即-128，最大的正数补码为 01111111，即+127（读者可以自己换算并验证）。为什么会出现这种不对称现象？这是因为在补码表示中，零的补码只有一个，即 00000000，而不像原码，有+0 和-0 两种针对零的编码。当补码的运算结果超出补码表示范围时，运算溢出，在机器字长的限制下，会得到错误的运算结果。因此，在计算机系统中，必须具有补码运算的溢出判别机制，而直接判断运算结果是否超过字长限制是比较困难和复杂的。那么，除了判断运

算结果是否超出范围外,有没有更简便、更利于机器实现的溢出判断方法呢?回答是肯定的,我们在这里给出判断补码加减运算是否溢出的简便方法。下面给出一个重要的等价性质:**补码的加减运算结果是否具有正确的符号位与补码运算是否溢出这两个命题是等价的**,即如果补码加减运算后得到错误的符号位,那么运算结果一定是溢出的;如果补码加减运算后得到正确的符号位,那么运算结果一定没有溢出。使用符号位判断补码运算是否溢出的方法我们简称为符号判断法。判断规则可以归纳如下:

- 1) “正数补码+负数补码”:这种情况是正数与负数抵消的过程,不会出现溢出。
- 2) “正数补码+正数补码”:若运算结果的符号为正,则没有溢出,反之,有溢出。
- 3) “负数补码+负数补码”:若运算结果的符号为负,则没有溢出,反之,有溢出。

补码加减运算中所能遇到的各种情形,都可以等价为以上三种。显然,这种判断方法非常简便。在这里,我们并不给出复杂的证明,而是通过示例来说明这一等价性质的正确性。

**例 2.3.8** 完成数+01000010 和数+01100001 的补码加法和补码减法,并根据符号判断法、补码定义分别判断运算是否溢出。

首先将两个给定的数转换为补码:

$$[+01000010]_{\text{补}} = 01000010$$

$$[+01100001]_{\text{补}} = 01100001$$

(1) 补码加法:

$$\begin{array}{r} 01000010 \\ + \\ 01100001 \\ \hline 10100011 \end{array}$$

符号判断法判断运算结果:因为,运算结果最高位为1,因此运算结果可解释为“正数补码+正数补码=负数补码”,显然运算结果中的符号位出错,有溢出发生。

按照补码定义判断运算结果:直接使用真值计算来判断。

$$\begin{array}{r} +01000010 \\ + \\ +01100001 \\ \hline +10100011 \end{array}$$

将真值运算结果转换为十进制为+163,而8位补码能够表达的最大正数为 $+2^{8-1}=+128$ ,显然



运算结果超出了补码能够表达的范围，有溢出发生。

## (2) 补码减法

$$\begin{array}{r} 01000010 \\ - 01100001 \\ \hline 11100001 \end{array}$$

符号判断法判断运算结果：所实现的补码运算为“正数补码-正数补码=正数补码+负数补码”，

这种运算是正数与负数的抵消过程，不可能导致溢出，运算结果为正或为负都是正确的。

按照补码定义判断运算结果：直接使用真值计算来判断。

$$\begin{array}{r} +01100001 \\ - +01000010 \\ \hline +00011111 \end{array}$$

上式中为保证真值计算过程的正确，使用了较大的正数作为被减数，因此运算结果应该为负数，为**-00011111**，换算为十进制为-31，运算结果在 8 位补码表示范围内，没有超出，因此没有溢出发生。

在上例中，我们可以看到使用补码定义和符号判断法同样都可以判断补码运算是否溢出，但使用符号判断法会简便很多。事实上，计算机硬件系统中使用的补码加减法溢出判断规则，正是符号判断法。

## 2. 3. 4 ASCII 码

在计算机系统中，除了参与各类数值运算的数需要编码外，其它非数值信息也需要编码。例如字符，就人类而言，字符是同特定的图案和发音联系在一起的，在计算机应用中，人们更加侧重于字符的图案形式，即字符模板。图 2.3.1 给出了字符模板的例子，在图中我们可以看到，字符形状是以像素为基本单位构成的。在屏幕上显示某一字符的形状时，使用字符模板是非常适合的。但是，我们在屏幕上显示某个字符形状时，如何找到相应的字符模板呢？要完成指定字符在屏幕上的显示，这就需要使用字符编码的概念，从本质而言，字符编码是字符模板的编号。当我们希望在屏幕上显示某个字符模板时，通过字符编码在内存中定位到对应字符模板的位置，再将字符模板送到显存中，最终显示在屏幕上。如果我们希望对文本文件中的字符序列进行处理，使用字符编码也是很恰当的，在现代字处理软件中，进行各种

字符处理时所使用的都是字符编码，而仅当显示字符时才使用字符模板。

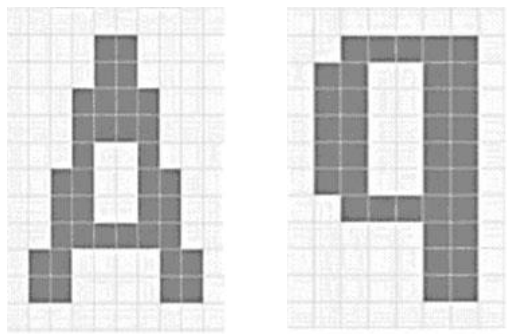


图 2.3.1 字符模板示例

ASCII 码是字符编码中的一个典型案例，其全称为 American Standard Code for Information Interchange，在微型计算机中，该字符编码已成为国际化标准。表 2.3.1 中给出了 ASCII 码与字符模板的对应关系。在表中我们可以发现，英文字母和数字的编码排列都是连续的，这种编码方式对于字符处理相当方便。表 2.3.1 中 ASCII 编码为十六进制形式，我们以末尾带“H”的数字来表示十六进制数。大写字符编码从 41H 至 5AH，小写字符编码从 61H 至 7AH，因此要完成字符的大小写转换只需要加或减一个常量即可。而数字字符的编码从 30H 至 39H，只需屏蔽编码的高位，低位即是数字字符对应的数值，对于数值与字符的转换是很方便的。

	0-	1-	2-	3-	4-	5-	6-	7-
-0	NUL	DLE	SP	0	@	P	`	p
-1	SOH	DC1	!	1	A	Q	a	q
-2	STX	DC2	"	2	B	R	b	r
-3	ETX	DC3	#	3	C	S	c	s
-4	EOT	DC4	\$	4	D	T	d	t
-5	ENQ	NAK	%	5	E	U	e	u
-6	ACK	SYN	&	6	F	V	f	v
-7	BEL	ETB	'	7	G	W	g	w
-8	BS	CAN	{	8	H	X	h	x
-9	HT	EM	}	9	I	Y	i	y
-A	LF	SUB	+	:	J	Z	j	z
-B	VT	ESC	*	;	K	[	k	{
-C	FF	FS	,	<	L	\	l	
-D	CR	GS	-	=	M	]	m	}
-E	SO	RS	.	>	N	^	n	~
-F	SI	US	/	?	O	_	o	DEL

表 2.3.1 ASCII 码表（十六进制编码，列为高位 ASCII 码，行为低位 ASCII 码）

ASCII 码是一种设计良好的字符编码，它能够支持方便的字符大小写转换、数字字符与数值的转换。汇编语言程序设计中有字符串处理的方法，我们将在非数值处理程序设计一章详细介绍。

## 2. 3. 5 BCD 码

我们都知道，前面所介绍的无符号数编码、原码、补码等，都是针对二进制数的二进制编码，那么有没有针对十进制数的二进制编码呢？我们在这一小节中，我们将介绍一种这样的编码，即 BCD 码。BCD 的全称为 Binary-Code Decimal，意即“采用二进制编码的十进制数”。在 2.2 节中，我们讨论了不同进位记数制间的转换，并且我们都已经理解 1 个十六进制位正好对应着 4 个连续的二进制位，1 个八进制位正好对应着 3 个连续的二进制位，但是 1 个十进制位却不能与整数个连续的二进制位相对应。3 个连续的二进制位表示的数值范围为 0~7，4 个连续的二进制位表示的数值范围为 0~F，这里的 F 是十六进制记法，等于十进制中的 15。因此，我们只能使用 4 个连续的二进制位来表达 1 个十进制数位，但是我们必须限定这个 4 位位组的取值范围只能在 0~9 之间，有了这样一个限制后，数位的对应关系才能保持正确。

十进制	4 位二进制位组 (BCD 码)
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

表 2.3.2 十进制系数与二进制位组 (BCD 编码) 间的对应关系

使用 4 位二进制位组，在限定取值范围为 0~9 之后表达的十进制数，称为 BCD 码。与上一小节介绍的 ASCII 码比较，可以发现数字字符的 ASCII 码与 BCD 码间具有良好的转换关系。BCD 码的设计目的正是针对从键盘输入获取数字字符串后，使用二进制运算指令直接针对字符编码进行十进制运算。由于 BCD 码运算得到的运算结果也是十进制的数字字符，因此可以方便的用于输出显示。表 2.3.2 列出了十进制系数与二进制位组取值间的关系。虽

然 BCD 码能够表达十进制系数，但是它却避免不了 4 个连续二进制位在二进制加减运算中产生的进、借位均为十六进制进、借位这一本质，也避免不了 4 个连续二进制位在二进制乘法中等于 1 个十六进制位这一本质，因此保证使用 BCD 码的正确性，还必须借助运算调整机制。我们将在数值运算程序设计一章详细讨论相关的内容。

内部资料，不得外传！

## 习题 2

1. 请将下列十进制数转换为二进制数，并统一使用 8 位二进制数表达。

- (1) 9            (2) 25            (3) 126            (4) 207            (5) 100  
(6) 10            (7) 16            (8) 92            (9) 63            (10) 255

2. 请将下列二进制数转换为十进制数。

- (1)  $(00000001)_2$             (2)  $(01000010)_2$             (3)  $(10010011)_2$   
(4)  $(01010101)_2$             (5)  $(01100100)_2$             (6)  $(10000000)_2$   
(7)  $(11100111)_2$             (8)  $(00000011)_2$             (9)  $(01111111)_2$

3. 请将下列八进制数或十六进制数转换为 8 位二进制数。

- (1)  $(137)_8$             (2)  $(062)_8$             (3)  $(005)_8$   
(4)  $(5A)_{16}$             (5)  $(39)_{16}$             (6)  $(1F)_{16}$

4. 请将下列二进制数分别转换为八进制和十六进制数

- (1)  $(01011010)_2$             (2)  $(01100001)_2$             (3)  $(01110111)_2$

5. 请将下列二进制数转换为 8 位补码。

- (1)  $(+00110001)_2$             (2)  $(-00010011)_2$             (3)  $(-01010100)_2$   
(4)  $(+00000000)_2$             (5)  $(-01110111)_2$             (6)  $(-01100101)_2$

6. 请将下列 8 位补码转换为二进制数。

- (1)  $[10010001]_{\text{补}}$             (2)  $[00110010]_{\text{补}}$             (3)  $[11100110]_{\text{补}}$   
(4)  $[11111111]_{\text{补}}$             (5)  $[11111001]_{\text{补}}$             (6)  $[11000000]_{\text{补}}$

7. 请将下列十进制数转换为 8 位二进制数，作为 8 位无符号数编码完成相应运算，给出运算结果，并且判断运算的有无溢出。

- (1)  $56+32$             (2)  $100+75$             (3)  $123+8$   
(4)  $125+134$             (5)  $200+105$             (5)  $26+34$

8. 请将下列十进制数转换为 8 位二进制补码，完成相应运算，给出运算结果，并且判断运算有无溢出。

- (1)  $-123+66$             (2)  $62-54$             (3)  $130+140$   
(4)  $-175-90$             (5)  $-78-9$             (6)  $66+75$

9. 请简述 ASCII 码的概念，并说明该编码中的优越性。

10. 请简述 BCD 码的概念，并说明该编码的主要用途。

## 第3章 计算机系统模型

### 3.1 计算机系统的基本结构

在本章中，我们将给读者搭建一个突出计算机系统主要功能的、易于理解的、简化的计算机系统模型。该模型不对应任何具体的计算机系统，但它却具备所有计算机系统所应具备的基本特征。对初学者而言，掌握这样一个基本特征全面、结构简单的计算机系统模型，能够达到快速理解计算机底层工作原理，并在短时间内建立硬件知识体系基本框架的目的，为学习汇编语言和相关的后续课程做好必要的准备。在本小节中，我们将讨论计算机系统的基本结构，这种基本结构是所有计算机系统所共有的，它与计算机系统的具体类型无关。

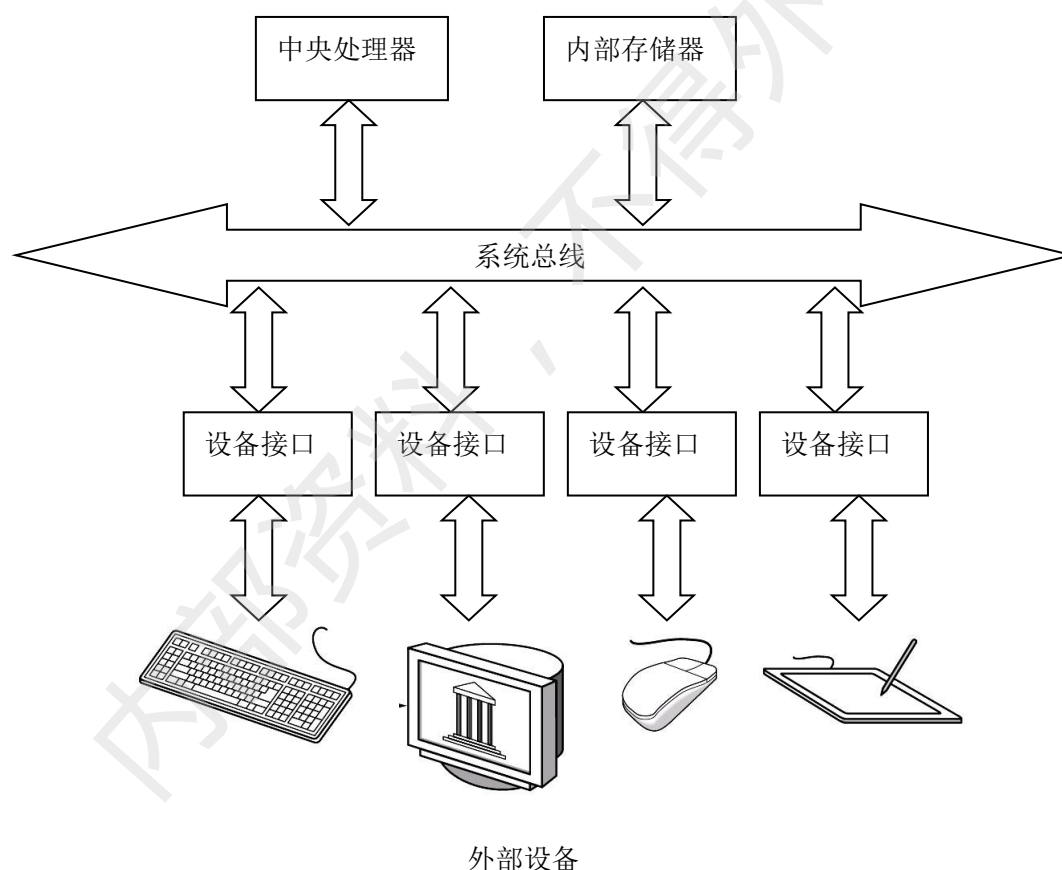


图 3.1.1 计算机系统基本结构

虽然计算机系统种类繁多，包括体积庞大而且性能非常强的巨型计算机，也包括我们日常生活中使用的手机、计算器，它们虽然功能简单、体积微小，但仍然符合计算机系统的结构特征。无论机器在体积、性能、功能上有何种差异，它们的基本结构却是一致的，都包括五种

功能部件。图 3.1.1 描述了由五种功能部件构成的计算机系统结构。如图 3.1.1 中所示，计算机系统的基本结构由中央处理器、内部存储器、系统总线、设备接口、外部设备五种功能部件构成。在计算机系统结构这一学科范畴内，主要讨论的是前四种功能部件，但对于接口设计技术而言，要求设计人员对外部设备的工作原理具备一定程度的理解，否则无法完成接口电路的设计。下面，我们将逐一介绍这五种功能部件在计算机系统中的作用，并且说明各部件与其它部件间的关系。

### 3.1.1 中央处理器

中央处理器的英文全称为 **Center Process Unit**，简称 **CPU**，是计算机系统中最核心的部件，如果与人体结构对比，它的功能接近于人体中的大脑。下面我们介绍中央处理器的主要功能。

#### (1) 读取、解释、执行机器指令

通常情况下，CPU 不断的通过系统总线从内部存储器中逐条读取出机器指令；在 CPU 内部对机器指令进行译码，即解释指令；在指令译码后，CPU 按照时间顺序产生一系列控制信号，控制计算机系统中其它功能部件按照指令要求完成操作，从而完成指令的功能。由多条指令组成的指令序列即为程序，因而程序的运转是依靠 CPU 作为硬件平台来实现的。我们在操作计算机时可能会有这样的感受，运行操作系统或某个应用程序后，我们也许在一定时间内什么操作都不做，那么 CPU 的指令执行功能在这种时候是否处于停滞状态呢？事实上，只要我们没有使计算机系统处于停机状态（或其它休眠状态），那么 CPU 始终处于不断读取指令、解释指令、执行指令这一周而复始的过程中，当我们没有通过输入设备进行操作时，操作系统与应用程序或许正在利用空闲时间处理其它事务，或者正处于循环等待状态，而循环等待也是由 CPU 执行指令序列来实现的。因为 CPU 具备执行指令和其它时序过程的功能，它成为**计算机系统的控制中心**。

#### (2) 与内部存储器、设备接口交换信息

如果要完成各类数据处理，CPU 通常通过总线从内部存储器读取原始操作数，完成处理或运算后，将结果数据写入内部存储器保存；如果要控制某个外部设备，那么 CPU 通常通过总线从设备接口读取外部设备状态，通过对状态进行分析后，再将控制信息写入设备接

口，以达到根据具体情况实施控制的目的。换言之，数据的处理、设备的控制都是 CPU 通过执行指令序列来完成的，因此它们也是 CPU 的主要功能之一。

### **(3) 执行除指令外的其它时序过程**

计算机系统的运转过程实际上是一种时序过程。时序过程则是指按照既定的时间顺序、分步骤完成操作的动态过程。指令的执行属于一种时序过程，但并不包括所有的时序过程。例如计算机系统中常见的中断请求、中断响应过程、DMA 过程等，都是时序过程，但它们并不是依靠执行指令来实现的。而 CPU 也会处理这样一些时序过程，并且，这些时序过程的重要性并不亚于指令的执行。举例来说，除非 CPU 正在执行循环程序等待键盘输入，通常情况下 CPU 对键盘输入的接收方式为中断处理方式，我们进行键盘输入时，就是在打断 CPU 执行当前程序，使之暂停当前程序的执行，转而接收键盘输入，那么这一打断当前程序执行的过程就是一种非指令执行的时序过程，即中断请求、响应过程。有关时序过程的基本概念和原理，我们将在本章后半部分讨论计算机系统工作原理时详细介绍。

### **(4) 完成算术或逻辑运算**

CPU 内部具有运算器，能够完成各类算术或逻辑运算，实际上除了复杂的设备接口中可能完成部分运算外，计算机系统中几乎所有与运算相关的操作都是在 CPU 内部通过执行各种运算指令来完成的。通常的运算包括加、减、乘、除四则运算，还包括与、或、非、异或等逻辑运算。由于这些运算都是在 CPU 内完成的，因此 CPU 是整个计算机系统的运算中心。

### **(5) 提供少量的存储单元**

按照不同功能部件的分工，多数的原始数据和运算结果保存在内部存储器中，CPU 仅为运算提供短暂的、临时的数据存储，运算结束后，原始数据没有必要继续在 CPU 中存在，而运算结果也会被保存到内部存储器。这种位于 CPU 内部，为运算数据提供短暂存储功能的存储单元称为寄存器。为了在满足各类运算功能的前提下，同时降低 CPU 内部结构的复杂程度，各个寄存器的功能都会有所不同。在汇编语言程序中，读者会发现多数指令都是针对寄存器进行操作的，因此对寄存器功能的必要记忆成为学习汇编语言程序设计的第一个难点，我们会在后续内容中逐步体会到这一点。



### 3.1.2 内部存储器

内部存储器提供大量的存储单元，存放当前计算机系统所使用的绝大部分软件资源，包括构成程序的指令序列、用于运算的各种数据等。因此，我们可以称内部存储器为**计算机系统的存储中心**。我们应当区分内部存储器与外部存储器的概念，内部存储器对应于微机结构中主板上的内存条，而外部存储器则对应于硬盘驱动器、光盘驱动器这样一些外部设备。换言之，外部存储器除了具备庞大的存储容量外，它们在概念上仍然属于图 3.1.1 中的外部设备，与键盘、鼠标、显示器这些外部设备并没有本质上的区别。而内部存储器则不同，它不属于外部设备，是五种功能部件中独立的一种，这是因为**内部存储器中所存放的信息都是 CPU 立即可以使用的**，并且，CPU 对它的访问方式是直接而特殊的，CPU 访问内存与外部设备时使用的控制信号也是不同的。我们可以通过其中一个现象来认识这一点，任何可执行的程序文件存放在硬盘上是不会得到 CPU 运行的，必须先将它装载到内存中之后，CPU 才能逐条读取、解释、执行其中的指令，这是由 CPU 执行指令的时序机限制定的。并且，由于在概念上，内部存储器的读写速度高于外部存储器，因而在系统结构上它的存在也是必要的。因此，无论在何种计算机系统内部，都有内部存储器这一概念。在以后的讨论中，我们简称内部存储器为内存。

### 3.1.3 系统总线

系统总线实质上是由几组连接线路构成的，用于连接计算机系统中其它功能部件，使它们之间能够相互通信。从图 3.1.1 中，我们可以看到，系统总线是连接中央处理器、内部存储器、设备接口的枢纽，这些功能部件如果要相互通信，则必须通过系统总线。因此，我们可以称系统总线为计算机系统的**信息交换枢纽**。如果与人体神经系统对比，它接近于中枢神经系统，但这仅仅是类比，并不是绝对化的对应。系统总线可以分为三组，即**地址总线、数据总线、控制总线**。两个功能部件要完成信息交换，这三组总线缺一不可。举例来说，CPU 要读取内部存储器中某一内存单元中的数据，那么它必须先在地总线上提供内存单元的地址，以确定要访问的是众多内存单元中的哪一个；并且需要在控制总线上提供“读”控制信号，以使内部存储器能够区分当前进行的操作是“读”还是“写”；当地总线和控制总线上所需的信号稳定后，内存单元才会执行读操作，将指定内存单元的数据读取出来，并传递到数据总线上，而 CPU 则通过数据总线取得所读取的数据。这里简要总结一下，地址总线

上提供的信号为地址信号,用于区分待操作的存储单元,控制总线上提供的信号为控制信号,用于标识当前的操作类型,数据总线上则是读取出来的或待写入的数据信号。

### 3.1.4 设备接口

设备接口是连接系统总线与外部设备的桥梁,CPU 要达到控制外部设备或与外部设备交换信息的目的,那么必须通过设备接口间接实现。从图 3.1.1 可以看到,任何外部设备都是通过设备接口连接在总线上的。例如,显示器必须通过显示接口卡才能连接在总线上,键盘必须通过键盘接口电路才能连接在总线上。那么,为什么外部设备一定要通过接口才能连接在总线上呢?换言之,设备接口在计算机系统结构中存在的必要性是什么呢?这主要有两个原因,这也是设备接口最基本的两个功能,下面分别来介绍这两个基本功能。

#### (1) 信号转换

不同用途的外部设备,往往使用不同的信号。例如,作为输入设备,话筒提供的是由声音转变而来的模拟信号,而作为输出设备,音箱能够接收的也是模拟信号,它能将模拟电信号重新转变为连续的声音。对于模拟信号,由于它在时间上、取值上都是连续的,计算机系统无法接收和处理,因为计算机系统所使用的是数字信号。那么,对于模拟的输入设备,它必须通过设备接口进行信号模/数转换,将输入的模拟信号转换为数字信号,并且作为数值序列保存在接口的存储单元中,CPU 才能通过系统总线来读取数值序列;对于模拟的输出设备,它仅能接收模拟信号,但 CPU 提供的却是二进制的数值序列,因此这些由 CPU 提供的数值必须经过设备接口经过数/模转换为模拟信号,然后才能发送到模拟输出设备。

即便外部设备所使用的是数字信号,但其定义也不完全一致,并且与系统总线上的信号标准也不一致。因为,首先数字设备中使用的进位记数制不一致,采用二进制、三进制、四进制、八进制信号都有可能,而计算机系统仅使用二进制信号;即便针对使用二进制信号的数字设备,其“0”和“1”信号的电平定义都可能与总线上的定义不一致。因此,对于数字设备,仍需设备接口完成不同数字信号间的转换。

#### (2) 数据缓冲

由于用途不同,各类外部设备的也处理速度各不相同,甚至与 CPU 处理速度相比时可能出现很大的差异。就输出设备而言,例如打印机,它打印字符的速度相对于 CPU 发送字

符的速度而言是很慢的，属于低速设备；就输入设备而言，例如摄像机，它在 1 秒钟内能够获取 30 帧左右的图像，其数据产生速度高于 CPU 接收数据的速度。那么，对于慢速的输出设备，如果 CPU 不顾及其速度因素，而将大量输出数据一连串的发送给外部设备，那么必然造成部分数据丢失的情况；同样的，对于高速的输入设备，如果它不考虑 CPU 的低速因素，在通过总线向 CPU 传递输入数据时就可能造成部分数据丢失。为了协调外部设备与 CPU 间较大的速度差异，设备接口中可以提供一定容量的存储器件，以缓冲高速一方所发送的数据，等待低速一方有时间接收新数据时再发送出去。

以上讨论的信号转换与数据缓冲这两个功能既是设备接口的基本功能，也是其在计算机系统结构中存在的必要性所在。通过上面的说明，我们也能体会到外部设备接口是具有两面性的，一方面是与外部设备接口，一方面是与系统总线接口。例如，显示接口卡一方面需要通过信号线与显示器连接，另一方面需要通过总线插槽与系统总线连接。

### (3) 与 CPU 交换信息

既然设备接口具有它存在的必要性，那么就有必要将它设计为易于 CPU 控制和访问的形式。换言之，设备接口需要为 CPU 控制外部设备、与外部设备交换信息提供硬件基础。而这些操作都是以设备接口中具有特定含义的存储单元为基础的，这些位于接口中的存储单元，我们称之为端口。如果与人的神经系统作比较，端口在计算机系统中的地位接近于神经系统中的神经末梢。无论是什么样的设备接口，设备接口中的端口都可以分为三类，即**状态端口**、**命令端口**、**数据端口**。状态端口中的二进制状态信息由设备接口根据外部设备的当前状态随时更新，CPU 则可通过执行读端口指令读取状态端口，并查询这些信息。例如，输入设备是否有新的输入数据等待 CPU 读取，输出设备是否准备好接收 CPU 发送的下一个输出数据，设备运转有无故障等。命令端口中的二进制命令信息是由 CPU 通过系统总线写入的，由设备接口对其进行解释并按照命令对外部设备实施必要的控制。例如，完成外部设备的初始化、改变外部设备的运转方式等。而数据端口则是 CPU 与外部设备进行数据交换的硬件基础，CPU 向输出设备发送的输出数据，必然先保存到数据输出端口，然后经过设备接口转换后提供给输出设备；输入设备向 CPU 发送的输入数据，经过设备接口转换为二进制数据后，必然先保存到数据输入端口，等待 CPU 来读取。

正是由于端口的存在，使得 CPU 能够控制设备接口，并与设备接口通信。值得注意的是，CPU 对外部设备的控制，只能深入到端口这一级别，更加细致的控制是依靠设备接口间接实现的。

#### (4) 可编程

由于 CPU 对接口的控制是由 CPU 对状态端口、命令端口、数据端口的综合使用来完成的,因此通常需要一段指令序列来完成控制,这就是接口控制程序,它具有自己的逻辑结构。在现代计算机系统中,接口控制程序通常被组织为中断服务程序的形式,它们可能被硬件中断请求触发并执行,也可能被应用程序直接以系统调用的方式来执行。这种以端口为硬件基础提供的可编程控制功能是几乎所有设备接口都具有的。通常,生产设备接口的厂商,都会提供相应的驱动程序,而这些驱动程序正是由 CPU 执行,并用于接口控制的。汇编语言是一种描述硬件操作的语言,因此掌握汇编语言对于深入理解各类设备接口的控制机制是非常必要的。设备接口是微机原理与接口技术中讨论的重点,也是汇编语言主要针对的操作对象,但本教材中只会涉及到接口的基本知识,重点讨论的是指令系统本身,读者应在深刻理解指令系统的同时对接口控制加以初步的体会。

### 3.1.5 外部设备

计算机系统的外部设备种类繁多、功能各异,因此在它们的具体设计上差异也相当大,但它们都是计算机与外部世界交互的必要手段。例如,用于人机交互的设备包括我们熟悉的显示器、打印机、键盘、鼠标等;用于计算机系统与自然界交互的设备包括摄像机、温度传感器、压力传感器、机械手臂等。外部设备功能越强,计算机系统与外部世界交互的能力也就越强,但这也必然导致外部设备的设计变得复杂,同时导致相应的设备接口设计复杂。要使计算机系统能够与结构复杂的外部设备交互,必须设计良好的设备接口。接口设计技术不但要求熟悉计算机系统的总线标准,而且还要求对外部设备本身的工作机制、信号定义有一定的了解,因为任何设备接口都既面向总线连接,也面向设备连接。外部设备不属于本教材讨论的主要范畴,但读者应查阅相关资料,了解常规外部设备的工作原理,这对后续的微机原理相关课程是有益的。

## 3. 2 计算机系统存储单元

计算机系统存储单元在 3.1 小节中已经有了分别的介绍,在这里我们归纳一下。无论是何种类型的计算机系统,系统中的存储单元都可以分为三类,即寄存器、内存单元、端口。位于 CPU 内部的存储单元通常称为寄存器;位于内部存储器中的存储单元通常称为内存单元;位于设备接口中的存储单元通常称为端口。但是也存在例外的情况,端口通常是特

指设备接口中有特殊用途的存储单元，如果设备接口中存在大容量的存储器，通常不被称为端口。例如显示接口卡中的显示存储器，通常就不被称为端口，而用于控制显示接口卡工作模式的那些专用存储单元，则被称为端口。并且，在不同的教材中，上面提到的这些存储单元名称可能出现不一致的情况，应根据实际情况加以理解。

我们在本小节集中讨论计算机系统存储单元，是由于包括指令执行在内的所有时序操作，除了在 CPU 内完成各类算术运算、逻辑运算外，大多数的操作就在于各存储单元间的数据交换。如果对计算机系统的时序操作进行一个宏观观测，我们可以在计算机系统模型上发现，多数时候总线上都在实行三种存储单元间的数据交换。寄存器与寄存器间的数据交换是在 CPU 内部完成的，不需要通过数据总线，如图 3.2.1 所示。图中所示为两个寄存器交换数据的情况，通常情况下，CPU 读取其中一个寄存器中的数据，并写入另一个寄存器中，被实行读操作的寄存器通常称为源寄存器，而被实行写操作的寄存器通常称为目的寄存器。

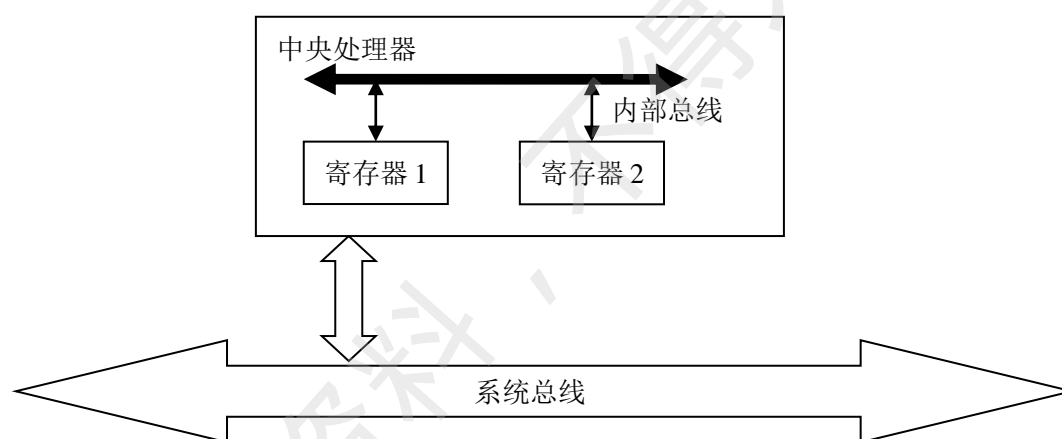


图 3.2.1 寄存器间的数据交换

CPU 会给源寄存器施加一个读控制信号，并给目的寄存器施加一个写控制信号，分别用以完成读写操作，这一控制方式对于任意两个实行数据交换的存储单元都适用，只不过施加的读写信号性质可能不同。在 8086/8088 系统中，对寄存器、内存单元、端口施加的读/写信号都是不同性质的，用以区别不同类型的存储单元。有时读信号与写信号会施加在同一存储单元上，例如我们将某个寄存器中的数据读取出来，执行逻辑非运算后，再写回该寄存器。读者应在后续学习过程中，以指令系统为基础，借助系统模型来体会类似的原理。寄存器与内存单元间交换数据，或寄存器与端口间交换数据，一定是通过系统总线完成数据传输的。寄存器与内存单元间的数据交换如图 3.2.2 所示，寄存器与端口间的数据交换原理与此相似。

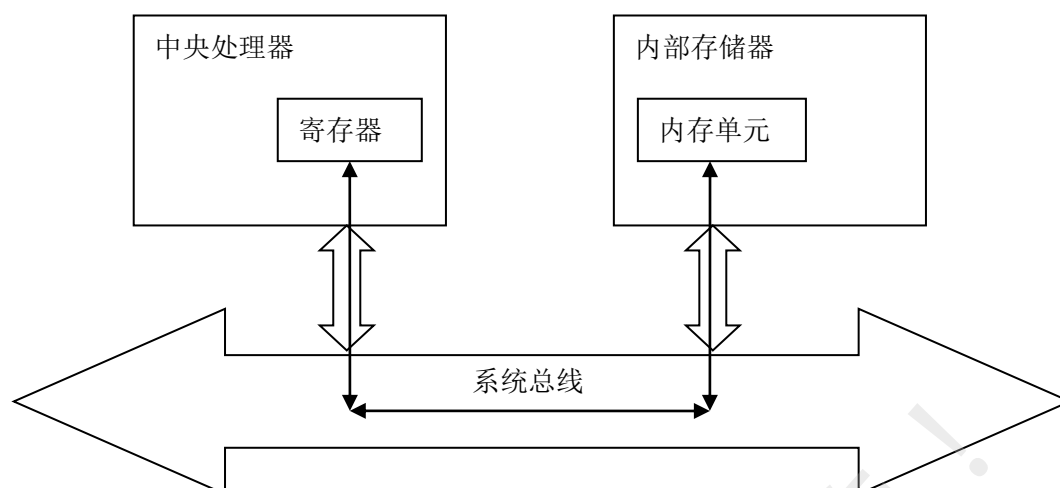


图 3.2.2 寄存器与内存单元间的数据交换

以上介绍的寄存器之间的数据交换，寄存器与内存单元、端口间的数据交换，都是可以通过 CPU 执行指令来完成的。除此之外，还有内存单元之间、端口之间、内存单元与端口之间的数据交换等操作，但在 8086/8088CPU 中仅支持内存单元与端口之间的直接数据交换，而且该交换方式只能通过 DMA 方式实现，不能通过执行指令来实现。8086/8088 系统不支持内存单元之间的直接数据交换，必须通过 CPU 中的寄存器作为中转。不同设备接口中端口之间的数据交换比较少见，但如果有这种需要，仍然可以设计实现，8086/8088 系统不支持这种数据交换，必须通过寄存器作为中转。值得注意的是，寄存器间的数据交换不需要使用总线，因而这种数据交换比需要总线操作的数据交换速度更快。因此，在复杂运算尚未执行完毕前，中间运算结果应尽量保存在寄存器中，以提高存取速度。

接下来，我们给出存储单元地址这一重要概念。CPU 访问任何存储单元时，都必须指出相应地址。存储单元的物理地址是一个固定长度的二进制无符号数编码。这就好像是在某一住宅小区寻找某一个人时，需要先知道他（她）的确切住址一样。住宅小区中的地址一般为 XX 幢 XX 号的形式。在计算机系统中，我们应将寄存器、内存单元、端口分别所在的 CPU、内存、设备接口理解为三个不同的住宅小区。在 8086/8088 系统中，CPU 是通过不同的读/写信号来区分它们的，而不是通过地址来区分，但这并不代表在其它计算机系统中也是这样来区分。系统中不同的内存芯片、不同的设备接口可以理解为住宅小区建筑中不同的“幢”，而此概念正对应于物理地址中的高位地址，它用于区分不同的内存芯片或不同的设备接口。同一内存芯片中的不同内存单元或同一设备接口中的不同端口则通过低位地址来区

分，这与一幢建筑中不同房间编号的概念相对应。计算机系统物理地址格式如图 3.2.3 所示。其中高位地址与低位地址的位数在不同系统中设置也不同。在图 3.2.3 中，我们假设该物理地址为内存单元的物理地址，高位地址为  $m$  位，则系统能够区分  $2^m$  个不同的内存芯片，低位地址为  $n$  位，则系统能够区分  $2^n$  个在同一内存芯片中的不同内存单元。如果将该物理地址解释为端口地址，也能得到类似的结论。

CPU 访问外部存储单元时，包括内存单元和端口，都需要在地址总线上提供完整的物理地址，否则无法定位所要访问的单元。8086/8088 系统中地址总线为 20 位，内存单元物理地址的范围为  $0 \sim 2^{20}-1$ ，以字节（8 位）为编址单位，1 个地址对应 1 个字节，总共 1M 字节的空间；而端口物理地址的范围为  $0 \sim 2^{16}-1$ ，同样以字节为编址单位，但端口地址仅使用地址总线的低 16 位，总共 64K 字节的空间。这里说明一下存储容量的度量单位，在各种不同的系统中，字节的概念是统一的，都是指 8 个连续的二进制位。但是，字的概念在不同系统中则可能不一致，通常字长有两种定义方式，一种是指 CPU 内部并行处理二进制位的最大位数；另一种则是指数据总线的宽度，即一次使用总线的读写操作能够并行操作的二进制位数。在本教材中，称前一种字长为内部字长，后一种字长为外部字长。对于 8086CPU，其内外字长都是 16 位；对于 8088CPU，其内部字长为 16 位，外部字长为 8 位。

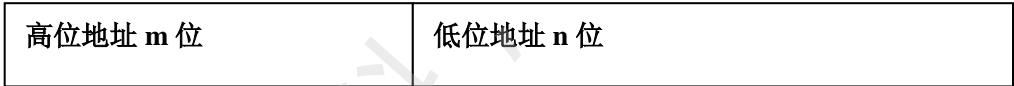


图 3.2.3 物理地址的一般构成

综上所述，计算机系统除了完成运算操作外，多数时候是在进行各种存储单元间的数据交换，在我们学习指令系统时，一定注意结合系统模型中的内容加深对指令功能、指令执行过程的理解。并且在本小节，物理地址的概念被明确提出，虽然物理地址不是在程序设计中所使用的，但最终访问存储单元时，总线上的地址都是这一形式，在以后学习指令系统时，应当有意识的贯穿存储单元的地址概念。

### 3. 3 控制信号与时序过程

在本小节中，我们将在计算机系统模型的基础上，分析最一般的控制信号，以及由这些控制信号生成的基本时序过程，使读者对计算机系统最一般的工作原理具有一种框架性的认

识。在 3.1 小节中，我们了解的计算机系统的一般组成结构，在 3.2 小节中，我们了解了计算机系统中的存储单元，并理解了系统的大量操作都是在不同存储单元间交换数据。但要使计算机系统有条不紊的工作，只具备系统结构是不够的，系统还必须为各种操作提供时序机制，使系统各部件能够按照正确的步骤完成各类操作。显然，无论是何种计算机系统，这样的时序机制都是存在的，并且，理解基本的控制信号与时序过程是理解系统中整个时序机制的基础。

### 3.3.1 控制信号

控制信号是计算机系统实施各类动态操作时生成的信号，多数控制信号由 CPU 生成，但其它功能部件也可能生成部分控制信号以配合 CPU 完成较复杂的操作。在操作寄存器时，CPU 产生的控制信号只在 CPU 内部作用于所操作的寄存器，不会出现在控制总线上；在操作外部存储单元时，包括内存单元和端口，CPU 产生的控制信号会出现在控制总线上。在这里，我们将最基本、最重要、最常见的控制信号做一个介绍。

#### (1) 时钟信号

时钟信号是一种周期性的方波信号，它最基本的控制信号，也是产生时序过程的必要条件。图 3.3.1 给出了时钟信号的一般形式，横坐标轴为时间轴，纵坐标轴为信号的幅度轴。时钟信号的周期称为时钟周期，其频率称为时钟频率，一个时钟周期通常被称为一个节拍。

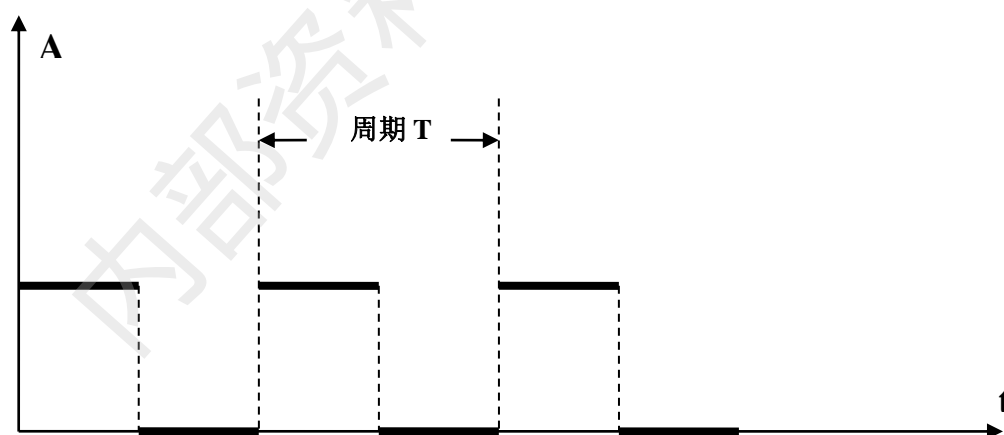


图 3.3.1 时钟信号图示

时钟信号一般由晶体振荡片产生的周期信号经过整形、分频后得到，一个计算机系统内，生成的时钟信号只有一个，所有功能部件都会通过控制总线接收到这个信号，并以此信号为时



间基准，进行有条不紊的工作。这就好像一个有周密规划的公司一样，在启动一个项目后，对每位工作人员都安排了工作任务，但这些工作任务是有先后的，后面的任务需要等待前面的任务完成后才能考试，因此，每位工作人员都要按照给定的时间限制来完成任务。那么，按照什么时间来作为标准呢？在公司里，采用的就是当地时间，例如在中国，我们采用北京时间；类似的，在计算机系统模型中，时间基准就是时钟信号，在启动一个时序过程后，各功能部件按照约定好的节拍完成各自的操作，这就保证了计算机系统能够有条不紊的工作。

## (2) 读/写信号

读信号和写信号是两种控制信号，但它们通常搭配在一起使用，因此将它们放在一起介绍。当 CPU 需要读取对某一存储单元中的数据时，它会产生读信号，并施加在相应的功能部件上；当 CPU 需要向某一存储单元保存数据时，它会产生写信号，并施加在相应的功能部件上。对于 8086/8088 CPU 芯片，针对寄存器、内存单元、端口的读写信号是不同的控制信号，但读/写信号在不同的系统中定义也有区别。总的来说，读信号针对存储单元的读操作，写信号针对存储单元的写操作，这一特征对任何计算机系统都适用。

## (3) 片选信号

片选信号是一种特殊的控制信号，它是 CPU 针对外部存储单元进行读写操作时产生的，但它是由地址总线的高位地址生成的，而非来自于控制总线的信号。物理地址中的高位地址用于区分不同的内存芯片或设备接口，这在 3.2 节中已经给予了说明。这种区分在硬件上就是通过片选信号来实现的，片选信号是依靠地址译码器生成的，其生成原理如图 3.3.2 所示。

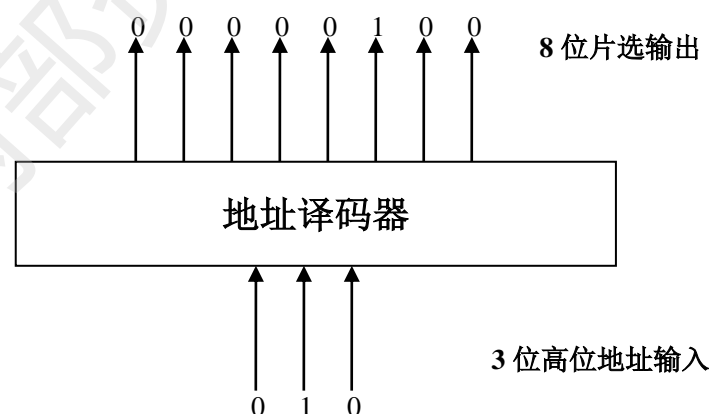


图 3.3.2 片选信号生成原理

在图 3.3.2 中，若地址译码器的输入为  $n$  位输入，则输出为  $2^n$  位，图中以 3 位输入为例。假

设地址译码器的输入为 3 位高位地址,那么这 3 位高位地址可以理解为不同内存芯片或不同设备接口的编号,地址译码器将高位地址译码后,选通编号对应的输出线路,而其余输出线路则处于未选通状态。例如,图 3.3.2 中,输入编号“010”,则选通第 2 号输出线路,使之输出“1”,而其余输出线路均输出“0”。这里输出线路编号是从第 0 号开始的,右侧为低编号输出。这些输出线路上的信号便是片选信号,它们分别连接到不同内存芯片或不同设备接口的片选端,只有当相应片选信号有效后,内存芯片或设备接口才会接受 CPU 的读写操作。由于在任意时刻,只有一个内存芯片或设备接口被选中,因此,在任意时刻,只有一个内存芯片或设备接口能接受 CPU 的读写操作。

#### (4) 中断请求/中断响应信号

中断请求与中断响应信号是一对控制信号,要完成完整的中断过程,这两个信号缺一不可。在现代计算机系统中,很多外部设备都属于中断型设备。最典型的中断型设备为键盘、鼠标等低速输入设备。在没有键盘、鼠标输入的情况下,CPU 不会访问它们的设备接口,而是执行系统程序或应用程序,仅当发生输入事件时,设备接口才会通过控制总线向 CPU 发送中断请求信号。注意,多数控制信号都是由 CPU 通过控制总线发出,其它功能部件只是接收相应的控制信号,但中断请求信号是例外的情况,它由中断型设备发出,由 CPU 接收。实质上,中断请求信号应属于外部设备的状态信号,而控制总线的全称也应当是控制/状态总线,包括 CPU 发送给其它功能部件的控制信号和其它功能部件向 CPU 发送的状态信号。如果没有在 CPU 内部设置中断屏蔽,那么 CPU 在执行完当前指令后,会暂停当前程序的执行,并向控制总线发出中断响应信号,提出中断请求的设备接口接收到中断响应信号后,会向数据总线发送中断类型号,CPU 则从数据总线接收中断类型号,并以此在内存中定位中断服务程序,执行中断服务程序调用,从而完成与设备接口的数据交换。除输入设备外,输出设备也可能采用中断方式与 CPU 交互,并且,除数据交换可以采用中断方式外,设备出现故障需要 CPU 处理时也可以向 CPU 提出中断请求。

### 3. 3. 2 时序过程

时序过程的含义在 3.1 节中介绍 CPU 功能时已作过说明,这里重新归纳一下。时序过程是指计算机系统中按照既定时间顺序、分步骤完成的动态操作过程。指令的执行过程属于

时序过程，并且也是计算机系统中主要的时序过程，但并不是全部，例如上一小节介绍的中断过程就不属于指令执行的范畴。在本小节中，我们结合前面介绍的基本控制信号，在计算机系统模型中给出一种最基本的时序过程，读者应由此建立对时序过程理解的基础，并在将来学习指令系统时，有意识的分析指令执行时可能的时序过程，以加深对指令执行的理解。

读/写操作是指令执行或其它时序过程中经常执行的，它本身也构成一个基本的时序过程。我们下面将分析 CPU 读取内存单元的时序过程。注意，这里给出的时序过程不针对任何一种具体的 CPU 芯片，只是一种概念性的说明。我们应当在此时序过程的分析中有意识的体会其顺序性、步骤性。系统设计把读操作分解为若干个步骤，每个步骤都能在一个节拍的时间内完成执行，下面分节拍列出各个步骤的操作。

- (1) 第一个节拍：CPU 将内存单元的物理地址发送到地址总线。
- (2) 第二个节拍：地址总线上信号稳定，等待片选信号的产生；CPU 将读控制信号发送到控制总线。
- (3) 第三个节拍：控制总线上的读信号已稳定，片选信号已产生并稳定；相应内存芯片被选通，低位地址信号进入内存芯片，内存芯片执行针对指定内存单元的读操作；
- (4) 第四个节拍：内存芯片读操作执行完毕，内存单元中的数据传送到数据总线；
- (5) 第五个节拍：数据总线上的信号稳定，CPU 将数据总线上的数据写入到寄存器。

从上面的例子中，我们应当认识到，任何时序过程所消耗的时间都是节拍的整数倍。对于写操作的时序过程，读者可以自己根据对读操作时序过程的理解来构造。上一小节在介绍中断请求/中断响应信号时，已对中断过程大致作了说明，读者可以进一步将其分割为以节拍为单位的步骤，构造出中断时序过程。

### 3. 4 机器指令系统

机器指令的概念在前面的章节中已经多次提到，对其概念，这里再重新归纳一次。机器指令实质上是一种二进制编码，不同编码对应不同的指令功能。并且，机器指令除了指明操作类型外，还包含了对操作数的寻址方式或操作数本身等信息。通常，一条完整的机器指令分为**操作码字段**、**寻址方式字段**、**立即数字段**和**位移量字段**几个组成部分，各字段占据的二进制位数因指令而异，如图 3.3.3 所示。在具体的机器指令中，除操作码字段是每种指令

都必须具有的字段外，可能并不会包含上述所有的字段。

操作码（OPCODE）	寻址方式	立即数	位移量（DISP）
-------------	------	-----	-----------

图 3.3.3 机器指令的一般构成

操作码字段用于标识指令功能，例如，加法指令、减法指令、逻辑与指令等等。寻址方式用于表示指令所使用的操作数使用何种寻址方式，例如，寄存器寻址、存储器寻址等等。立即数字段用于提供操作数本身，位移量字段用于提供存储单元的地址分量。机器指令格式在不同的计算机系统中各不相同，同类指令执行的时序过程在不同系统中也各不相同。某一计算机系统的机器指令全集则称为该计算机系统的机器指令系统。机器指令的执行分为读取指令、解释指令、执行指令几个步骤，这几个步骤都是通过时序过程来完成的，执行指令仅是指令执行过程的其中一个组成部分，请参照第 3.1 节对 CPU 功能的详细说明。

汇编指令实质上是机器指令的助记版本，由于使用英文助记符，它比机器指令更加容易记忆。但是值得注意的是，大多数汇编指令的格式都是从机器指令格式演化而来的，而不是随意定义的，而机器指令格式又是与 CPU 中指令译码器的设计是对应起来的。因此，我们在学习汇编指令系统时，应当推敲相应机器指令的格式，这对于类似《计算机组成原理》这样的后续课程是有帮助的。

## 习题 3

1. 请简述计算机系统的基本结构，并描述各功能部件在系统中的主要作用。
2. 系统总线可分为几类？各类总线的主要功能是什么？
3. 设备接口存在的必要性是什么？设备接口的主要功能有哪些？
4. 计算机系统中存储单元可以分为几类？它们分别位于计算机系统的哪些功能部件中？
5. 请简述时钟信号的概念，以及它在系统中的作用。
6. 请根据自己的理解，构造系统模型中 CPU 写内存单元操作的时序过程。
7. 请简述片选信号的概念，并说明其生成原理。
8. 请简述物理地址的概念及其构成。
9. 请简述机器指令的概念，并描述 CPU 执行机器指令的大致过程。
10. 请简述时序过程的概念，并说明指令的执行过程与时序过程的关系。

## 第 4 章 8086/8088 CPU

在上一章中，我们给出了计算机系统模型，它并不特别针对某一种具体的计算机系统，只是为读者快速搭建一个深入计算机系统底层的框架。并且，在上一章中，着重介绍的是整个计算机系统的基本组成以及它的工作原理。由于 CPU 是计算机系统核心部件，机器指令的执行是由它来完成的，因此，细化管理 CPU 的基本结构以及它的工作原理成为学习指令系统的基础。本章将针对 8086/8088 CPU 芯片，介绍其基本构成与工作原理，为学习下一章的 8086/8088 CPU 指令系统做好准备。读者应当注意，这一章的内容组织与上一章不同，一方面，它着重介绍 CPU 这一个功能模块的结构和原理，而不是整个计算机系统；另一方面，它针对的是特定型号的 CPU 芯片，而不是一般化的模型。

### 4.1 8086/8088 CPU 基本结构与工作原理

8086、8088 两种 CPU 芯片的区别主要在于数据引脚的数量，也即它们所连接的数据总线宽度。

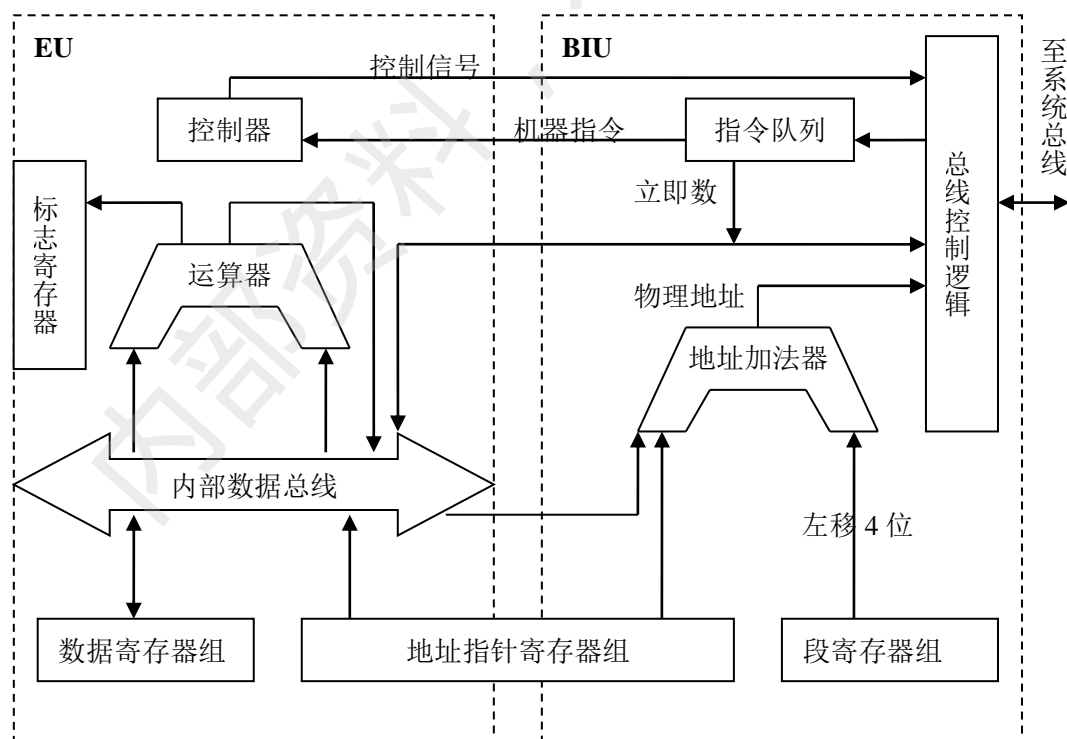


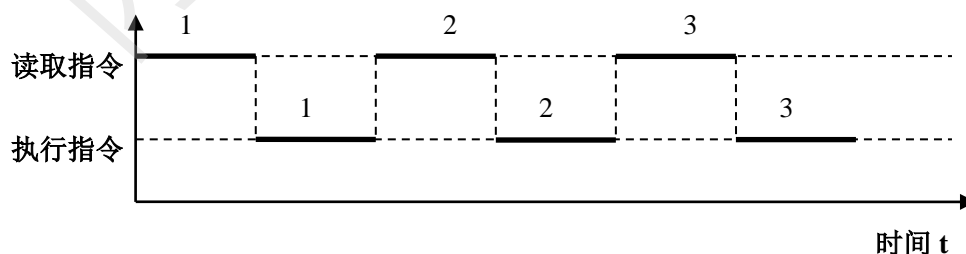
图 4.1.1 8086/8088 CPU 内部结构

8086 能够连接 16 位数据总线，一次总线操作最多可以读取一个字（在 8086/8088 中为 16 位）的数据量；而 8088 只能连接 8 位数据总线，一次总线操作最多只能读取一个字节（8 位）。但这两种 CPU 芯片的内部结构几乎是完全一致的，CPU 内部能够并行处理的最大二进制位数都是一个字。本小节介绍 8086/8088 CPU 芯片的基本结构与它们的工作原理，芯片的基本结构如图 4.1.1 所示。

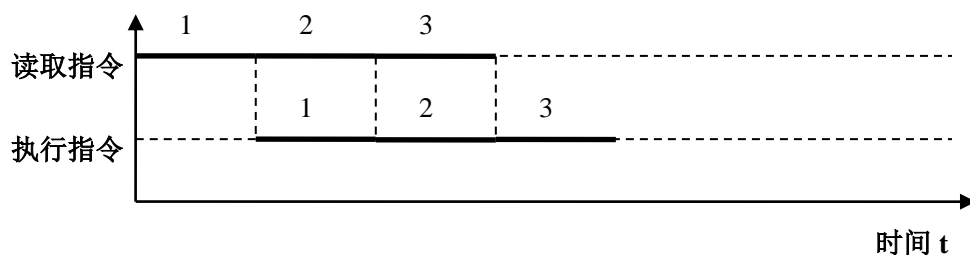
### （1）基本的流水线结构

从图 4.1.1 可以看到，CPU 内部结构分为 BIU 和 EU 两个功能模块。其中 BIU 的全称为 Bus Interface Unit，即总线接口单元，其主要功能是实现 CPU 与系统总线的信号连接、数据交换。所有与总线操作相关的时序过程都由 BIU 模块执行，其中最频繁的是内存单元、端口的数据读写操作；所有有关物理地址的计算机过程也由 BIU 完成。EU 的全称为 Execute Unit，即执行单元，它的主要功能为解释并执行指令。将 CPU 分为两个功能模块的目的在于使读取指令与执行指令这两个步骤能够在时间上重叠，从而实现基本的流水线结构。读者可以想象，如果 EU 模块所执行的当前指令不需要使用总线，那么 BIU 模块就在 EU 执行当前指令的同时启动读取下一条指令的操作。这样，执行当前指令的时间与读取下一条指令的时间就可以重叠，而加快整个程序的执行速度。当然，在时间上使两个步骤重叠的必要条件是当前指令不使用总线操作。换言之，有时这两个步骤并不能在时间上重叠。

我们可以通过一个实际例子来体会流水线结构加快程序执行速度的功能。假设有一个由 3 条指令组成的程序片段，并且假设这 3 条指令都不使用总线操作，我们通过串行读取 / 执行指令的方式和 EU / BIU 流水线方式分别来执行这段程序。那么，这两种方式在读取 / 执行指令方面消耗的时间分别如图 4.1.2（a）和 4.1.2（b）。



（a）串行读取 / 执行指令方式消耗的时间



(b) EU / BIU 流水线方式消耗的时间

图 4.1.2 串行方式与流水线方式连续执行 3 条指令时的耗时对比

在图 4.1.2 (a) 和 (b) 中，横坐标轴都代表时间，纵坐标轴代表读取指令、执行指令两种不同的操作步骤，时间段上的数字标号表示当前操作的是指令序列中哪一条指令。对比可知，由于在流水线方式中，使执行当前指令的时间与从内存读取下一条指令的时间重叠，因而大幅提高了程序的执行速度，程序中包含的指令越多，则优势越突出。在现代计算机系统中，流水线已经被设计得非常复杂，指令的执行步骤被进一步细分，流水线的级数也有 8086/8088 CPU 中的两级扩展为多级。在以后的讨论中，我们将 CPU 启动总线进行操作所耗费的时间称为**总线周期**，将 CPU 启动总线读取机器指令所耗费的时间称为**取指周期**，将执行机器指令所耗费的时间称为**执行周期**。有关流水线结构的进一步知识请读者参阅《计算机系统结构》相关教材。

## (2) 指令队列

指令队列这一部件位于 BIU 模块，它具有多个字节的存储单元。每次 BIU 模块启动读取下一条指令的操作后，将所读取到的指令（字节数随不同指令而变化）暂存在指令队列中。只要总线出现空闲，BIU 模块就会自动启动读取指令的时序操作，直到指令队列填满为止。位于 EU 中的控制器会将指令队列中的机器指令操作码字段读取出来，并译码生成相应控制信号，从而执行指令。指令队列中的机器指令被读取的顺序按照**先进先出 (FIFO)** 的队列结构原则，最先由 BIU 保存在指令队列中的指令，最先被 EU 读出执行。应当注意，读取指令这一时序过程与执行指令无关，它是由 BIU 自动启动的，因而指令队列与我们后面将要介绍的寄存器组不同，它们不能被机器指令访问。换言之，我们在设计汇编语言程序时，不能操作指令队列，它是由 CPU 自动管理的。



### (3) 控制器

控制器位于 CPU 的 EU 模块，它是 CPU 的控制中心，无论是作用于 CPU 内部的控制信号，还是 CPU 发送到系统总线的控制信号，几乎全部在这里产生，而产生控制信号的依据则来源于对机器指令操作码字段的译码。由于本教材只说明 CPU 工作的基本原理，因此指令译码器的原理与设计方法请读者参阅《计算机组成原理》相关教材。控制器按照机器指令规定的时序产生控制信号，如果控制信号需要发送到控制总线，则控制器将这些控制信号传送到 BIU 模块的总线控制逻辑这一部件，由总线控制逻辑将控制信号传递到控制总线。

### (4) 地址加法器

地址加法器位于 BIU 模块，它的功能是由逻辑地址计算物理地址，并最终由该器件将物理地址传送到 BIU 的总线控制逻辑，并由总线控制逻辑将物理地址传送到地址总线。在说明地址加法器的运算原理前，我们将先说明逻辑地址的概念，以及 CPU 对内部存储器的分段管理方式。我们在第 3 章的 3.2 节中介绍物理地址概念时，已经说明了 8086/8088 CPU 能支持的地址总线宽度以及物理地址范围，8086/8088 CPU 能支持的地址总线宽度为 20 位，对于内部存储器而言，其寻址范围为 1M 字节，而访问端口时仅使用地址总线的低 16 位，因而端口寻址范围为 64K 字节。在 CPU 内部，所有寄存器的二进制位数最大为 16 位，因此，没有哪一个寄存器能够存放 20 位的物理地址。那么，我们如何在 CPU 内部表达外部存储单元的物理地址呢？为了协调 CPU 内外地址位数的差异，8086/8088 CPU 采用了对存储器的分段管理方式，并使用逻辑地址为 CPU 内部的物理地址表达形式。

首先，解释段的概念，一个段是指位于内部存储器中的一块连续存储空间，它由物理地址连续的多个字节单元构成。那么，何谓逻辑地址？它与物理地址的关系又是怎样的呢？逻辑地址由两个 16 位的地址分量构成，其中一个为段基值，另一个为偏移量，两个分量均为无符号数编码。逻辑地址转换为物理地址运算过程可由式 (4.2.1) 来表示。

$$\begin{aligned} \text{段基址} &= \text{段基值} \times 16 \\ \text{物理地址} &= \text{段基址} + \text{偏移量} \end{aligned} \quad (\text{式 4.2.1})$$

首先，16 位的段基值左移 4 位，低位补充 4 个 0，相当于乘以  $2^4$ ，得到 20 位的段基址。段

基址为 20 位，已经可以理解为物理地址，但它仅是一个段的起始地址，要使 CPU 能够定位段内的所有内存单元，那么必须在段基址的基础上加上偏移量。偏移量的含义为距离段基址的偏移字节数。偏移量指定不同，那么所访问的内存单元也就不同。应当注意，20 位的段基址与 16 位的偏移量相加，是低位对齐的加法，加法结果即为 20 位物理地址。我们还应当注意，任意指定一个确定的逻辑地址，那么经过式 4.2.1 转换后得到的物理地址是唯一的，换言之，给定一个逻辑地址，就能唯一的定位一个内存单元。但是相反的过程并不能得到唯一的结果，给定一个物理地址，我们并不能得到唯一的段基值、偏移量分解，好在这种逆转换在系统内是没有用途的，地址加法器也只实施由逻辑地址到物理地址的转换。那么，既然在 CPU 内部使用的是逻辑地址，相应的，我们在进行汇编语言程序设计时所使用的内存单元地址也就是逻辑地址，而非物理地址。物理地址仅在 CPU 执行总线操作时出现，换言之，物理地址存在于程序的执行阶段，而非程序的设计阶段。对端口地址而言，由于其仅使用地址总线的低 16 位，因此，逻辑地址等同于物理地址，我们不去区分它们。

地址加法器所完成的操作就是将逻辑地址转换为物理地址后，提交给总线控制逻辑。这样的地址转换可能发生在 BIU 读取指令时，也可能发声在 EU 执行指令时，还可能发生在其它的时序过程中。最后，读者也许会问：段基值与偏移量分别保存在 CPU 中的什么位置？或者它们又是怎样计算得到的呢？CPU 中提供了一个段寄存器组，段基值就是由这一组 16 位段寄存器提供的。而偏移量的获取方式则丰富得多，也复杂得多，它们既可以来自于机器指令中的位移量字段，也可以来自于指令指针寄存器，或者由几种分量相加而得到。有关偏移量的获取方式，我们将在第 5 章分析寻址方式时详细讨论。

## （5）运算器

运算器位于 CPU 的 EU 模块，用于完成机器指令要求的算术运算、逻辑运算功能，是 CPU 内的运算中心，最大支持双操作数运算，例如两个操作数相加或相减等。它除了完成机器指令指定的数据运算外，还可用于生成逻辑地址中的偏移量分量。若为数据运算，其操作数可能来源于数据寄存器、内存单元、指令中的立即数；若为偏移量运算，其操作数均解释为偏移量分量，来源于地址指针寄存器、指令中的位移量。如果机器指令有这样的要求，运算器会将运算过程和运算结果的某些性质按照标志位的形式保存到标志寄存器。有关运算器的详细原理，我们将在第 5 章学习基本指令系统时体会到。

## （6）总线控制逻辑

总线控制逻辑为 BIU 模块，用于控制在适当的时刻向总线传递相应信号，或从总线上接收相应信号，CPU 只能通过总线控制逻辑才能与系统总线实现信号交互。由于总线控制逻辑是 CPU 自动进行管理的，我们的指令没办法直接操作该器件，因此在这里不详述其原理，读者可参阅《计算机组成原理》相关教材。

## 4. 2 8086/8088CPU 的寄存器组

在 8086 / 8088 CPU 的指令集中，大多数指令会使用 CPU 内部的寄存器，并且这些寄存器都是各不相同的，每个寄存器都有自己特殊的功能。因此，理解 CPU 中各寄存器的功能是进一步学习指令系统的前提条件。本小节将结合上一小节对 8086 /8088 CPU 内部结构与工作原理的分析，集中介绍这两种 CPU 芯片的寄存器组。在学完本小节后，读者也许会觉得对寄存器及其用途的理解仍然不够清晰，这是正常的，因为深入理解寄存器的功能必须结合指令系统进行学习，并通过学习丰富的示例才能达到。但是通过本小节的学习，读者应对各寄存器的大致功能有所理解。

### 4. 2. 1 数据寄存器组

数据寄存器是指用于实现指令功能的寄存器。换言之，数据寄存器是在机器指令执行时，被机器指令所使用的寄存器。从图 4.1.1 可以看到，数据寄存器都与 CPU 内部的局部数据总线相连接，使得它们可以为运算器或其它寄存器提供原始数据，即提供**源操作数**，也可以从局部数据总线接收并保存运算器的运算结果或从其它寄存器读取的数据，即保存**目的操作数**。在物理上，共有 4 个 16 位数据寄存器，分别为 AX、BX、CX、DX。AX 的全称为累加器，BX 为基址寄存器，CX 为计数寄存器，DX 为数据寄存器，这些名称似乎很奇怪，我们会在将来学习指令系统时逐步体会这些名称的含义。注意这些寄存器的名称是我们将来在汇编指令中使用的，在机器指令中，我们应当知道它们仅仅是二进制的地址编码。在逻辑上，每个 16 位的数据寄存器又可以拆分为高 8 位寄存器和低 8 位寄存器分别使用，但应注意，这些 8 位寄存器与物理上的 16 位寄存器是相互重叠的。数据寄存器的名称和相互间的关系如图 4.2.1 所示。

	15	8	7	0
AX	AH		AL	
BX	BH		BL	
CX	CH		CL	
DX	DH		DL	

图 4.2.1 数据寄存器组

从图 4.2.1 可以看到，AX 寄存器的第 0 位到第 7 位为低 8 位寄存器 AL，第 8 位到第 15 位为高 8 位寄存器 AH，对 BX、CX、DX，也有类似的情况。在程序设计时需要注意，逻辑上不同的寄存器在物理可能相互覆盖。例如，当我们修改 AL 寄存器的内容时，同时也修改了 AX 寄存器的低 8 位。这里还需说明一点，同一个寄存器可能属于不同的寄存器组，例如 BX 寄存器，它既属于数据寄存器组，也属于地址指针寄存器组，后面我们还将介绍它作为地址指针寄存器的用途。

## 4. 2. 2 段寄存器组

在 8086/8088 CPU 中，总共有 4 个 16 位段寄存器，它们分别用于保存 4 个段的段基值，为生成内存单元的物理地址提供必要条件。通常情况下，在一个程序运行之初，这 4 个段寄存器就应当初始化，之后，它们仅在特定情况下被改变。CPU 中仅有 4 个段寄存器，这也意味着在任意时刻，CPU 能够立即访问的当前段也仅有 4 个。4 个段寄存器分别为 CS、DS、SS、ES，它们的全称分别为 Code Segment、Data Segment、Stack Segment、Extra Segment，既代码段段寄存器、数据段段寄存器、堆栈段段寄存器、附加段段寄存器。

在汇编语言程序设计中，源程序是以段的形式组织起来的，最多会出现 4 种类型的段，即代码段、数据段、堆栈段、附加段，4 个段寄存器分别用于指示 4 种段在内存中的起始地址，一个段对应内存中一块连续的存储区域。代码段用于存放由机器指令组成的程序；数据段通常用于保存变量数据；堆栈段用于保存中断断点、子程序返回点、用户使用堆栈临时保存的数据等；附加段通常用于串操作，也可以根据程序员的设计而具有其它功能。虽然 4 种类型的段在源程序中都分别可以定义多个，但每种类型的段在任意时刻仅有一个可以被 CPU 访问，那就是在相应段寄存器中保存了段基值的当前段。段基值是逻辑地址中的一个分量，有关逻辑地址到物理地址的转换方法请参阅 4.1 节中有关地址加法器的介绍。

## 4. 2. 3 地址指针寄存器组

地址指针寄存器用于提供内存单元逻辑地址中的偏移量或构成偏移量的分量。有关偏移量的形成方式，我们将在下一章介绍寻址方式时给予详细说明。这里我们仅就地址指针寄存器大致的功能和使用方法给出说明。地址指针寄存器总共包括 5 个 16 位寄存器，分别为 BX、SP、BP、SI、DI，它们的全称分别为基址寄存器、堆栈指针寄存器、基址指针寄存器、源变址寄存器、目的变址寄存器。这里我们暂时不解释这些名称的含义，而是等到学习指令系统时来逐步体会它们的含义。偏移量是逻辑地址中的一个分量，有关逻辑地址到物理地址的转换方法请参阅 4.1 节中有关地址加法器的介绍。既然地址指针寄存器提供的是偏移量或其分量，那么它们一定具有搭配使用的段寄存器，因为只有这样才能构成完整的逻辑地址。在这种搭配关系中，部分是固定搭配，部分是默认搭配，读者应当在学习中加以记忆。图 4.2.2 给出了这些地址指针寄存器与段寄存器间的搭配关系。

地址指针寄存器	与段寄存器的搭配关系
BX	默认与 DS 搭配，但在程序设计时可以修改，可与任意段寄存器搭配
SP	只能与 SS 搭配
BP	默认与 SS 搭配，但在程序设计时可以修改，可与任意段寄存器搭配
SI	默认与 DS 搭配，但在程序设计时可以修改，可与任意段寄存器搭配
DI	一般指令中默认与 DS 搭配，但在程序设计时可以修改，可与任意段寄存器搭配；在串操作指令中只能与 ES 搭配

图 4.2.2 地址指针寄存器与段寄存器的搭配关系

在 8086/8088 CPU 中，数据寄存器与地址指针寄存器一并称为通用寄存器。

### 4. 2. 4 控制寄存器

8086/8088 CPU 中的控制寄存器包括指令指针寄存器 IP 和标志寄存器 FR，控制寄存器是指能够直接或间接控制程序执行流程的寄存器。IP 寄存器的英文全称为 Instruction Pointer，即指令指针，其含义非常明确，因为该寄存器中保存的是 CPU 下一条将从内存中读取的指令在当前代码段（CS 指向的段）中的首字节偏移量，这里的首字节是指多字节指令的首字节。**IP 固定与 CS 搭配使用**，形成下一条即将被读取指令的逻辑地址。CPU 中的 BIU 模块每次从内存中读取一条机器指令后，都会自动修改 IP 中的偏移量，使之指向下一条机器指令的首字节。换言之，**当执行某条机器指令时，IP 中的偏移量已经指向下一条指令**。从上述分析看来，IP 寄存器的属性本来更接近于上一小节的地址指针寄存器。但是转移指令、循环控制指令、子程序调用与返回等指令都是通过修改 IP 中的偏移量来达到控制程序流程的目的，以便实现程序中的分支、循环结构以及子程序的调用与返回等功能。因此，**由于修改指令指针能够控制程序流程，这里将它归类到控制寄存器。**

接下来我们将介绍寄存器中最重要、使用最复杂、初学者最难理解的寄存器，即标志寄存器 FR（或 PSW），其英文全称为 Flag Register（或 Program Status Word）。这是 8086/8088 CPU 中唯一按位操作的寄存器，该寄存器为 16 位寄存器，标志寄存器中的有效的标志位共有 9 位，9 个标志位各自具有独特的含义。其中，CF、PF、AF、ZF、SF、OF 为六个状态标志，用于反映最近一次影响标志位的算术或逻辑运算中，运算过程、运算结果的一些性质；TF、IF、DF 为三个控制标志，用于控制 CPU 对某些特定事件的处理方式，以及控制 CPU 的工作模式。多数条件转移指令和部分循环控制指令会根据某些状态标志的取值来确定是否改变程序的执行流程，换言之，**FR 的状态标志能够间接影响程序执行的流程，因此，FR 被归类到控制寄存器。**状态标志位是通过运算指令根据运算过程、运算结果，通过特定逻辑电路、按照一定的生成规则来间接影响的；而控制标志位则无特定逻辑电路来生成它们，程序员可根据需要使用指令直接对它们置 1 或清 0，以达到控制的目的。各标志位在标志寄存器中的位置如图 4.2.3 所示。下面我们将逐个介绍各标志位的含义。注意，这里所指出的仅仅是各标志位主要的含义，而不是其所有的含义。由于在各指令中，标志位的解释可能出现不同，在这里不可能将所有情况全部列举。

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

图 4.2.3 标志寄存器中的标志位

### (1) 进位标志位 (CF)

执行加、减算术运算指令时，如果最高位（字节为第 7 位，字为第 15 位）产生进位或借位，那么 CF 标志将被置 1，否则 CF 标志被清 0。CF 标志的这一性质可用于判断无符号数编码的加减运算是否溢出，假定操作数解释为无符号数编码，执行加减运算指令后，若 CF 为 0，则运算无溢出，若 CF 为 1，则运算溢出。另外，更重要的是，使用 CF 标志位这一性质可以实现高、低字间运算的衔接，从而实现长操作数运算，突破机器字长的限制。这一点在我们分析加减法指令时详细介绍。

执行移位指令时，按照顺序被移出的最后一位保存在 CF 中；执行部分逻辑运算指令时，CF 标志会被强置为 0。应当注意，部分算术运算指令不影响 CF 标志。

### (2) 奇偶标志位 (PF)

执行算术或逻辑运算后，如果运算结果的低 8 位中包含偶数个为“1”的数据位，PF 将被置 1，如果包含奇数个“1”，PF 将被清 0。PF 标志位可以用于编码的奇偶校验，奇偶校验可用于网络中数据传输错误的检测。

### (3) 辅助进位标志位 (AF)

执行加、减算术运算指令时，如果第 3 位（最低位为第 0 位）产生进位或借位，则 AF 标志被置 1，否则 AF 标志被清 0。AF 标志的产生规则与 CF 一致，只是判断的进借位位置不同。AF 标志主要应用于 BCD 码运算调整指令，这将在介绍数值运算程序设计一章中详细说明。应当注意，部分逻辑运算指令会影响 AF 标志，但其影响无意义。

### (4) 零值标志位 (ZF)

当执行算术运算或逻辑运算指令时，如果运算结果为 0，则 ZF 标志被置 1，否则 ZF 标志被置 0。ZF 标志的用途非常广泛，例如，比较两个编码是否相等、判断计数是否为 0、判断存储单元中指定位的取值等。我们在学习指令系统以及程序设计时逐步体会 ZF 标志的重要性。

### (5) 符号标志位 (SF)

当执行算术运算或逻辑运算指令后，SF 标志与运算结果的最高位保持一致，当程序员将参加运算的操作数解释为补码时，SF 可以看作运算结果的符号位。但应当注意，当程序员未将操作数解释为补码时，SF 无意义；并且，当补码运算溢出时，运算结果的符号位会出现错误，SF 是运算结果符号位的直接反映，因而它也会出现错误。SF 标志位在条件转移指令中通常与 OF 标志配合使用，以判断两个补码间的大小关系。

### (6) 溢出标志位 (OF)

当执行算术运算指令后，如果按照补码解释的运算出现溢出，则 OF 标志被置 1，如果按照补码解释的运算没有溢出，则 OF 标志被清 0。仅当程序员将运算指令的操作数解释为补码时，OF 标志才有意义。OF 标志可用于判断补码运算是否溢出，也可与 SF、ZF 配合使用，以判断两个补码间的大小关系。在部分逻辑运算指令中，OF 标志会被强置为 0。

### (7) 单步跟踪标志位 (TF)

TF 标志位为控制标志位，当它被清 0 时，CPU 工作于连续执行指令的工作模式下，它会不断从内存中读取机器指令，并不断地解释和执行它们；当 TF 标志被置 1 时，CPU 工作于单步模式下，每执行完当前指令后，CPU 内部就会产生一次单步中断，将当前 CPU 中寄存器、标志位的取值都在屏幕上显示出来，并暂停执行。正是由于 TF 标志能够触发单步中断，才使我们能够单步调试机器指令（或汇编指令）构成的程序。因此，我们可以称 TF 标志位为单步调试程序的硬件基础，不仅在 8086/8088 CPU 中有这种单步工作模式，在其它型号的 CPU 中，也有类似的工作模式。

### (8) 中断使能标志位 (IF)

IF 标志位为控制标志位，当它被清 0 时，CPU 不会响应任何可屏蔽中断，但仍然会响应不可屏蔽中断；当它被置 1 时，CPU 会响应所有可屏蔽中断和不可屏蔽中断。可屏蔽中断通常对应于外部设备的数据输入、输出事件，而不可屏蔽中断往往对应着系统中的故障或紧急事件。

### (9) 方向标志位 (DF)

DF 标志位为控制标志位，用于控制串操作指令的操作方向。当它被清 0 时，串操作指



令按照地址递增的方向进行操作；当它被置 1 时，串操作指令按照地址递减的方向进行操作。在我们介绍非数值处理一章的内容时，会详细分析串操作指令。

最后，我们讨论**标志位的有效性**。标志位提供的信息并不总是有效的，一个标志位提供的信息是否有效，**取决于三个重要的必要条件**：

**(1) 首先，我们所关心的运算指令是否会影响我们判断的标志位？**不同的指令对标志位的影响是不同的，在程序设计时需要注意，所使用的指令是否会影响我们判断的标志位。例如，INC 指令的功能为把指定存储单元中的数据加 1，但是该指令并不影响 CF 标志，因此我们根据 CF 标志来判断 INC 指令的执行结果就是错误的。

**(2) 其次，我们所关心的指令是否对标志位形成了有意义的影响？**部分指令对某些标志位会产生影响，会改变标志位的取值，但这种改变没有任何意义。例如 AND 指令，其功能为逻辑与操作，它要影响 AF 标志，但无任何意义。对这种无意义的标志位影响也需要特别注意，因为这些指令确实改变了相应标志位，只是对这种影响不作任何有意义的解释，对我们的程序设计也没有任何帮助。但是如果在程序设计中不引起重视，认为这些标志位没有改变，也会引起一些设计上的错误。

**(3) 最后，程序设计人员对操作数的解释是否与标志位的含义相符？**部分标志位能够被有效使用是有操作数假定的。例如，我们可以使用 OF 标志判断补码运算是否溢出，但如果设计人员将运算指令中的操作数解释为无符号数编码，而没有解释为补码，那么使用 OF 标志来判断运算结果是没有任何意义的。作为程序设计人员，在设计汇编语言程序时，一定要明确自己所使用的数据是何种编码，这样才能避免标志位使用的混淆。

综上所述，我们在程序设计中使用的标志位是否提供了有效信息，取决于人、机两方面的因素，上面这三个必要条件缺一不可。在将来的程序设计练习中，请读者注意将对这三个必要条件的理解贯彻始终。

## 4. 2. 5 寄存器的隐含使用与特定使用

寄存器的功能、以及它们在指令中的使用方法都是需要记忆的，并且对于初学者而言，这种记忆过程可能是比较困难的。但在刚开始学习汇编语言时，这种记忆是非常必要的。除了寄存器的常规使用方法外，我们在学习指令系统时，还应特别注意寄存器的隐含使用和特定使用。如果对这些细节不加以注意，那么容易导致我们在进行程序设计时出现语法错误或逻辑错误。

寄存器的隐含使用是指在汇编指令的格式中，我们并观察不到该寄存器的存在，但是它却确实被该指令使用了，并且其中存放的数据也可能发生改变。虽然这里还没有开始讲解指令系统，但我们可以通过简单例子来说明。寄存器的隐含使用如例 4.2.1 所示。在以后的讨论中，我们采用“(寄存器名称)”的形式来表示寄存器中存放的数据内容。

**例 4. 2. 1** 下列汇编指令中存在寄存器的隐含使用。

(1) `PUSH AX`

该指令将 (AX) 压栈，隐含使用 (SP) 作为指向堆栈栈顶的偏移量，压栈操作完成后，(SP) 会被减 2。

(2) `MUL BL`

该指令完成无符号数编码的乘法操作，将 (BL) 与 (AL) 相乘，乘法结果保存在 AX 中。

该指令对 AL、AX 寄存器的使用都是隐含使用方式。

如果对于寄存器的隐含使用不了解，那么对指令隐含引起的寄存器内容改变也就不会理解，最终在程序设计时因此而导致设计的逻辑错误，而在调试程序时查找逻辑错误是相对困难的。

寄存器的特定使用是指在汇编指令的格式中，某一操作数位置只能使用特定的寄存器，而不能使用其它寄存器。寄存器特定使用的示例如例 4.2.2。

**例 4. 2. 2** 下列汇编指令中存在寄存器的特定使用。

(1) `IN AL, 60H`

该指令读取 60H 号端口中的 8 位数据，并保存到 AL 寄存器，这里 AL 寄存器的使用属于特定使用。

(2) `SHL AX, CL`

该指令对（AX）实施逻辑左移操作，左移的位数有（CL）给出，这里 CL 寄存器的使用属于特定使用。

如果对寄存器的特定使用不熟悉，不会导致程序设计的逻辑错误，但会导致语法错误，使源程序无法通过汇编。读者在后面的学习过程中，应当对寄存器的隐含使用、特定使用给予充分的重视。

## 习题 4

1. 8086/8088 CPU 被划分哪两个功能模块？这样划分的目的是什么？
2. 请简述逻辑地址、物理地址的概念，并说明二者间的转换关系。
3. 8086/8088 CPU 中的寄存器被分为哪几个类别，它们分别的大致功能是什么？
4. IP 寄存器的功能是什么？（IP）在哪些情况下会被修改？
5. 8086/8088 CPU 中的指令队列是如何工作的？它与 CPU 中的寄存器有何不同？
6. CF、OF 标志的含义是什么？它们分别在什么时候提供的才是有效的标志位信息？
7. TF 标志的含义是什么？它的重要性体现在哪里？
8. 请按照你自己的理解，简述状态标志位、控制标志位的区别。
9. 请简述状态标志位提供有效信息的必要条件。
10. 请说明寄存器隐含使用、特定使用的含义。

## 第 5 章 8086/8088 基本指令系统

### 5.1 汇编指令的基本格式

本章我们将正式进入 8086/8088 CPU 指令系统的学习。首先我们给出 8086/8088 CPU 中汇编指令的基本格式。由于汇编指令格式是由机器指令格式演化而来，因此，尽管我们没有直接讨论机器指令，读者也能体会到机器指令的构成。**构成汇编指令的基本元素包括操作助记符与操作数（或操作数地址）。**操作助记符指明指令的功能，而操作数指明指令操作的数据，**尽管我们把它称为“操作数”，但它实质上是某种编码，而非通常意义下的数。**按照指令中明确给出的操作数数量，8086/8088 CPU 的指令可以分为双操作数指令、单操作数指令、无操作数指令三类。我们通过例 5.1.1 来说明三类指令格式。

**例 5.1.1** 本例给出汇编指令的基本格式。

(1) 双操作数指令

MOV	AX,	BX
操作助记符	目的操作数（地址）	源操作数（地址）

上例给出了双操作数指令的一般格式，操作助记符在前，目的操作数在中间，源操作数在最后，目的与源操作数之间用逗号隔开。该指令的功能为将（BX）保存到 AX 寄存器。这里，**源操作数**是指提供给指令作为原始数据的操作数，**目的操作数**则是指操作完成后的结果数据。实际上，本例中给出的两个操作数都是寄存器地址，而且在多数指令中，除非在源操作数位置给出立即数外，其余情况基本都是给出操作数地址。因此，在以后的讨论中，除非特别说明，我们将不严格区分“操作数”与“操作数地址”这两个概念，统一将它们叫做操作数。

(2) 单操作数指令

NOT	AX
操作助记符	源、目的操作数

上例给出了单操作数指令的一般格式，操作助记符在前，单操作数在后，在不同的指令中，该操作数可能被解释为源操作数、目的操作数，或者既是源操作数又是目的操作数。在本例中，NOT 指令的功能为逻辑非操作，它会将 (AX) 按位取反，然后保存回 AX 寄存器。因此，在本例中，AX 既提供原始数据，又作为保存运算结果的地址，充当了两个角色。

### (3) 无操作数指令

#### NOP

操作助记符

上例给出了无操作数指令的一般格式，在无操作数指令中，没有明确给出的操作数，只有操作助记符。本例中，NOP 指令的功能为空操作，它什么操作都不会做，只会让 CPU 空转三个节拍。在上例中，我们也能体会到，**作为一条完整的指令，操作助记符是必不可少的构成元素。**

现在，读者应当清楚理解了汇编指令的基本格式。接下来，我们将在汇编指令中具有共通性的一些重要问题提出来，以便读者在正式开始学习指令功能时能够引起对这些问题的注意。

(1) **指令中明确给出的操作数数量不一定与指令实际使用的操作数数量相符**，因为在指令中可能出现某个操作数可能同时具备源、目的操作数双重含义，因而这种操作数实质上应理解为两个操作数，只是它们在物理上重叠罢了；并且，指令中可能出现对操作数的隐含使用。因此，指令中实际的操作数数量大于等于指令中明确给出的操作数数量。

(2) **双操作数指令中仅允许最多一个操作数为内存单元或端口，不能两个操作数同为内存单元或同为端口**。此限制存在于 8086/8088 CPU 指令系统中，但并不具有推广性质，在其它 CPU 中可能会支持这样的一些操作方式。

(3) **单操作数指令中，操作数不能使用立即数，只能使用寄存器、内存单元**。此限制存在于 8086/8088 CPU 指令系统中，但并不具有推广性质。由于这里还没有正式介绍立即数寻址的概念，因此，请读者在学完下一小节中的立即数寻址后，再回过头来加深理解。

## 5. 2 寻址方式

在学习指令功能前，我们先讨论一个重要的、在指令系统中带有普遍性的问题，即指令中操作数的寻址方式。**寻址方式是指指令获取操作数的方式**，而获取操作数的关键在于其地址的获取，因此**寻址方式也可解释为指令获取操作数地址的方式**。在任何指令中都存在寻址方式，它在指令系统中具有普遍性。因此，掌握寻址方式是精确理解指令功能的基础。值得注意的是，一条指令中可能出现多个操作数，而各个操作数可能使用不同的寻址方式，即在一条指令中，可能会出现多种寻址方式。换言之，寻址方式的概念是针对操作数的，而非针对指令。

### 5. 2. 1 寄存器寻址方式

如果操作数在寄存器中，则指令中以寄存器名称（地址）的形式给出，并且称该操作数的寻址方式为寄存器寻址方式。由于寄存器在 CPU 内部，CPU 访问寄存器时不需要启动总线操作，因而寄存器寻址方式是所有寻址方式中速度最快的。寄存器寻址方式的示例如例 5.2.1 所示。

**例 5. 2. 1** 以下指令中的源操作数、目的操作数均为寄存器寻址方式。

```
MOV  AX, BX      ; (BX) 保存至 AX
ADD  AL, DL      ; (AL) 与 (DL) 相加后，运算结果保存至 AL
SUB  BX, CX      ; (BX) 减去 (CX)，运算结果保存至 BX
```

### 5. 2. 2 立即数寻址方式

如果操作数包含在机器指令内，以立即数字段的形式体现，则称该操作数为立即数寻址方式。立即数寻址方式仅能用于源操作数寻址，因为立即数是操作数本身，而不是地址，不能用于保存目的操作数。并且，在单操作数指令中不能使用立即数寻址。由于立即数是包含在机器指令内部的，当 BIU 模块启动取指周期从内存读取机器指令时，立即数就随同机器指令被保存在 BIU 的指令队列中，EU 模块执行指令时，只需从指令队列中取得立即数，而不需要单独启动总线操作来读取。在上一章介绍 CPU 流水线结构时已经说明了 BIU 与 EU 的并行工作机制，只要当前指令不使用总线，则下一条指令的取指周期与当前指令的执行周

期重叠，因此在部分情况下，立即数的获取可以认为是在 CPU 内部完成的。因此，立即数寻址方式的速度仅次于寄存器寻址方式，但比其它需要在指令执行周期中启动总线操作的寻址方式要快。立即数寻址方式的示例如例 5.2.2 所示。

**例 5. 2. 2** 下面的指令中都使用了立即数寻址方式

`MOV AL, 25`

源操作数为立即数寻址，目的操作数为寄存器寻址，该指令将立即数 25 传送至 AL 保存。

`ADD BL, 42`

源操作数为立即数寻址，目的操作数为寄存器寻址，该指令完成 (BL)+42 的运算，运算结果保存至 BL 寄存器。

## 5. 2. 3 存储器寻址方式

如果操作数位于内存单元中，则称该操作数为存储器寻址。读者应注意存储器寻址与立即数寻址的区别。最初立即数与存储器寻址的操作数都在内存单元中，但立即数是随机器指令一起在 BIU 的取指周期读取到 CPU 内部的，而存储器寻址却发生在指令的执行周期，需要在执行周期内单独启动总线操作来完成。存储器寻址比寄存器寻址、立即数寻址速度都要慢。要定位一个内存单元，需要确定其物理地址，而通过上一章的学习，我们知道物理地址是由逻辑地址转换得到的，逻辑地址则由段基值与偏移量构成。在上一章介绍逻辑地址概念时，我们曾经提到过偏移量可以由多种方式来获取，不同的偏移量获取方式则形成了不同的存储器寻址方式。内存单元的偏移量可以看作是由三种偏移量分量组合相加得到的，这三种分量分别为机器指令位移量字段提供的位移量、基址寄存器提供的基址分量、变址寄存器提供的变址分量。不同的分量组合就形成不同的存储器寻址。在 8086/8088 系统中，共有 5 种不同的存储器寻址方式，我们将逐个进行说明。在术语方面，偏移量也被称为**有效地址**，其英文名称为 **effective address**，简写为 **EA**，在以后的讨论中，这几种术语都有可能出现。

### (1) 直接寻址方式

如果直接使用机器指令中的位移量字段作为内存操作数的偏移量，则称该操作数为直接寻址方式。直接寻址方式的 EA 计算公式如式 5.2.1 所示，其示例如例 5.2.3 所示。

$$EA = DISP$$

(式 5.2.1)

式 5.2.1 中 DISP 表示位移量，即 disparity，仅在直接寻址方式中，偏移量才等于位移量。在程序设计中，偏移量始终被看作无符号数编码，但位移量既可以看作无符号数编码，也可看作补码，切勿混淆这两个概念。在初学存储器寻址时，建议读者将位移量理解为无符号数编码，关于它作为补码的用法，在我们学习了条件转移指令后就会清楚。

**例 5.2.3** 以下指令中使用了直接寻址方式

MOV AL, [1000H]

该指令源操作数为直接寻址方式，目的操作数为寄存器寻址方式，方括号内为十六进制的数值位移量（注意在语法上与立即数的区别），它将直接作为偏移量，如果没有指定段前缀，则默认使用（DS）为段基值。该指令将逻辑地址为（DS）：1000H 的字节单元数据从总线读取出来，并保存在 AL 中。

MOV ES: [0100H], BX

该指令源操作数为寄存器寻址，目的操作数为直接寻址，指明了段前缀 ES，即指明使用（ES）作为段基值。该指令将（BX）通过总线写入字内存单元（16 位），字内存单元的逻辑地址为（ES）：0100H。

MOV VAR1, BL

该指令源操作数为寄存器寻址，目的操作数为直接寻址，指令中所给位移量是一个变量名称，其实质为符号位移量，在汇编后，符号位移量会被替换为数值位移量，二者本质上是相同的。直接寻址方式中的符号位移量默认使用（DS）作为段基值。

MOV VAR1+2, AL

该指令源操作数为寄存器寻址，目的操作数为直接寻址，变量名称后加了一个常量 2，表示 EA 为 VAR1 对应的数值位移量加 2，该加法运算在汇编阶段完成，而不是指令执行阶段，因此寻址方式仍然属于直接寻址。

## (2) 寄存器间接寻址方式



如果内存操作数的偏移量由地址指针寄存器 **BX**、**BP**、**SI**、**DI** 其中之一提供，则称该操作数为寄存器间接寻址方式。其中，(**BX**)、(**BP**) 称为基址分量，(**SI**)、(**DI**) 称为变址分量。寄存器间接寻址的 **EA** 计算公式如式 5.2.2 所示，其示例如例 5.2.4 所示。

$$\begin{aligned} EA &= (BX) \\ EA &= (BP) \\ EA &= (SI) \\ EA &= (DI) \end{aligned} \quad (\text{式 5.2.2})$$

**例 5.2.4** 以下指令中都使用了寄存器间接寻址方式

**ADD CL, [BX]**

该指令的目的操作数为寄存器寻址，源操作数为寄存器间接寻址，使用 (**BX**) 基址分量直接作为偏移量，默认使用 (**DS**) 作为段基值。该指令将 (**BX**) 指示的字节内存单元数据与 (**CL**) 相加，运算结果保存在 **CL** 寄存器。

**SUB ES: [SI], AX**

该指令的源操作数为寄存器寻址，目的操作数为寄存器间接寻址，使用 (**SI**) 变址分量直接作为偏移量，使用 (**ES**) 作为段基值。该指令用 (**SI**) 指示的字内存单元数据减去 (**AX**)，运算结果保存在字内存单元中。

读者应注意，在寄存器间接寻址方式中，**BX**、**SI**、**DI** 默认与 **DS** 搭配，**BP** 默认与 **SS** 搭配。

### (3) 基址寻址与变址寻址

如果内存操作数的偏移量由基址分量与位移量分量相加得到，则称该操作数为基址寻址方式；如果内存操作数的偏移量由变址分量与位移量分量相加得到，则称该操作数为变址寻址方式。这两种寻址方式的 **EA** 计算公式分别如式 5.2.3 与式 5.2.4 所示，示例如例 5.2.5 所示。

基址寻址的 **EA** 计算：

$$\begin{aligned} EA &= (BX) + DISP \\ EA &= (BP) + DISP \end{aligned} \quad (\text{式 5.2.3})$$

变址寻址的 EA 计算:

$$\begin{aligned} EA &= (SI) + DISP \\ EA &= (DI) + DISP \end{aligned} \quad (\text{式 5.2.4})$$

**例 5. 2. 5** 下列指令中部分使用了基址寻址，部分使用了变址寻址

`MOV AL, [BP]100H`

该指令的目的操作数为寄存器寻址，源操作数为基址寻址， $EA = (BP) + 0100H$ ，默认使用 (SS) 作为段基值。

`MOV BYTE PTR [SI]220H, 30H`

该指令的源操作数为立即数寻址，目的操作数为变址寻址， $EA = (SI) + 0220H$ ，默认使用 (DS) 作为段基值。注意，“**BYTE PTR**”指明内存单元类型为字节类型，在无寄存器寻址的指令中，需指明内存单元的类型，否则数据类型将出现二义性，汇编时将出现一个警告错误。

读者应注意，在基址和变址寻址中，BX、SI、DI 默认与 DS 搭配，而 BP 仍然默认与 SS 搭配。

#### (4) 基址变址寻址

如果内存操作数的偏移量由基址分量、变址分量、位移量分量三种分量相加得到，那么称该操作数为基址变址寻址。其中，基址分量由 (BX) 或 (BP) 提供，变址分量由 (SI) 或 (DI) 提供。基址变址寻址的 EA 计算公式如式 5.2.5 所示，示例如例 5.2.6 所示。

$$EA = \begin{Bmatrix} BX \\ BP \end{Bmatrix} + \begin{Bmatrix} SI \\ DI \end{Bmatrix} + DISP \quad (\text{式 5.2.5})$$

**例 5. 2. 6** 下面指令中均使用了基址变址寻址方式

`AND AL, [BX][SI]0020H`

该指令的目的操作数为寄存器寻址，源操作数为基址变址寻址， $EA = (BX) + (SI) + 0020H$ ，默认使用 (DS) 为段基值。该指令将字节内存单元数据与 (AL) 作逻辑与运算，运算结果保存在 AL 中。

OR [BP][DI]0010H, BL

该指令的源操作数为寄存器寻址，目的操作数为基址变址寻址， $EA = (BP) + (DI) + 0010H$ ，默认使用 (SS) 作为段基值。该指令将 (BL) 与字节内存单元数据作逻辑或运算，运算结果保存在内存单元中。

读者应注意，在基址变址寻址中，默认的段寄存器搭配由基址寄存器决定，BX 默认与 DS 搭配，BP 默认与 SS 搭配。

## 5. 2. 4 其它寻址方式

除了上面介绍的寄存器寻址、立即数寻址、存储器寻址方式外，指令中的操作数还可能出現其它寻址方式。由于这些寻址方式与具体的指令相关，我们在这里仅给出基本概念，其详细解释将结合指令系统的学习来展开。

### (1) 串操作寻址方式

只要所使用的指令为串操作指令，则**内存操作数**的寻址方式均为串操作寻址方式。我们将在非数值处理程序设计一章中体会这种寻址方式。

### (2) 端口寻址方式

在 8086/8088 CPU 指令系统中，能够访问端口的指令只有两种，in 指令用于读端口，out 指令用于写端口。在端口读写指令中，位于端口中的操作数均为端口寻址方式，其中又可分为直接端口寻址方式和间接端口寻址方式。我们对端口寻址方式在输入/输出程序设计一章中给出详细解释。

### (3) 隐含寻址方式

在上一章中，我们提到了**寄存器的隐含使用**，实际上，在众多指令中，**对内存单元的隐含使用也是经常出现的**。如果操作数在指令中没有明确给出，但指令隐含地使用了它，那么这些隐含操作数的寻址方式统称为隐含寻址方式。普通的寻址方式都是由机器指令的寻址方式字段来指定，但隐含寻址方式却是由机器指令的操作码字段来确定的。换言之，指令的功能确定其对应的隐含寻址。从这里我们也能体会到汇编指令格式与机器指令格式的一致性，因为，对于**机器指令中未使用寻址方式字段来寻址的操作数，在汇编指令中都是不可见的**。

## 5.3 基本指令系统

在本小节中，我们将正式开始讨论 8086/8088 CPU 的指令系统。按照指令的功能分类，8086/8088 指令可分为传送类指令、算术运算类指令、位操作类指令、程序转移类指令、串操作类指令、处理器控制指令。在本章内容中，为了符合初学者的学习规律，我们仅讨论指令系统中的基本部分，包括传送类指令、算术运算类指令中的加法与减法指令、位操作类指令、处理器控制指令。对于算术运算类指令中的乘、除法指令、BCD 码调整指令，我们放在数值运算程序设计一章中，结合程序设计示例加以介绍；对于程序转移类指令中的无条件转移指令、条件转移指令、循环指令，我们放在分支与循环程序设计一章加以介绍；程序转移类指令中的子程序调用、返回指令、中断调用、返回指令将放在子程序与系统调用一章中介绍；串操作类指令放在非数值处理程序设计一章中介绍。

### 5.3.1 传送类指令

传送类指令的功能为将源操作数复制一份，并传送至目的操作数保存，除非标志寄存器是指令的目的操作数，否则传送类指令不影响任何标志位。下面将逐条介绍传送类指令。

#### (1) 数据传送指令 (Move)

指令格式: `MOV DEST, SRC`

指令功能:  $(SRC) \Rightarrow DEST$  或  $SRC \Rightarrow DEST$

标志位影响: 无

指令格式中的 MOV 为指令助记符，是英文单词 move 的缩写，DEST 表示目的操作数，SRC 表示源操作数，若源操作数为地址形式，则将地址所指示的存储单元数据传送至目的操作数保存，若源操作数为立即数，则将立即数传送至目的操作数保存。应当注意，双操作数指令中只能有一个内存操作数出现，若要实现两个内存单元的数据传送，则需要使用两条 MOV 指令，并通过通用寄存器作中转才能实现，如例 5.3.1 所示。

**例 5.3.1** 将字节内存单元 VA1 中的数据传送至字节内存单元 VA2 中保存。

```
MOV AL, VA1
```

```
MOV VA2, AL
```

MOV 指令不能对段寄存器直接传送立即数，需要通过通用寄存器中转，并且 MOV 指令也不能直接在两个段寄存器之间直接传递数据，需要通过通用寄存器中转，这是 8086/8088 CPU 结构所限定的，如例 5.3.2 所示。

**例 5.3.2** 将立即数 1100H 传送至段寄存器 DS、ES 保存。

```
MOV AX, 1100H
```

```
MOV DS, AX
```

```
MOV ES, AX
```

## (2) 交换指令 (Exchange)

指令格式: XCHG DEST, SRC

指令功能:  $(SRC) \Rightarrow DEST$   
 $(DEST) \Rightarrow SRC$  (两个操作同时完成)

标志位影响: 无

该指令将源操作数与目的操作数交换保存，指令中不能使用立即数，也不能使用段寄存器，与其它双操作数指令一样，最多只能出现一个内存操作数。

**例 5.3.3** 交换字节内存单元 VA1 与字节内存单元 VA2 中的数据。

```
MOV AL, VA1
```

```
XCHG AL, VA2
```

```
MOV VA1, AL
```

## (3) 取标志位指令 (Load register AH from Flag)

指令格式: LAHF

指令功能:  $(FR)_{0\sim7} \Rightarrow AH$

标志位影响: 无

该指令将标志寄存器低 8 位数据传送到 AH 保存。读者应注意, 该指令是无操作数指令, 但实质上具有 AH、标志寄存器两个操作数, 对 AH 寄存器、标志寄存器的寻址方式均为隐含寻址方式。AH、FR 没有出现在指令格式的操作数位置, 而是成为指令助记符的一部分, 这与“机器指令隐含寻址时操作数由操作码字段指出”这一含义是对应的。在程序设计中, 若需要暂时保护标志寄存器的低 8 位, 可以使用该指令实现。

#### (4) 存标志位指令 (Store register AH into Flag)

指令格式: SAHF

指令功能:  $(AH) \Rightarrow FR_{0\sim7}$

标志位影响: 标志寄存器低 8 位, 包括 SF、ZF、AF、PF、CF

该指令的功能与 LAHF 正好相反, 将 (AH) 传送到标志寄存器低 8 位保存, 由于标志寄存器是该指令的隐含目的操作数, 因此位于低 8 位范围的 5 个状态标志将受到影响。在 SAHF 指令中, AH 和标志寄存器均为隐含寻址方式。在程序设计中, 若要将先前保存在 AH 中的标志位恢复到标志寄存器低 8 位, 则可通过该指令实现。

#### (5) 入栈指令 (Push)

指令格式: PUSH SRC

指令功能:  $(SP) - 2 \Rightarrow SP$   
 $(SRC) \Rightarrow (SP)$

标志位影响: 无

该指令先将 (SP) 减 2, 使之指向一个空的栈顶, 再将源操作数传送至栈顶保存。该指令为单操作数寻址, 源操作数必须为 16 位 (字类型) 的寄存器或内存单元, 不能为 8 位存

储单元或立即数。但该指令实质上有两个操作数，除源操作数外，隐含使用了（SP）指向的栈顶内存单元作为目的操作数。在程序设计中，如果需要在堆栈中临时保存数据，可以使用 PUSH 指令实现。入栈指令的示例与出栈指令合并在一起，见例 5.3.4。

#### （6）出栈指令（Pop）

指令格式：POP DEST

指令功能： $((SP)) \Rightarrow DEST$   
 $(SP) + 2 \Rightarrow SP$

标志位影响：无

出栈指令的功能与入栈指令相反，它先将栈顶字单元中的数据传送至目的操作数保存，然后将（SP）加 2，以丢弃出栈数据并修改栈顶。指令功能表达式中（（SP））表示由（SP）指示的栈顶内存单元的内容。出栈指令对操作数的要求与入栈指令一致，且目的操作数只能是字类型。在程序设计中，如果要将先前保存在堆栈中的数据恢复到指令存储单元中，那么可以使用 POP 指令来实现。入栈、出栈操作均遵循堆栈结构的后进先出原则（LIFO），即最先入栈的数据最后才能出栈。下面，我们通过例 5.3.4 来体会入、出栈操作。

**例 5.3.4** 假设（AX）=1020H，（BX）=0030H，（SP）=0040H，下列程序片段先将（AX）、（BX）入栈保存，然后再出栈恢复，通过图示可以观察到堆栈的变化。

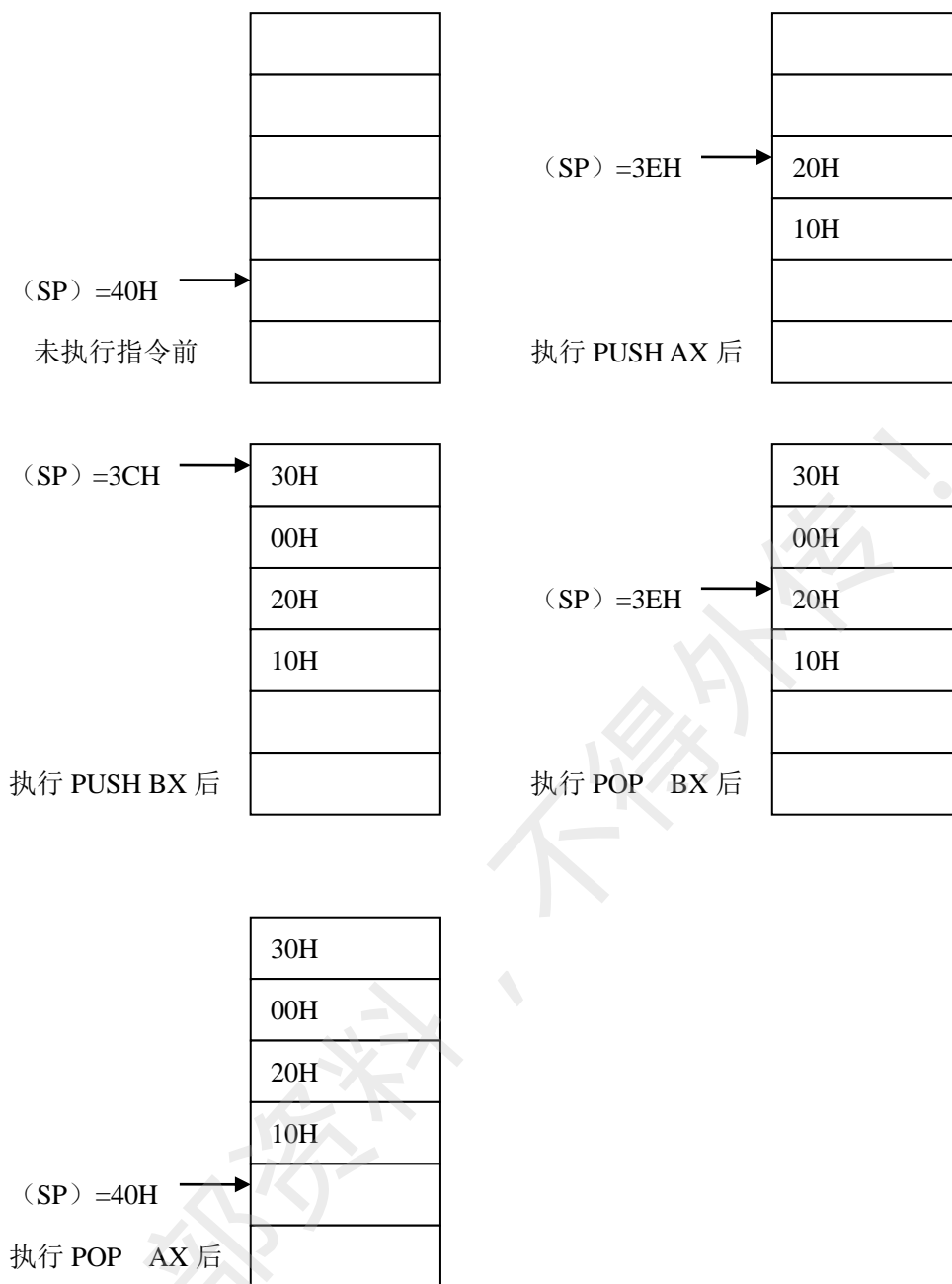
PUSH AX

PUSH BX

POP BX

POP AX

执行各条指令后，堆栈的变化如下图所示：（图中每个方框表示一个字节存储单元，内存单元地址按照由上至下方向递增）



在例 5.3.4 中，读者应注意体会入栈、出栈操作的后进先出原则，应注意到 **PUSH** 指令序列与 **POP** 指令序列的顺序必须相反，才能保证入、出栈的正确顺序。

#### (7) 标志压栈指令 (Push Flag)

指令格式: **PUSHF**



指令功能:  $(SP) - 2 \Rightarrow SP$   
 $(FR) \Rightarrow (SP)$

标志位影响: 无

**PUSHF** 指令先修改  $(SP)$ ，使之指向空栈顶，然后将 16 位标志寄存器中的数据传送到栈顶保存。**PUSHF** 指令为无操作数指令，但其中有两个隐含操作数，隐含的源操作数为标志寄存器，隐含的目的操作数为栈顶内存单元。在程序设计中，如果需要暂时保护标志寄存器，则可以使用 **PUSHF** 指令实现。

### (8) 标志出栈指令 (Pop Flag)

指令格式: **POPF**

指令功能:  $((SP)) \Rightarrow FR$   
 $(SP) + 2 \Rightarrow SP$

标志位影响: 全部标志位

**POPF** 指令的功能与 **PUSHF** 指令相反，它先将  $(SP)$  指示的栈顶字内存单元数据传送到 16 位标志寄存器保存，然后将  $(SP)$  加 2，以丢弃已出栈数据并修改栈顶位置。**POPF** 为无操作数指令，具有两个隐含操作数，源操作数为栈顶内存单元，目的操作数为 16 位标志寄存器。因为 16 位标志寄存器是 **POPF** 指令隐含的目的操作数，因此，所有标志位均受到该指令的影响。在程序设计中，如果需要将先前保存在堆栈中的标志位数据恢复到标志寄存器，那么可以使用 **POPF** 指令实现。

### (9) 装入有效地址指令 (Load Effective Address)

指令格式: **LEA DEST, SRC**

指令功能:  $EA_{SRC} \Rightarrow DEST$

标志位影响: 无

该指令将源操作数的有效地址传送至目的操作数保存。由于具有有效地址概念的存储单元只能是内存单元，因此该指令的源操作数只能是内存单元；并且，由于有效地址为 16 位，因此能够存储有效地址的目的操作数只能是 16 位存储单元，又由于双操作数指令中最多允许出现一个内存单元，所以，目的操作数只能是 16 位的通用寄存器。应注意该指令与 MOV 指令的区别，MOV 指令访问源操作数的数据，LEA 指令却是获取源操作数的有效地址，示例如例 5.3.5 所示。在程序设计中，如果需要获取某内存单元的有效地址，则可使用该指令实现。

**例 5.3.5** 假设 (DS) = 1000H，字内存单元 1000H: 0100H 中的数据为 0820H，试分析下面两条指令的执行结果。

MOV SI, [0100H]

LEA DI, [0100H]

执行第一条指令后，(SI) = 0820H，执行第二条指令后，(DI) = 0100H。

### (10) 装入地址指针指令

指令格式： LDS DEST, SRC

LES DEST, SRC

指令功能：  $(SRC) \Rightarrow DEST$   
 $(SRC + 2) \Rightarrow DS \text{ 或 } ES$

标志位影响：无

装入地址指令包括 LDS、LES 两条指令，源操作数只能是内存单元，目的操作数只能是 16 位通用寄存器，并隐含使用 DS 或 ES 段寄存器作为隐含目的操作数。该指令将源操作数地址指向的字内存单元数据传送至目的操作数保存，将源操作数地址加 2 指向的字内存单元数据传送至 DS 或 ES 保存。装入地址指针指令是将 SRC 源操作数地址指向的双字数据作为地址指针看待，低字解释为有效地址，高字解释为段基值，并将低字传送到 DEST 指定的 16 位通用寄存器保存，高字传送到 DS 或 ES 保存。与 LEA 指令不同，LDS 或 LES 是将内存单元中的数据解释为地址，而不是直接获取内存单元的有效地址，二者区别如例 5.3.6 所示。

**例 5.3.6** 假设  $(DS) = 1000H$ ，字内存单元  $1000H$ ：0100H 中的数据为 0820H，字内存单元  $1000H$ ：0102H 中的数据为 2000H，试分析下列指令的执行结果。

LEA BX, [0100H]

LDS BX, [0100H]

执行 LEA 指令后， $(BX) = 0100H$ ，执行 LDS 指令后， $(BX) = 0820H$ ， $(DS) = 2000H$ 。

实际上，在装入地址指针指令中，SRC 源操作数地址指向的双字数据与高级语言中的“指针”有相同的含义，如果在程序设计中需要装入指定的完整逻辑地址，可以使用指针变量与装入地址指针指令来实现。

## 5.3.2 算术运算类指令

在本小节中，我们介绍算术运算类指令中的加减法指令，总体上看，这些指令的功能是用于完成加法、减法两种算术运算，但具体用途各有差别。本小节中介绍的指令都会对状态标志位形成影响，读者应注意理解各标志位的含义，以及在什么样的前提下，标志位才能提供有效的标志信息。对初学者而言，标志位的理解是难点之一。

### (1) 加法指令 (Addition)

指令格式：ADD DEST, SRC

指令功能： $(SRC) + (DEST) \Rightarrow DEST$  或  $SRC + (DEST) \Rightarrow DEST$

标志位影响：所有状态标志，包括 OF、SF、ZF、AF、PF、CF

加法指令将源操作数与目的操作数相加，运算结果保存至目的操作数。源、目的操作数均可使用寄存器、内存单元，但至多允许出现一个内存单元；操作数的类型可以是字节，也可以是字；源操作数可以使用立即数，目的操作数不能使用立即数。加法指令的操作数实际上有三个，因为指令中目的操作数既用于提供原始数据，又用于保存运算结果，即它充当了源、目的操作数的双重角色。对标志位的理解如例 5.3.7 所示。

**例 5.3.7** 假设 (AL)=32H, (AH)=0F1H, 试分析执行如下加法指令后, 各寄存器的内容与各标志位的取值, 并对溢出情况加以分析。

ADD AL, AH

这里将操作数转换为二进制来分析其运算过程:

```

    00110010
+   11110001
-----
1 00100011

```

因此运算结果为 23H, 执行该指令后 (AL)=23H, (AH)=0F1H, 只有目的操作数发生改变。下面我们来分析各标志位在运算后的取值。

CF: 由于运算时最高位产生了进位, CF=1。

PF: 运算结果的低 8 位中共有 3 个为“1”的数据位, “1”数据位个数为奇数, PF=0。

AF: 由于运算时第 3 位 (最低位为第 0 位) 没有向第 4 位进位, AF=0。

ZF: 运算结果为非零编码, 因此 ZF=0。

SF: 运算结果最高位为“0”, 因此 SF=0。

OF: 如果操作数解释为补码, 则为“正+负”的情况, 不会出现溢出, OF=0。

下面我们再对溢出情况作出分析:

如果程序员将操作数解释为无符号数编码, 由于 CF=1, 无符号数运算超出了表达范围, 产生了溢出。

如果程序员将操作数解释为补码, 由于 OF=0, 补码运算没有溢出。

从上例中对运算溢出的分析, 我们可以体会到加法指令的二意性, 参加运算的操作数既可以解释为无符号数编码, 也可以解释为补码。换言之, 执行一条加法指令实际上同时完成了两种不同解释的运算, 即无符号数编码运算和补码运算, 两种运算过程的状态则由两套标志位生成电路分别记载于 CF 和 OF 标志。那么, 让人困惑的是, 运算中的编码在任意时刻都可以作两种理解吗? 回答是否定的, 每次对编码运算的解释只能取其中之一。加法指令对标志位的影响之所以如此设计, 是为了简化指令系统, 否则补码的加法运算和无符号数的加法

运算就要设计成两条不同的指令。但应当注意，编码的含义在任意时刻都是由程序员来解释的，程序设计人员应当非常清楚自己所使用的是无符号数还是补码，不同的解释导致标志位选用的不同。当程序员将操作数解释为无符号数编码时，OF、SF 等标志位就是没有价值的，因为它们是按照补码逻辑来解释的，不能为无符号数运算提供有效的标志信息；反之，当程序员将操作数解释为补码时，CF 标志就不同提供有效的标志信息。这样的标志位解释对于本小节所介绍的所有算术运算指令均适用。

## (2) 带进位加法指令 (Addition with Carry)

指令格式：ADC DEST, SRC

指令功能： $(SRC) + (DEST) + (CF) \Rightarrow DEST$  或  $SRC + (DEST) + (CF) \Rightarrow DEST$

标志位影响：所有状态标志，包括 OF、SF、ZF、AF、PF、CF

带进位加法指令将源操作数、目的操作数、进位标志 CF 相加后，运算结果保存至目的操作数。注意，运算时 CF 加在最低位。ADC 指令对操作数的限制与 ADD 指令一致，对标志位的解释也与 ADD 指令一致。与 ADD 指令不同的是，ADC 指令的功能主要是用于长操作数（长度超过机器字长限制）的加法，它可以将低字（或低字节）数据加法产生的进位反映到高字（或高字节）的加法中，以保证长操作数加法的连贯性、正确性，如例 5.3.8 所示。

**例 5.3.8** 假设  $(AX) = 08E7H$ ， $(BX) = 0485H$ ，现在要求完成两个操作数的加法，但为了体现长操作数的运算特征，我们把加法过程拆分为低字节和高字节两个加法操作，分别使用 ADD 指令和 ADC 指令来完成。试完成长操作数的加法，并分析无符号数编码、补码运算的溢出情况。

实现长操作数加法的指令序列如下：

ADD AL, BL

ADC AH, BH

低字节加法过程如下：

```

  11100111
+
 10000101
-----
1 01101100

```

由于最高位产生进位，CF=1

由于“负+负=正”，补码运算溢出，OF=1

高字节加法过程如下：

```

  00001000
+ 00000100
-----
  00000001
0 00001101

```

由于最高位没有产生进位，CF=0

由于“正+正=正”，补码运算没有溢出，OF=0

溢出分析：

从上面 CF、OF 标志的判断我们可以看到，低字节运算与高字节运算得出了截然相反的结论，那么，哪一次判断是正确的呢？无论程序员将操作数看作无符号数编码或补码，低字节部分的加法都是一次局部运算，它并没有完成整个长操作数的加法运算，如果理解为补码，低字节的最高位也不是符号位，因为它在长操作数的正中间。因此，使用低字节运算得到的标志位并不能用于溢出判断，正确的判断应在整个加法过程完成之后才能进行，换言之，只有在高字节加法完成后，分别使用 CF、OF 对无符号数、补码的溢出进行判断才是正确的。在本例中，无论作为无符号数编码或补码，运算都没有溢出。但低字节运算的 CF 标志是有价值的，它将 ADD、ADC 指令的执行衔接起来，将低字节的进位反映到了高字节的运算中，使整个长操作数的运算保持了连贯性、正确性。

上例说明的问题是非常重要的，如果我们要完成长操作数运算，实际长度可能比例中给出的 16 位要长出很多，我们通常通过 ADC 指令配合程序的循环结构来实现长操作数加法，但应当注意，仅在最高字（或最高字节）完成加法后，溢出判断才是正确。另外，CF 标志作为衔接长操作数高、低字（字节）间进位的必要工具，其意义是深远的。正因为有了 CF 标志，加法（包括后面介绍的减法）运算才能突破机器字长的限制，在理论上使任意长度的操作数都能在有限字长下完成运算。对于 CF 标志位的此重要用途，读者可以做任意机型推广，无论什么样的 CPU，都应有这样的机制来支持长操作数的加（减）运算。

### (3) 加 1 指令 (Increase)

指令格式: INC DEST

指令功能:  $(DEST) + 1 \Rightarrow DEST$

标志位影响: 五个状态标志, 包括 OF、SF、ZF、AF、PF

INC 指令将目的操作数加 1 后保存回目的操作数地址, 目的操作数兼作源、目的操作数两种角色。对操作数的限制符合一般单操作数指令的限制, 即不能使用立即数, 可以使用字或字节类型的通用寄存器、内存单元。读者应在程序设计时注意, INC 指令并不会影响 CF 标志。INC 指令的用途一般为循环结构中进行计数, 或在循环结构中修改用于表示数组下标的存储单元, 使程序能够逐个访问数组中的元素。

### (4) 减法指令 (Subtraction)

指令格式: SUB DEST, SRC

指令功能:  $(DEST) - (SRC) \Rightarrow DEST$  或  $(DEST) - SRC \Rightarrow DEST$

标志位影响: 所有状态标志, 包括 OF、SF、ZF、AF、PF、CF

减法指令将目的操作数作为被减数, 将源操作数作为减数, 完成减法操作后, 运算结果保存至目的操作数。减法指令对操作数的限制与加法指令相同。与加法指令一样, 减法指令本身也是具有二意性的, 究竟是完成的是无符号数编码运算还是补码运算, 需要由程序员来确定, 程序员确定编码的具体方式就是选择与编码对应的标志位作为后续的判断条件。减法指令的示例如例 5.3.9 所示。

**例 5.3.9** 假设  $(AL) = 32H$ ,  $(AH) = 0F1H$ , 试分析执行如下减法指令后, 各寄存器的内容与 CF、OF 标志位的取值, 并对溢出情况加以分析。

SUB AL, AH

二进制编码的减法过程如下: (运算结果左边的“1”表示有借位产生)

```

    00110010
  - 11110001
  -----
1 01000001

```

减法指令执行后，(AL)=41H，(AH)=0F1H。

由于最高位产生借位，CF=1，无符号数编码运算溢出。

由于“正-负=正+正=正”，补码运算没有溢出，OF=0。

请读者注意上例中对 CF 标志的判断，此判断是使用减法操作来实施的。在第一章我们介绍补码时曾经提到，多数现代计算机中的 CPU 内只有加法器，没有减法器，减法运算是通过补码加法来实现的。那么，上例中的减法运算也是先将减数变反加 1 后，再与被减数相加来实现，但是读者应注意理解，**CF 标志位的生成电路是基于无符号数运算逻辑的，与运算的实现过程无关**。因此，我们不能使用相应的补码加法来判断无符号数进位标志 CF，读者可以尝试将上例中的减法转换为等价的补码加法来实现，并判断标志位，一定会得出错误的判断结果。因此，在我们自己判断减法操作中的 CF 标志时，一定要采用减法来判断，这样才与 CPU 中 CF 生成电路的判断逻辑一致，**而不能采用补码加法来判断 CF 标志，因为采用带符号数的运算逻辑来判断无符号数标志位，在逻辑上本身就是混淆的**。这种混淆是初学者容易出现的理解错误，应在后续的学习中加以重视。

### (5) 带借位减法指令 (Subtraction with Borrow)

指令格式：SBB DEST, SRC

指令功能： $(DEST) - (SRC) - (CF) \Rightarrow DEST$  或  $(DEST) - SRC - (CF) \Rightarrow DEST$

标志位影响：所有状态标志，包括 OF、SF、ZF、AF、PF、CF

带借位减法指令与带进位加法指令含义相似，主要用于长操作数的减法过程，它能够将低字（字节）产生的借位反映到高字（字节）的减法操作中，使长操作数的减法过程保持连贯性、正确性。与长操作数的加法类似，对运算溢出的判断需要在最高字（字节）运算完成后才能进行，运算顺序也是由低字（字节）到高字（字节）进行。带借位减法指令 SBB 的操作数限制、标志位解释与减法指令 SUB 一致，同样需要注意，判断 CF 标志只能使用减法，不能使用补码加法。长操作数减法的示例请读者自己构造。



### (6) 减 1 指令 (Decrease)

指令格式: DEC DEST

指令功能:  $(DEST) - 1 \Rightarrow DEST$

标志位影响: 五个状态标志, 包括 OF、SF、ZF、AF、PF

减 1 指令将目的操作数减 1 后保存回目的操作数地址。与加 1 指令类似, 减 1 指令主要用于循环结构中的计数或数组下标修改。同样需要注意, DEC 指令不影响 CF 标志。

### (7) 求相反数指令 (Negative)

指令格式: NEG DEST

指令功能:  $0 - (DEST) \Rightarrow DEST$

标志位影响: 所有状态标志, 包括 OF、SF、ZF、AF、PF、CF

NEG 指令的操作数通常解释为补码, 其功能是求目的操作数中补码所对应的相反数补码, 运算结果保存至目的操作数地址。标志位解释与减法指令一致。NEG 指令的功能受机器字长的限制, 即操作数不能超过 16 位, 如果求长操作数的相反数, 则需要借助 SBB 指令来配合, 如例 5.3.10 所示。

**例 5.3.10** 假设 VA+2、VA 两个字内存单元中分别存放了一个 32 位补码的高字和低字, 要求设计程序片段获取该 32 补码的相反数补码。

NEG VA

MOV AX, 0

SBB AX, VA+2

MOV VA+2, AX

## (8) 比较指令 (Compare)

指令格式: `CMP DEST, SRC`

指令功能:  $(DEST) - (SRC)$  或  $(DEST) - SRC$

标志位影响: 所有状态标志, 包括 OF、SF、ZF、AF、PF、CF

CMP 指令除了不保存运算结果外, 其余所有解释与减法指令 SUB 相同。换言之, CMP 指令仅通过两个操作数的减法运算来影响标志位, 而对最终运算结果不保存。在程序设计中, 它通常用于程序员对两个操作数的大小关系感兴趣, 而对减法结果不感兴趣的情况。在数值运算程序的分支或循环结构设计中, 多数情况下需要使用 CMP 指令, 即 CMP 指令为判断两个操作数大小关系提供了必要的标志信息, 而条件转移指令可以根据此大小关系实现程序的分支或循环结构。那么, CMP 指令提供的标志信息如何指导我们判断两个操作数的大小关系呢? 另外, 如果运算出现溢出, 这些标志信息还能不能指导我们正确判断两个操作数的大小关系呢?

首先, 我们考虑无符号数编码的大小关系判断。如果两个无符号数编码相减,  $CF=0$ , 则说明最高位没有产生借位, 对应的情况为被减数大于等于减数; 如果  $CF=1$ , 则说明最高位产生了借位, 对应的情况为被减数小于减数, 而  $CF=1$  正是无符号数编码运算溢出的情况。换言之, 无论无符号数编码的减法运算有没有溢出, 我们都能根据 CF 的状态来判断两个操作数间的大小关系。如果结合 ZF 标志进行判断, 那么我们将能够得到更细致的判断结果, 读者可以自己考虑如何结合 CF 与 ZF 标志来判断两个无符号数编码大小关系。

然后, 我们考虑补码的大小关系判断。两个补码相减后, 我们可以根据 OF、SF 的不同取值分别进行分析:

1) **OF=0, SF=0:** 根据  $SF=0$  我们可以知道运算结果的最高位 (符号位) 为 0, 得到的运算结果应为一个正数补码 (包括 0),  $OF=0$  说明运算是没有溢出的, 那么此正数运算结果也就是正确的。这种情况对应于被减数大于等于减数的情况。

2) **OF=1, SF=1:** 根据  $SF=1$  我们可以知道运算结果的最高位 (符号位) 为 1, 得到的运算结果应为一个负数补码,  $OF=1$  说明运算出现了溢出, 那么此负数运算结果是错误的。因此,

我们可以推论，正确的运算结果必然是一个正数补码（包括 0），这种情况对应于**被减数大于等于减数**的情况。

3) **OF=0, SF=1**: 根据 SF=1 我们可以知道运算结果的最高位（符号位）为 1，得到的运算结果应为一个负数补码，OF=0 说明运算是没有溢出的，那么此负数运算结果也就是正确的。这种情况对应于**被减数小于减数**的情况。

4) **OF=1, SF=0**: 根据 SF=0 我们可以知道运算结果的最高位（符号位）为 0，得到的运算结果应为一个正数补码（包括 0），OF=1 说明运算出现了溢出，那么此正数运算结果是错误的。因此，我们可以推论，正确的运算结果必然是一个负数补码，这种情况对应于**被减数小于减数**的情况。

综上所述，无论补码运算有没有溢出，我们都可以通过  $SF = OF$  这一条件来得到被减数大于等于减数的结论，并通过  $SF \neq OF$  这一条件来得到被减数小于减数的结论。换言之，无论减法操作有没有溢出，我们都可以通过 OF、SF 这两个标志位的相等和不等关系来判断补码的大小关系。如果再结合 ZF 标志，我们可以得到更细致的判断结果。

在分支、循环程序设计一章我们将讨论条件转移指令，而条件转移指令正是根据 CF 的状态判断无符号数编码大小关系，根据 OF、SF 的状态来判断补码大小关系。在这里，我们应对这些标志位加深理解，作为理解条件转移指令的基础。

### 5. 3. 3 位操作类指令

位操作类指令是指能够以二进制位为单位来操作数据的指令，主要包括逻辑运算指令与移位指令。位操作指令的用途是多样化的，可以完成逻辑运算、算术运算，也可以对状态端口的各个状态位实施判断、对控制端口的各个控制位实施设置，也可以完成对数据位的各种拆分、组合操作。下面，我们将逐个介绍位操作类指令。

### (1) 逻辑与指令

指令格式: `AND DEST, SRC`

指令功能:  $(SRC) \wedge (DEST) \Rightarrow DEST$  或  $SRC \wedge (DEST) \Rightarrow DEST$

标志位影响: 所有状态标志, 包括 OF、SF、ZF、AF、PF、CF, 其中 CF、OF 被强行置为 0, AF 不确定, SF、ZF、PF 的解释与算术运算指令保持一致。

AND 指令将源操作数与目的操作数**按位相与**, 并将运算结果保存至目的操作数。注意, 标志位说明中, “**AF 不确定**”的含义是指 AND 指令会影响 AF, 但这种影响没有任何含义。程序员必须清楚的知道 AF 会被 AND 指令影响, 以免在程序设计中不知道 AF 的改变而导致设计中的逻辑错误, 但 AF 在 AND 指令中没有任何含义, 它对程序员的程序设计并不能提供任何帮助。读者应对这种被说明为“不确定”的标志位加以注意。这种不确定的标志位对于程序设计有害无益, 它们的产生是由于维护 CPU 中指令系统的规整性、维护指令译码器设计的简捷性而导致的, 关于这一点, 读者可以参阅计算机组成原理相关的教材。另外, 在 AND 指令中被强置为 0 的 CF、OF 也应引起读者的重视, 它们虽然没有实际应用价值, 但如果忽略它们的变化, 同样会引起程序设计中的逻辑错误。逻辑与指令除了完成通常的逻辑与运算外, 还可用于“**取位操作**”, 即将指定的二进制数位从字节或字数据中分离出来, 如例 5.3.11 所示。

**例 5.3.11** 假设  $(AL) = 01010110B$ , 要求将  $(AL)$  的第 1 位 (最低位为第 0 位) 分离出来。

`AND AL, 00000010B`

按位相与后,  $(AL) = 00000010B$ , 我们能够体会到,  $(AL)$  的第 1 位已经被分离出来, 其它数据位均被屏蔽。指令中立即数称为**取位模板**, 目的操作数中与取位模板中为“1”的位对应的数据位被取出, 而与取位模板中为“0”的位对应的数据位则被屏蔽。**这对我们在程序设计中分离状态端口中不同的状态位, 然后再分别加以判断提供了方便的手段。**

除“取位操作外”, 逻辑与指令还可以用于“**位清零操作**”, 即将指定的二进制位清零, 如例

5.3.12 所示。

**例 5. 3. 12** 假设 (AL) = 01010110B, 要求将 (AL) 的第 1 位 (最低位为第 0 位) 清零。

AND AL, 11111101B

执行按位相与后, (AL) 中除第 1 位被清零外, 其余数据位均不受影响。指令中的立即数可称为“清零模板”。

位清零操作可用于将控制端口中指定的控制位清零, 达到相应的控制目的。

## (2) 逻辑或指令

指令格式: OR DEST, SRC

指令功能:  $(SRC) \vee (DEST) \Rightarrow DEST$  或  $SRC \vee (DEST) \Rightarrow DEST$

标志位影响: 所有状态标志, 包括 OF、SF、ZF、AF、PF、CF, 其中 CF、OF 被强行置为 0, AF 不确定, SF、ZF、PF 的解释与算术运算指令保持一致。

OR 指令将源操作数与目的操作数按位相或后, 将运算结果保存至目的地址。其标志位的解释、注意事项与 AND 指令相同。逻辑或指令除了完成通常的逻辑或运算外, 还可用于“置位操作”, 如例 5.3.13 所示。

**例 5. 3. 13** 假设 (AL) = 01010110B, 要求将 (AL) 的第 0 位 (最低位为第 0 位) 置 1。

OR AL, 00000001B

执行按位相或后, (AL) = 01010111B, 除第 0 位被置 1 外, 其余数据位均不受影响。指令中的立即数可称为“置位模板”。

置位操作可用于将控制端口中指定的控制位置 1, 达到相应的控制目的。

### (3) 逻辑异或指令

指令格式: XOR DEST, SRC

指令功能:  $(SRC) \oplus (DEST) \Rightarrow DEST$  或  $SRC \oplus (DEST) \Rightarrow DEST$

标志位影响: 所有状态标志, 包括 OF、SF、ZF、AF、PF、CF, 其中 CF、OF 被强行置为 0, AF 不确定, SF、ZF、PF 的解释与算术运算指令保持一致。

异或操作的含义为: 如果两个相异或的数据位相等, 则异或结果为 0; 如果两个相异或的数据位不等, 则异或结果为 1。异或指令将源操作数与目的操作数按位异或后, 运算结果保存至目的操作数。其标志位的解释、注意事项与逻辑与指令相同。除了完成通常的异或运算外, 异或指令还可以用于“位变反操作”, 即将指定的二进制位变反, 如例 5.3.14 所示。

**例 5.3.14** 假设 (AL) = 01010110B, 要求将 (AL) 的第 7 位 (最低位为第 0 位) 变反。

XOR AL, 10000000B

执行按位异或后, (AL) = 11010110B, 除最高位变反外, 其余数据位均不受影响。指令中的立即数可称为“变反模板”。

与置位操作、位清零操作类似, 位变反操作可将控制端口中指定的控制位变反, 达到相应的控制目的。

### (4) 逻辑非指令

指令格式: NOT DEST

指令功能:  $\neg(DEST) \Rightarrow DEST$

标志位影响: 无

逻辑非指令将目的操作数中的数据位按位取反后, 保存回目的操作数地址。读者应注意, 与其它逻辑运算指令不同, NOT 指令不影响任何标志位。

## (5) 测试指令

指令格式: `TEST DEST, SRC`

指令功能:  $(SRC) \wedge (DEST)$  或  $SRC \wedge (DEST)$

标志位影响: 所有状态标志, 包括 OF、SF、ZF、AF、PF、CF, 其中 CF、OF 被强行置为 0, AF 不确定, SF、ZF、PF 的解释与算术运算指令保持一致。

除不保存运算结果外, TEST 指令的功能与标志位解释都和 AND 指令完全一致。TEST 指令与 AND 指令的对应关系, 与算术运算指令中 CMP 指令与 SUB 指令的对应关系非常类似。在程序设计中, 如果我们仅对逻辑与运算影响的标志位感兴趣, 而对具体的运算结果不感兴趣, 那么我们可以采用 TEST 指令。前面我们讲解 CMP 指令时曾经提到, CMP 指令常用于实现基于两个编码大小关系的分支或循环结构, 而这里的 TEST 指令的主要功能也是配合条件转移指令来实现分支或循环结构。TEST 指令主要通过取位操作对指定的单个二进制位进行分离, 并通过 ZF 标志来判断所分离二进制位的取值, 从而实现基于二进制位取值的分支或循环结构, 如例 5.3.15 所示。

**例 5. 3. 15** 假设  $(AL) = 01010110B$ , 要求测试  $(AL)$  的第 1 位 (最低位为第 0 位) 的取值。

`TEST AL, 00000010B`

执行按位相与后, 取位模板中为“0”的数据位必然导致运算结果相应数据位取值也为“0”, 由于取位模板中仅第 1 位为“1”, 因此, 运算结果中仅第 1 位的取值由  $(AL)$  的第 1 位决定。并且, 运算结果的第 1 位为“0”与否等价于整个运算结果为零与否。如果  $(AL)$  中第 1 位为 1, 那么逻辑与运算的运算结果非零,  $ZF=0$ ; 如果  $(AL)$  中第 1 位为 0, 那么逻辑与运算的运算结果为零,  $ZF=1$ 。如此, 我们就可以根据 ZF 的状态来判断  $(AL)$  中指定二进制位的取值, 并实现相应的分支或循环结构了。在本例中,  $ZF=0$ 。

TEST 指令通常可用于测试状态端口中指定状态位的状态, 并结合条件转移指令根据当前状态实现分支或循环处理; TEST 指令也可用于对寄存器、内存单元中的指定据位进行测试, 根据不同的取值作不同的处理, 具体示例请参见分支与循环程序设计一章的内容。

## (6) 算术左移指令

指令格式: SAL DEST, COUNT

COUNT 的含义为移位位数, 如果 COUNT 为 1 时直接用 “1” 表示, 如果 COUNT>1 则只能以 CL 寄存器表示, 以 (CL) 给出移位位数。

指令功能: 将 (DEST) 左移 COUNT 位, 从左边移出的最低一位保存到 CF 中, 右边空出的数据位则补充 0, 其含义为补码乘以  $2^{\text{COUNT}}$ 。

标志位影响: 所有状态标志, 包括 OF、SF、ZF、AF、PF、CF, 其中 AF 标志不确定, 其余标志位的解释与算术运算指令一致, 但仅当 COUNT=1 时, OF 标志才有意义。

SAL 指令实际上完成的是一种算术运算, 因为目的操作数每左移 1 位, 相当于乘以 2, 则将各二进制位的权值升高 2 倍, 左移 COUNT 位, 相当于乘以 COUNT 次 2, 则权值升高  $2^{\text{COUNT}}$  倍, 相当于乘以  $2^{\text{COUNT}}$ 。这种多次乘 2 的运算可以通过目的操作数不断自加来实现, 每次自加则相当于乘以 2, 因此, CF 标志仍可理解为目的操作数多次自加后的进位标志, 它可作为长操作数 (长度超过机器字长) 左移操作的衔接标志。**SAL 指令将目的操作数解释为补码**, 操作数的解释与算术运算指令不同, 它没有二意性, 完成的是补码乘以  $2^{\text{COUNT}}$  的操作, 可由 OF 来判断运算是否溢出。但应注意, 由于移位操作在硬件上是一次完成的, 即便 COUNT>1 时也是如此, 因此, 当 COUNT>1 时, 可能出现符号位多次变化在一次移位操作中发生的情况, 但 CPU 中的逻辑电路无法检测到这一情况, 最终导致 OF 仅在 COUNT=1 时才能提供有效的标志信息。如果 COUNT=1, 并且移位前后 (DEST) 的符号位发生了变化, 则 CPU 认为补码自加溢出, OF=1, 若符号位没有变化, 则认为没有溢出, OF=0。SAL 指令的示例如例 5.3.16、5.3.17 所示。

**例 5.3.16** (AL) = 11100010B, 将 (AL) 算术左移 1 位。

SAL AL, 1

移位前 (AL) 为 1 1 1 0 0 0 1 0



移位后 (AL) 为 1 1 0 0 0 1 0 0, (AL) 移位前的最高位移入 CF 中, 即 CF=1, 最低位被补充 0, 并且由于移位前后符号未变, 补码自加没有溢出, OF=0。

**例 5. 3. 17** (AL) =11100010B, 将 (AL) 算术左移 2 位。

MOV CL, 2

SAL AL, CL

移位前 (AL) 为 1 1 1 0 0 0 1 0

移位后 (AL) 为 1 0 0 0 1 0 0 0, (AL) 移位前的第 6 位移入 CF 中, 即 CF=1, 最低两位被补充 0, 由于移位位数大于 1, OF 标志无效。

#### (7) 算术右移指令

指令格式: SAR DEST, COUNT

COUNT 的含义为移位位数, 解释与 SAL 指令相同。

指令功能: 将 (DEST) 右移 COUNT 位, 从右边移出的最高一位保存到 CF 中, 左边空出的数据位则补充 (DEST) 的符号位, 其含义为补码除以  $2^{\text{COUNT}}$ 。

标志位影响: 所有状态标志, 包括 OF、SF、ZF、AF、PF、CF, 其中 AF 标志不确定, 其余标志位的解释与算术运算指令一致, OF 在算术右移中始终为 0, 参见例中解释。

与 SAL 指令的功能相反, SAR 指令的功能是将 (DEST) 解释为补码, 并将其除以  $2^{\text{COUNT}}$ 。移位后在左边空出的数据位补充符号位是为了保持补码的性质不被改变。与 SAL 指令类似, SAR 指令对操作数的解释没有二意性, 固定将 (DEST) 解释为补码。CF 标志用于衔接长操作数右移的高字 (字节) 和低字 (字节) 移位操作。SAR 指令的示例如例 5.3.18。

**例 5. 3. 18** (AL) =11100010B, 将 (AL) 算术右移 2 位。

MOV CL, 2

SAR AL, CL

移位前 (AL) 为 1 1 1 0 0 0 1 0

移位后 (AL) 为 1 1 1 1 1 0 0 0, (AL) 移位前的第 1 位移入 CF 中, 即 CF=1,

由于右移操作使补码真值的绝对值越来越小, 因此在算术右移中不可能出现溢出, OF=0。

(AL) 最高两位补充符号位, 在本例中符号位为 “1”。

### (8) 逻辑左移指令

指令格式: SHL DEST, COUNT

COUNT 的含义为移位位数, 解释与 SAL 指令相同。

指令功能: 将 (DEST) 左移 COUNT 位, 从左边移出的最低一位保存到 CF 中, 右边空出的数据位则补充 0, 其含义为补码乘以  $2^{\text{COUNT}}$ 。

标志位影响: 所有状态标志, 包括 OF、SF、ZF、AF、PF、CF, 其中 AF 标志不确定, 其余标志位的解释与算术运算指令一致, 但仅当 COUNT=1 时, OF 标志才有意义。

SHL 指令的功能与 SAL 指令相同, 都是完成 (DEST) 的左移操作, 只是 SHL 指令将操作数解释为无符号数编码。但实质上, SHL 与 SAL 指令是同一条机器指令, 因为无符号数编码的左移与补码的左移在本质上并没有区别, 只是程序员在判断溢出时选择的标志位不同而已, 判断无符号数自加溢出使用 CF 标志, 判断补码自加溢出使用 OF 标志。注意, 这里的解释显得比较晦涩, 因为如果 SHL、SAL 指令是同一条机器指令, 那么它们的操作数就具有二意性了, 但作为汇编指令来理解, SHL、SAL 是不同的指令, 它们分别将操作数解释为无符号数编码和补码, 这又不具有二意性, 那么哪种理解是正确的呢? 我们可以这样来回答, 两种理解都是正确的, 它们在机器指令系统中是同一条指令, 其操作数是具有二意性的, 但在汇编指令系统中, 它们被区分为不同的指令, 这是为了提醒程序员在程序设计时明确自己所使用的编码究竟是哪一种类型。SHL、SAL 指令也是在汇编语言中不多见的, 与机器指令之间违反一一对应关系的一个特例。逻辑左移的示例请读者自己构造。

### (9) 逻辑右移指令

指令格式: SHR DEST, COUNT

COUNT 的含义为移位位数, 解释与 SAL 指令相同。

指令功能: 将 (DEST) 右移 COUNT 位, 从右边移出的最高一位保存到 CF 中, 左边空出的数据位则补充 0, 其含义为无符号数编码除以  $2^{\text{COUNT}}$ 。

标志位影响: 所有状态标志, 包括 OF、SF、ZF、AF、PF、CF, 其中 AF 标志不确定, 其余标志位的解释与算术运算指令一致, 但仅当 COUNT=1 时, OF 标志才有意义。

SHR 指令的功能与 SHL 指令相反, 它将 (DEST) 解释为无符号数编码, 并将它除以  $2^{\text{COUNT}}$ 。应注意 SHR 指令与 SAR 指令的区别, 它们是不同的指令, SAR 指令在左侧补充符号位, 而 SHR 指令在左侧补充 0, 这正是因为 SHR 指令将操作数解释为无符号编码的缘故。逻辑右移指令仍然按照算术右移指令的规则来判断 OF 标志, 但由于其操作数解释为无符号数编码, 因此, **OF 标志在多数情况下没有应用价值**, 只能用于体现最高位在移位前后有无变化, 而不能作为符号变化来解释。逻辑右移的示例如例 5.3.19 所示。

**例 5. 3. 19** (AL) = 11100010B, 将 (AL) 逻辑右移 2 位。

MOV CL, 2

SHR AL, CL

移位前 (AL) 为 1 1 1 0 0 0 1 0

移位后 (AL) 为 0 0 1 1 1 0 0 0, (AL) 移位前的第 1 位被移入 CF, 即 CF=1,

移位位数大于 1, 因而 OF 无意义, (AL) 最高两位补充 0, 而不是补充符号位。

### (10) 循环左移指令

指令格式: ROL DEST, COUNT

COUNT 的含义为移位位数，解释与 SAL 指令相同。

指令功能：将 (DEST) 左移 COUNT 位，从左边移出的所有数据位按原有顺序补充到右边，从左边移出的最低一位保存到 CF 中。

标志位影响：CF、OF。CF 用于保存从左边移出的最低一位，OF 的判断规则与算术左移指令一致，当 COUNT=1 时，若移位前后 (DEST) 的最高位未发生变化，则 OF=0，若发生变化，OF=1；当 COUNT>1 时，OF 无意义。

循环左移指令不与任何算术、逻辑运算相对应，其用途主要为配合其它逻辑运算指令实现位操作。例如，我们在说明 TEST 指令时，曾提到 TEST 指令可以对状态端口的各状态位实现取位操作，并分别加以判断，如果配合使用循环移位指令对端口数据或取位模板加以循环移位，并在循环结构程序中实现，那么就能方便地对状态端口的各个状态位逐个实施检测了。循环左移指令的示例如例 5.3.20 所示。

**例 5. 3. 20** (AL) =11100010B，将 (AL) 循环左移 2 位。

MOV CL, 2

ROL AL, CL

移位前 (AL) 为 1 1 1 0 0 0 1 0

移位后 (AL) 为 1 0 0 0 1 0 1 1，(AL) 移位前的第 6 位被移入 CF，即 CF=1，由于移位位数大于 1，因此 OF 无意义。

### (11) 循环右移指令

指令格式：ROR DEST, COUNT

COUNT 的含义为移位位数，解释与 SAL 指令相同。

指令功能：将 (DEST) 右移 COUNT 位，从右边移出的所有数据位按原有顺序补充到左边，从右边移出的最高一位保存到 CF 中。

标志位影响：CF、OF。CF 用于保存从左边移出的最低一位，OF 的判断规则与算术左移指令一致，当 COUNT=1 时，若移位前后（DEST）的最高位未发生变化，则 OF=0，若发生变化，OF=1；当 COUNT>1 时，OF 无意义。

循环右移指令除移位方向与循环左移指令相反外，其余解释均与循环左移指令相同，其示例请读者自己构造。

### （12）带进位循环左移指令

指令格式：RCL DEST, COUNT

COUNT 的含义为移位位数，解释与 SAL 指令相同。

指令功能：将（CF）看作（DEST）的最高位，将（CF）、（DEST）构成的整体数据循环左移 COUNT 位。

标志位影响：CF、OF。CF 作为移位数据的最高位，OF 的判断规则与算术左移指令一致，当 COUNT=1 时，若移位前后（DEST）的最高位（指存储单元最高位，不是 CF）未发生变化，则 OF=0，若发生变化，OF=1；当 COUNT>1 时，OF 无意义。

带进位循环左移指令的用途主要是配合算术（逻辑）左移指令实现长无符号数编码或长补码（超过机器字长限制）的乘 2 操作，如例 5.3.21 所示。

**例 5. 3. 21** 假设一个 48 位的无符号数编码由低位到高位分别存放于 DA、DA+2、DA+4 这三个字内存单元（16 位）中。要求实现此 48 位无符号数编码乘以 2 的运算。

乘以 2 的运算即为左移 1 位，程序片段如下：

```
SHL DA, 1
```

```
RCL DA+2, 1
```

```
RCL DA+4, 1
```

读者可以从本例体会到，每次低地址字的最高位都移位至 CF 保存，下次移位时，CF 被移

位至高地址字的最低位，而高地址字的最高位又被移位至 CF 保存，如此继续，直到所有存储单元处理完毕。

从上例可以看到，CF 标志在长操作数移位中充当了衔接工具的作用，带进位循环移位指令通过 CF 标志将低位移位与高位移位贯穿起来，在原理上可实现任意长度操作数的移位操作。读者应注意，长操作数左移操作是由低位到高位顺序进行的，并且第一条指令使用 SHL (SAL) 指令，它是常规的左移指令，后续移位操作才使用 RCL 指令。由于无符号数编码与补码的左移操作没有本质区别，因此例 5.3.21 中说明的方法也可用于长补码左移（乘以 2）操作，不过一般习惯将第一条指令写作 SAL 形式，表明我们将操作数解释为补码。但是对长无符号数编码左移、长补码左移的溢出判断却是不同的，无符号数编码自加溢出判断使用 CF 标志，补码自加溢出判断使用 OF 标志。并且，无论是无符号数编码还是补码，只有当最高字（字节）完成左移操作后，即对长操作数的移位完全完成后，才能对移位是否溢出进行判断。读者还应当注意，对于长操作数的移位操作只能逐位进行，不能一次移动多位。原因有两个，第一，CF 每次只能容纳一个被移出的位，第二，同时移动多位将导致 OF 标志无意义，无法对补码移位的溢出进行判断。

### （13）带进位循环右移指令

指令格式：RCR DEST, COUNT

COUNT 的含义为移位位数，解释与 SAL 指令相同。

指令功能：将 (CF) 看作 (DEST) 的最低位，将 (CF)、(DEST) 构成的整体数据循环右移 COUNT 位。

标志位影响：CF、OF。CF 作为移位数据的最高位，OF 的判断规则与算术左移指令一致，当 COUNT=1 时，若移位前后 (DEST) 的最高位（指存储单元最高位，不是 CF）未发生变化，则 OF=0，若发生变化，OF=1；当 COUNT>1 时，OF 无意义。

与 RCL 指令相似，RCR 指令的用途是配合 SHR 与 SAR 指令分别实现长无符号数编码与长补码的右移（除以 2）操作。如例 5.3.22 所示。

**例 5.3.22** 假设一个 48 位的无符号数编码由低位到高位分别存放于 DA、DA+2、DA+4 这三个字内存单元（16 位）中。要求实现此 48 位无符号数编码除以 2 的运算。

除以 2 的运算即为右移 1 位，程序片段如下：

```
SHR DA+4, 1
```

```
RCR DA+2, 1
```

```
RCR DA, 1
```

读者应注意，长操作数的右移操作顺序是由高位到低位进行的，与左移操作顺序相反。

长无符号数编码右移时第一条指令使用 SHR 指令，长补码右移时第一条指令使用 SAR 指令，与左移操作不同，SHR 与 SAR 是不同的指令，因此在程序设计中一定要明确所使用的编码是无符号数还是补码。长操作数右移操作的其它注意事项与左移操作相似，请参见 RCL 指令的说明。

长操作数的移位操作与长操作数的加、减法操作非常类似。ADD 与 ADC 指令配合可实现长操作数的加法，SUB 与 SBB 指令配合可实现长操作数的减法，长操作数的加、减法都是由低位到高位顺序进行。SHL（SAL）与 RCL 指令配合可实现长操作数左移操作，操作顺序是由低位到高位，SHR 或 SAR 指令与 RCR 指令配合可实现长操作数右移操作，操作顺序是由高位到低位。在长操作数的运算操作中，有一点是共通的，即判断无符号数运算是否溢出使用 CF 标志，判断补码运算是否溢出使用 OF 标志，并且仅当整个长操作数运算结束后，才能对溢出进行正确判断。

### 5.3.4 处理器控制类指令

处理器控制类指令是指能够改变 CPU 工作模式或改变 CPU 对某些特定事件处理方式的指令，但同时也包括了一些其它指令类别中未包含的指令。下面我们将逐条介绍这些指令的功能。

### (1) 进位标志清除指令 (Clear carry flag)

指令格式: CLC

指令功能: 清除 CF 标志, 使 CF=0

标志位影响: CF

该指令最容易理解的用途为配合 ADC 指令实现长操作数加法。进入程序的循环结构前将 CF 清零, 循环结构中仅使用 ADC 指令完成长操作数各字 (字节) 的带进位加法, 但第一次执行循环时, 由于 CF=0, 则 ADC 指令等同于 ADD 指令, 如此可省略对 ADD 指令的使用, 简化程序结构。除此之外, CLC 指令还可用于其它场合, 例如配合带进位循环移位指令, 对存储单元中的数据位进行逐位清零操作。在将来的程序设计学习中, 请读者逐步去体会其用途。

### (2) 进位标志置位指令 (Set carry flag)

指令格式: STC

指令功能: 置位 CF 标志, 使 CF=1

标志位影响: CF

### (3) 进位标志取反指令 (Complement carry flag)

指令格式: CMC

指令功能: 将 CF 标志取反

标志位影响: CF

### (4) 方向标志清除指令 (Clear direction flag)

指令格式: CLD

指令功能: 清除 DF 标志, 使 DF=0

标志位影响: DF



当 CPU 执行串操作指令时，会按照地址递增的方向来修改变址寄存器，在非数值处理程序设计一章中，请读者结合串操作指令来理解。

#### **(5) 方向标志置位指令 (Set direction flag)**

指令格式: STD

指令功能: 置位 DF 标志, 使 DF=1

标志位影响: DF

当 CPU 执行串操作指令时，会按照地址递减的方向来修改变址寄存器，在非数值处理程序设计一章中，请读者结合串操作指令来理解。

#### **(6) 中断使能标志清除指令 (Clear interrupt-enable flag)**

指令格式: CLI

指令功能: 清除 IF 标志, 使 IF=0

标志位影响: IF

关闭 CPU 内部的可屏蔽中断响应开关, 当设备接口向 CPU 提出可屏蔽中断请求时, CPU 不会响应。

#### **(7) 中断使能标志置位指令 (Set interrupt-enable flag)**

指令格式: STI

指令功能: 置位 IF 标志, 使 IF=1

标志位影响: IF

开启 CPU 内部的可屏蔽中断响应开关, 当设备接口向 CPU 提出可屏蔽中断请求时, CPU 会做出响应。

### (8) 停机指令 (Halt)

指令格式: HLT

指令功能: 切断 CPU 的时钟输入, 系统停机

标志位影响: 无

该指令执行后, CPU 的时钟信号输入被切断, CPU 无法继续执行所有时序过程, 处于“休克”状态, 系统停机。执行 HLT 指令后, 仅能通过 RESET 复位信号使系统重新启动, 才能恢复 CPU 的运转。

### (9) 空操作指令 (No operation)

指令格式: NOP

指令功能: 无操作, CPU 空转 3 个节拍

标志位影响: 无

该指令不执行任何有意义的操作, 仅使 CPU 空转 3 个节拍。类似于数字“0”, NOP 看起来没什么实际价值, 但它却很重要。在调试可执行程序时, 如果调试人员想要屏蔽某些机器指令功能, 那么使用 NOP 指令最合适, 因为它是单字节的机器指令, 任意长度的机器指令都可以由它来填充并屏蔽, 这是 NOP 指令最大的应用价值。除此之外, 有时也可以使用多条 NOP 指令来进行延时, 例如, 我们启动了某个外部设备, 但它要等一小段时间才会从数据端口返回数据, 那么我们可以适当使用 NOP 指令来延时等待。当然, NOP 指令的延时功能具有较差的移植性, 因为它实现的延时是以节拍为单位的, 不同型号的 X86 芯片, 其节拍长度自然不同。如果读者需要精确延时, 建议使用定时器中断相关的系统调用。

## 习题 5

1. 试说明以下指令中各操作数的寻址方式，如果是存储器寻址，请给出其 EA 计算公式，并说明所使用的段寄存器。

- (1) MOV AL, 08H ;注意立即数寻址和直接寻址的区别
- (2) MOV [0120H], BL
- (3) ADD [BX], AL
- (4) PUSH [SI]0200H
- (5) SUB AX, [BP] ;BP 的默认段寄存器是 SS
- (6) AND VAR1+4, DL ;符号地址，本质上是[XXXXH+4]
- (7) PUSHF
- (8) MOV ES: [BX]0100H, AL ;这是段超越，本质上是“段基址强制”
- (9) ADC BYTE PTR [BP][SI]0210H, 45H
- (10) OR ARRAY[BX][DI], CL ;以所使用的基址寄存器为准，确定使用 DS 或 SS

2. 试分析下列汇编指令是否存在语法错误，如果有语法错误存在，请说明是怎样的错误。

- (1) PUSH 8243H
- (2) POP AL
- (3) MOV AL, 6543H
- (4) ADD [0100H], 64H
- (5) ADC VAR1, VAR2
- (6) MOV DS, ES
- (7) MOV DS, 0620H
- (8) LEA BX, AX
- (9) DEC AL, AH
- (10) SHR BL, 3

3. 试说明分别执行下列各组指令后，CF、OF、AF、ZF、SF、PF 这六个状态标志分别是怎样的取值。

- (1) MOV AL, 08H

ADD AL, 0F9H

(2) MOV AL, 0E1H

ADD AL, 0F4H

(3) MOV AL, 01H

SUB AL, 02H

(4) MOV AL, 02H

INC AL

(5) MOV AL, 01H

AND AL, 02H

4. 按要求分析下面程序片段的执行结果。

MOV AL, 0C2H

MOV AH, 0E4H

ADD AL, AH

执行该程序片段后，(AL) = ? ，(AH) = ? ，如果将 ADD 指令的两个操作数解释为无符号数，运算有没有溢出？为什么？如果将 ADD 指令的两个操作数解释为补码，运算有没有溢出？为什么？

5. 按要求分析下面程序片段的执行结果。

MOV AL, 98H

MOV BL, 42H

XCHG AL, BL

SUB AL, BL

执行该程序片段后，(AL) = ? ，(BL) = ? ，如果将 SUB 指令的两个操作数解释为无符号数，运算有没有溢出？为什么？如果将 SUB 指令的两个操作数解释为补码，运算有没有溢出？为什么？如果将 SUB 指令的两个操作数解释为补码，其减法运算对应的十进制真值表达式应如何书写？

6. 按要求分析下面程序片段的执行结果。

STC

```
MOV AL, 03H
```

```
AND AL, 02H
```

```
ADC AL, 00H
```

执行该程序片段后，(AL) = ?

7. 假设 (DS) = 1000H, (SS) = 2000H, 字内存单元 (10200H) = 0870H, (10202H) = 2000H, (20870H) = 0203H, (20872H) = 0405H, 括号内所给为内存单元物理地址, 括号表示该地址所指示单元中保存的数据, 分别执行下列程序片段后, 按要求分析各程序片段的执行结果。

(1) MOV AL, [0200H]

执行该程序片段后, (AL) = ?

(2) MOV BP, 0871H

```
MOV BL, [BP]
```

执行该程序片段后, (BL) = ?

(3) LEA SI, [0200H]

执行该程序片段后, (SI) = ?

(4) MOV SI, [0200H]

```
LEA SI, [SI]
```

执行该程序片段后, (SI) = ?

(5) LDS BX, [0200H]

```
MOV AL, [BX]0002H
```

执行该程序片段后, (AL) = ?

8. 按要求分析下面程序片段的执行结果。

```
MOV AX, 651CH
```

```
SHL AL, 1
```

```
RCL AH, 1
```

执行该程序片段后, (AX) = ? , 该程序片段的功能是什么? 如果将 (AX) 解释为无符号数, 那么运算是否溢出? 为什么? 如果将 (AX) 解释为补码, 运算是否溢出? 为什么? SHL 与 SAL 指令间有什么关联和区别?

9. 按要求分析下面程序片段的执行结果。

```
MOV AL, 35H
```

```
AND AL, 0FH
```

执行该程序片段后, (AL) =? CF、OF、AF、ZF、SF、PF 标志取值是什么? 该程序片段的功能是什么?

10. 假设一个 48 位的补码按照由低位到高位顺序保存在字类型的内存单元 VA1、VA1+2、VA1+3 中, 试按下列要求完成程序片段设计。

(1) 设计程序片段, 实现将该 48 位补码除以 4 的功能, 运算结果仍然保存在原内存单元中。

(2) 设计程序片段, 求该 48 位补码的相反数补码, 运算结果仍然保存在原内存单元中。

11. 试说明如何使用 CMP 指令提供的标志位判断两个补码操作数大小关系的原理。

12. 假设 (SP) = 0060H, 执行两次 PUSH 指令后, (SP) =? 假设 (SP) = 0038H, 执行三次 POP 指令后, (SP) =?

13. 按要求分析下面程序片段的执行结果。

```
MOV AL, 01H
```

```
NEG AL
```

```
INC AL
```

执行该程序片段后 (AL) =?, CF、OF 标志的状态是什么?

14. 按要求分析下面程序片段的执行结果。

```
MOV BL, 51H
```

```
AND BL, 0FEH
```

```
XOR BL, 50H
```

```
DEC BL
```

执行该程序片段后 (BL) =?, CF、OF 标志的状态是什么?

15. 按照各小题的要求分别设计程序片段。

(1) 将 AL 寄存器的高 4 位与低 4 位交换

(2) 将 TF 标志位置 1

(3) 将 AL 寄存器的第 7 位清 0, 但不影响其它数据位

(4) 分离 AL 寄存器的最低两位, 其它数据位清 0。

(5) 分离 AL 寄存器的高 4 位与低 4 位, 并分别保存在 BL、BH 的低 4 位

## 第 6 章 汇编语言源程序组织

前面若干章，我们用了不小的篇幅介绍了 8088/8086 CPU 的寄存器和访问内存的方法以及基本的汇编语言指令；又通过上一章最后一节介绍的 DEBUG 调试工具，我们已经可以观察寄存器和存储单元的内容，并能实践单条指令甚至编写一些短程序了。不过所有汇编语言的指令还不是汇编语言的全部，完备的编程语言还必须包含常量和变量定义、程序结构描述、程序编译控制等——这些是通过汇编语言的“伪指令”来实现的。本章从汇编语言的语句开始，将较为完整地介绍的数据类型、常量与变量、表达式、段定义语句等汇编语言的基本元素和语法规则，最后以汇编语言源程序的一般框架作为小结。

尽管汇编语言是符号化的机器语言，但也不能被计算机直接识别和执行，也必须和其他编程语言一样经过编译（习惯上叫“汇编”）才能形成计算机能够识别的目标代码。在本章的最后一节，我们将介绍汇编语言编辑、编译（汇编）、连接和调试的基本过程和汇编语言工具基本用法。

### 6.1 汇编语言的语句种类和格式

和其他编程语言一样，汇编语言的程序也是由语句组成的。汇编语言的语句分以指令助记符为核心的“指令语句”和以“伪指令”为核心的“伪指令语句”。汇编语言指令我们已经不陌生了，故伪指令是这一章介绍的重点。

#### 6.1.1 指令语句

我们先看看指令语句的一般格式：

标号： 指令助记符 目标操作数表达式，源操作数表达式 ;注释

除操作数间用逗号分开外，语句各字段间至少要用一个空格作为分隔符；另外除注释或字符串中的内容外，语句中用到的标点符号、空格和字符等都是 ASCII 字符。

相对我们前面接触过的指令语句，上述指令语句的通用格式似乎有些繁琐，但由于标号和注释字段是可选的，掐头去尾后我们就很熟悉了。下面通过一个字段完整的指令语句对每个字段加以说明：

**Next:   MOV    al,[si]   ;取出一个字节的内容送 al**

其中，标号是指令语句的符号地址。其实源程序经编译（汇编）后，每条指令语句的代码都会按先后顺序自然包含程序运行时在代码段的存放位置，即每条指令语句都有各自的地址，而标号是给要使用这条语句地址的代码作了一个标记，程序编译（汇编）时会将其他使用到这个符号地址的语句中的这个标号替换成相应的地址。比如在同一代码段内其他位置有一条跳转指令：

**Jmp     Next**

编译（汇编）后这条指令语句的操作数“**Next**”将被替换成定义 **Next** 的那条语句在代码段的偏移量；执行到这条语句时，程序将跳转至定义 **Next** 的那条语句继续执行。

需要明确的是，无论一条指令语句是否有标号，编译后的代码是没有区别的。尽管带有标号的指令语句的源代码多了一个字段的内容，但编译出的目标代码和没有语句标号是一样的，不会多占存储空间，标号在这条语句中只是给这句代码作了一个标记供编译使用。

指令字段就无须多说了，只能是汇编语言的助记符；操作数字段除要求必须遵循操作数的寻址方式，本章后面还将介绍一些用于操作数的伪指令前缀。

注释对汇编语言编程尤为重要。这是因为汇编语言描述的是机器的动作，相对面向问题的高级语言，代码和其处理问题之间的联系更不直白、明显，所以必要的注释对汇编语言的编程和维护就比其他高级语言更为重要；另一方面，汇编语言编译时的语法检查将忽略源代码文本每一行分号之后的所有内容，即在分号之后，我们可以用英语、中文等自然语言描述语句或语句段的功能而不受汇编语言语法的约束。



## 6.1.2 伪指令语句

我们先介绍伪指令的基本概念。

伪指令是用于指示编译（汇编）程序如何编译（汇编）源程序的，例如描述源程序如何分段和各段分别由哪个段寄存器指向，以及常量和变量及子程序的定义等。

其次谈谈伪指令与指令的不同：伪指令是用于控制源程序编译（汇编）而不是控制 CPU 运行的；伪指令是在源程序编译时被执行的，而指令是在程序运行时被执行的。

下面是伪指令语句的一般格式：

符号名 伪指令 操作数表达式 1,操作数表达式 2,...,操作数表达式 n ;注释

和指令语句在形式上的区别除语句核心字段是伪指令而不是指令外，符号名后没有冒号也是明显和重要的区别特征。

语句中的“符号名”是常量、变量、子程序或段结构等的命名。

伪指令是语句的核心，用于指示编译（汇编）程序要完成的编译操作。

每条伪指令语句可以没有或多个操作数；操作数可以是常量、变量名、表达式等；操作数的功用在后面结合例子说明。

分隔符及注释和指令语句的语法要求是一致的。

这里我们对伪指令的上述描述有个字面上的印象就行了，通过后面各种伪指令的逐条介绍我们才会对伪指令的功能及其和指令的区别有准确的了解。

## 6.1.3 标识符

在汇编语言中，指令语句的标号和伪指令语句的符号名被统称为标识符。下面是标识符的组成规则：

1. 标识符是长度不超过 31 的字符串；
2. 标识符的字符集是英文字母、数字和下述特殊字符：

? @ \_ . \$

的 ASCII 码，但第一个字符不能是数字。

3. 标识符不能和汇编语言的保留字冲突。即不能和寄存器、指令、伪指令等同名。

尽管汇编语言在语法上对标识符要求只有上述 3 点，但和其他编程语言的符号命名一样，标识符应尽量体现代码的功能或被命名对象的含义，这样对提高程序可读性的意义是显然的。

## 6.2 常量与变量

无论使用何种计算机编程语言，也无论是什么样的应用，计算机总是通过处理数据来实现程序功能的。本节将介绍汇编语言的数据类型及其两种存在形式：常量和变量。

### 6.2.1 常量

8088/8086 汇编语言常量是数字常量或字符串常量；编程语言所指的常量是：在编译完成时被确定，在程序运行过程中始终不变。

#### 1. 数字常量

数字运算和数字处理是计算机最基本的功能，这是任何编程语言必须首先面对的问题。在汇编语言的源程序中，数字常量允许用各种进制表示。不过需要明确的是，这并不是计算机能够直接识别它们，而是源程序在通过编译（汇编）时，这些各种进制的数被转换成了二进制数。

- ① 十进制数：以字母 D 结尾（也可以忽略）的 0~9 组成的数字序列。
- ② 二进制数：以字母 B 结尾的 0 和 1 组成的数字序列。如 01001101B。
- ③ 十六进制数：以字母 H 结尾的 0~9 及 A~F 组成的序列（这里 A~F 对应 10 进制数 10~15）；当十六进制数的高位是 A~F 的某个字母时，为了避免与标识符混淆，规定在这种情况下十六进制数前面必须加一“0”作为区别。如：0F4H。
- ④ 八进制数：以字母 O 结尾的 0~7 的数字序列。
- ⑤ 浮点数。如：12.34E-6（表示  $12.34 \times 10^{-6}$ ）。

八进制数和浮点数至少在汇编语言学习中可以基本不用。汇编语言最常用的数据形式还是人习惯使用的十进制数和机器可直接识别的二进制数；而十六进制数和二进制数的关系简单直接，且较二进制数识别容易，书写简捷，所以在讨论二进制数时实际上大都是以十六进制数来进行的。

## 2. 字符串常量

处理字符和字符串的能力是计算机文字处理软件的重要基础。我们后面会看到，在 8088/8086 CPU 的指令集中，专门设计了用于提高文字处理能力的字符串操作指令。

字符串常量由单引号或双引号括起来的 1 个或多个字符组成，如：

'Hello, World!'

'G'

编译（汇编）后每个字符被翻译成对应的 ASCII 码，每个字符占一个字节。如 'ABC' 编译（汇编）后所占 3 个字节的内容是 41h、42h、43H。

常量的用途举例：

	ADD	AL,6	;用作立即数
	MOV	AH,[100H]	;用作直接寻址方式中的偏移量
	MOV	AL,[BX+20]	;用作基址或变址寻址方式中的位移量
X	DB	20H	;给变量 X 赋初值

## 3. 符号常量定义伪指令

8088/8086 汇编语言中有两条符号常量定义伪指令。使用符号常量，可方便程序维护，提高代码的可读性。

### ① 等值伪指令 equ

Pi equ 3.1415926

这是我们第一次正式接触到的伪指令语句。这条语句给编译（汇编）程序的指示是：将源程序中所有的 **Pi** 都用 3.1415926 替代。

## ② 等号伪指令 “=”

简单地将 “**equ**” 改写成 “=”：

```
Pi      =      3.1415926
```

亦将 “**Pi**” 定义成了 3.1415926。该指令一样指示编译（汇编）程序将源程序中有出现 **Pi** 的地方都用 3.1415926 替代。“=” 伪指令和 “**equ**” 的差别是：后者可以对同一常量作多次定义，比如在程序的另一处可以：

```
Pi      =      3.1416
```

即此后遇到 **Pi** 的地方都被编译成 3.1416 而不是 3.1415926。需要说明的是，尽管这条伪指令有点灵活性，但笔者认为这样做会增加编程时为避免混乱的额外负担，所以建议定义常量就使用 **equ** 伪指令即可。

## 6.2.2 简单变量定义

变量在人工或计算机数据处理中的作用是不可或缺的。

计算机存储单元的内容是可以被读取或被修改的，所以编程语言用符号命名的若干存储单元来表示一个变量。汇编语言中，常见的数据类型有字节、字和双字等。

### 1. 变量定义的一般形式

变量定义语句的一般格式是：

```
[变量名]    数据定义符    表达式 1[, 表达式 2, ..., 表达式 n]    [;注释]
```

其中：

- ① 变量名必须是一个合法的标识符；
- ② 数据定义符用于确定变量的数据类型，常用的定义符有：DB、DW 和 DD 等；
- ③ 表达式用于定义内存单元的初值，一个定义语句可以有多个表达式，各表达式之间必须用西文逗号‘,’分开；如果某个存储单元没有初值表达式，则必须用一个西文问号‘?’来替代；
- ④ 变量定义语句的注释根据具体需要，可写可不写。

其实变量名就是某个或某若干个字节存储单元的符号地址，这和指令语句的标号很相似：变量名是所描述的变量在数据段的偏移量，这是汇编程序在编译（汇编）时，将操作数第一个字节的偏移地址赋给了这个符号。

汇编语言允许直接使用数字偏移量访问内存单元，我们在前面使用 DEBUG 的“A”命令编写的小程序中就这样用过，但这样做比起使用变量名有两个缺点：首先代码可读性的差距是明显的；直接使用偏移量的第二个问题是当偏移量调整时，所有直接引用偏移量的地方都必须修改，这显然增加了程序的维护量和出错的机率。而使用变量名，偏移量的变化会在程序编译时自动修正。

## 2. 定义字节变量

字节变量的定义符为 DB，如：

```
X    DB    0
```

该语句定义 X 为字节型变量，分配给 X 一个字节，且初值为零。这条伪指令语句和下面 C 语言的变量定义语句一样的：

```
char    x = 0;
```

如果：

```
X    DB    ?
```

这里的西文问号表示没有给 X 赋初值，但分配一个字节的空間。这和 C 语言的变量定义语句：

```
char    x;
```

是一样的。

字节型变量常用作小值域数值变量或单字节字符（如 ASCII 码）变量。

下面的伪指令语句是使用多个表达式的例子：

```
ODD_TABLE DB 1,3,5,7,9
```

```
HEX_TABLE DB '0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F'
```

前一语句为字节型变量 ODD\_TABLE 分配了 5 个字节并依序被初始化成了 10 以内的奇数；后一语句定义了一个 16 进制数的 ASCII 码表。

DB 伪指令还支持用“字符串”表达式定义变量初值，如：

```
MSGHI    DB    'Hello,World!'
```

其实，前面提到的“16 进制数的 ASCII 码表”也可以这样定义：

```
HEX_TABLE DB '0123456789ABCDEF'
```

这样显然简洁得多。

### 3. 字变量定义

字变量的定义符为 DW；每个字占用两个连续的字节单元。如：

```
WVAR    DW    2008H
```

上述定义初始化后的内存分配如下所示：

...
08H
20H
...

注意字变量数据是按“高位存放到高地址字节、低位存放在低地址字节”的方式存于存储单元之中的，而字节数据是按照顺序排列在存储单元中的，比如：

```
BVAR  DB    20H,08H
```

数据在内存里是这样存放的：

...
20H
08H
...

使用时除注意不要弄错了之外，也不要把字类型变量或下面将要介绍的双字变量用于存放 8 位字符串数据。

#### 4. 双字变量

双字变量的定义符为 DD，每个双字变量占用二个连续的字单元共四个字节。

例如：

```
DDVAR  DD    12345678H
```

上述定义的内存分配右图所示。

字和双字类型变量多用作存储较大值域的数据，不适合用作存储 8 位字符串。

...
78H
56 H
34 H
12 H
...

## 5. 重复说明符 DUP

我们知道 C 语言是这样定义数组的：

```
char x[100];
```

而汇编语言用“重复说明符”DUP 来定义同类型变量：

```
BVAR    100  DUP  (?)
```

带重复说明符 DUP 的表达式的一般形式是：

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

其中：**count** 是重复次数，括号内的若干表达式是被重复的部分；“表达式”可以直接是存储单元的初值，也可以是另一个被嵌套的 DUP 表达式；表达式括号中的多个表达式用西文逗号“,” 分开。下面是一些使用 DUP 的例子：

```
WVAR    DW    50  DUP  (?)           ;定义了 50 个字单元变量
STRHI    DB    20  DUP  ('Good!')    ;定义了 100 个字节的字节型变量并在这
                                           ;100 个字节的存放了 20 个'Good!'。
```

顺便提及：变量定义表达式中的问号，在字节、字及双字变量中分别代表 1、2、4 个字节，即各种类型变量的一个单元。

## 6.2.3 标号和内存变量的属性及属性操作符

尽管标号和变量有很多不一样的东西，但它们都是符号地址，代表一个存储单元的地址，所以，它们都具有存储单元的属性：段属性、偏移量属性等。除此之外，它们还有各自的特殊属性。

下面介绍标号和内存变量的属性及相关运算符。



### 1. 段属性运算符 SEG

段属性运算符 **SEG** 指示编译程序返回变量或标号操作数所在段的段基址。如在同一数据段定义了如下两个变量：

```
BVAR  DB    200 DUP(?)
WVAR  DW    100 DUP(0)
```

而下述代码：

```
MOV    SI,SEG BVAR
MOV    DI,SEG WVAR
```

因为 **BVAR** 和 **WVAR** 在同一数据段中，所以编译后，**SEG BVAR** 和 **SEG WVAR** 是相同的，且和变量本身的类型无关。

### 2. 偏移量属性运算符 OFFSET

偏移量属性运算符指示编译程序返回变量或标号操作数的偏移量。编程实践的实际情况说明，一般情况，程序员只会取内存变量的偏移量，而不太关心标号的偏移量。如在某数据段定义了如下变量：

```
WVAR  DW    100 DUP(0)
```

而下述代码：

```
MOV    SI,OFFSET WVAR
```

的源操作数表达式在程序编译时将得出一个立即数的结果；程序执行这条语句时，**SI** 将获得变量 **WVAR** 的偏移量。和 **LEA** 指令相比，指令执行后的效果是一样的，但实现的手段是有区别的：**LEA** 是纯指令实现的功能，而偏移量运算符主要是靠编译程序的功能实现的；另外 **LEA** 的目标操作数只能是 16 位通用寄存器，而带 **OFFSET** 运算符及变量的表达式经编

译后是一个立即数，所以目标操作数还可以是 16 位存储器单元。

不论使用 **OFFSET** 运算符 **LEA** 指令使 **SI** 获得了某变量的编译量，程序就可以通过“寄存器间接寻址”方式访问那个变量了。对于单变量来说，通过寄存器取得偏移量然后再访问该变量，显然不及直接使用符号地址访问来得简便；但对于上面提及的有 100 个字单元的变量 **WVAR** 或类似情况来说，几乎只能采用寄存器间接寻址方式并配合循环结构或串操作指令，才能有效地处理这类数据。

### 3. 类型属性运算符 **TYPE**

类型属性运算符 **TYPE** 返回变量的字节数或语句标号的 **FAR** 或 **NEAR** 类型。见下表：

标号及变量类型属性表

	类型属性	类型数字
变量	<b>BYTE</b>	1
	<b>WORD</b>	2
	<b>DWORD</b>	4
标号	<b>NEAR</b>	-1
	<b>FAR</b>	-2

由上表可见，标号和变量都有类型属性，都用数字来区分类型，但含义却很不一样。变量的类型数字就是变量的字节数，直观实用；而标号的属性用于说明该标号能否被其他段的代码引用：

**NEAR**            表示该标号只能被同段代码引用；

**FAR**            表示该标号能被其他段代码引用。

语句标号的属性在语句定义时，被默认成 **NEAR** 属性；标号的 **FAR** 属性的须通过 **LABEL** 伪指令设置。

## 4. LABEL、THIS 和 PTR

LABEL 伪指令用于设置或更改变量或标号的属性：

变量名    LABEL    类型

或：

标号    LABEL    类型

其中变量的数据类型可以是 BYTE, WORD, DWORD；标号的类型可以是 NEAR 或 FAR。比如：

```
WVAR LABEL WORD
BVAR DB 100 DUP(0)
```

这样定义后，就可以用字节或字属性访问这 100 个字节了。比如在做数据块清零或复制等操作时，用字属性操作是 16 位的，代码效率会更高。而：

```
NEXTF LABEL FAR
NEXT: MOV SI, BX
```

为上述数值语句定义了不同属性的标号，段内转移指令使用标号 NEXT，段间使用 NEXTF。还是需要明确的是：尽管这条语句的汇编语言源代码多了很多东西，但编译（汇编）之后，目标代码和没有标号说明时占用的空间是一样的，只是编译（汇编）程序会将同段其他位置的标号替换成该语句的段内偏移量，将不同段的使用该语句 FAR 属性标号替换成 32 位的地址指针，即 16 位的段基址和 16 位的偏移量；即使在段内使用 FAR 属性的标号，编译（汇编）程序也只会将其替换成该语句的段内偏移量，这是编译（汇编）程序优化的结果。

顺便说明一下，在上述语句中使用 EQU THIS 替换 LABEL 可以实现同样的目的，即：

```
WVAR EQU THIS WORD
BVAR DB          100 DUP(0)
```

```
NEXTF EQU THIS FAR
NEXT: MOV        SI, BX
```

给标量和语句赋予两种属性。

对于变量来说，如果只是个别地方需要用和定义时不同的属性对该变量进行访问，可使用强制类型转换运算符 **PTR** 来实现。比如变量定义如下：

```
BVAR DB          100 DUP(0)
```

而这样访问它：

```
MOV AX, BVAR
```

编译（汇编）会因类型不匹配而不能通过。而将语句改成：

```
MOV AX, WORD PTR BVAR
```

就没有问题了。强制类型转换字段的一般格式为：

数据类型 **PTR** 地址表达式

其中：数据类型是有：**BYTE**、**WORD**、**DWORD**。用操作符 **PTR** 强制类型转换后，不管其后的地址表达式原来定义的数据是什么类型，在本语句中就以 **PTR** 前面的类型为准。强制类型转换给变量赋予一种临时性的属性，只在本条语句有效。

## 5. LENGTH 和 SIZE

长度属性操作符 **LENGTH** 是内存变量操作符，它指示编译（汇编）程序返回重复操作符 **DUP** 中的重复数。如果有嵌套的 **DUP**，则只返回最外层的重复数；如果 **DUP**，则返回 1。显然，这个操作符的功能是很有限的，完全不能和 C 语言的 **Strlen** 函数相提并论。

有同样毛病的是容量属性操作符 **SIZE**，它的返回值按下列公式计算：

$$\text{SIZE 变量} = (\text{LENGTH 变量}) \times (\text{TYPE 变量})$$

**SIZE** 操作符显然“继承”了 **LENGTH** 局限。

提到这两个操作符的理由只是让我们在看代码遇到它们时不至于卡壳，建议在编程中不使用它们。

## 6.3 汇编语言的表达式

需要说明的是，汇编语言的表达式和高级语言的表达式有本质的区别：高级语言涉及到变量的运算表达式将用于在目标代码运行时控制计算机的运作，而汇编语言的表达式是用于控制编译（汇编）的。比如 C 语言的语句：

```
S=3.1415926*r*r;
```

如果 **r** 是变量，那么表达式 **3.1415926\*r\*r** 就是用于控制计算机运作的代码，其结果是在该语句执行后确定的；而汇编语言的表达式在编译（汇编）后已经是被确定的常量了。

在汇编语言中，表达式分为数值表达式和地址表达式。

### 6.3.1 数值表达式

数值表达式是在编译（汇编）过程中能够确定计算值的表达式，即编译（汇编）完成后表达式的值就完全确定；数值表达式通常是一些常量的运算组合。下面分类介绍数值表达式的运算符。

## 1. 算术运算符

算术运算符包括：

符号：+（正）、-（负）；

运算符：+（加）、-（减）、\*（乘）、/（除）和取模 MOD。

这些运算符和常量、括号等可组成数值表达式。如计算 24 小时的秒数：

24\*60\*60

我们当然可以通过计算得出这些表达式的值来，但写成表达式，计算是由编译（汇编）程序来完成的，显然比人工准确高效，也在一定程度上提高了代码的可读性。

## 2. 关系运算符

关系运算符包括符号：EQ（相等）、NE（不等）、LT（小于）、GT（大于）、LE（小于等于）和 GE（大于等于）。这些关系运算符和常量、括号也可组成数值表达式。这类数值表达式的计算结果为逻辑量：若关系不成立，数值表达式的结果为 0，否则为 0FFFFH。如：

1+1 GT 2 ;结果为 0（假）

1+1 EQ 2 ;结果为 0FFFFH（真）

## 3. 逻辑运算符

逻辑运算符包括按位操作符和移位操作符。具体是：AND（逻辑与）、OR（逻辑或）、NOT（逻辑非）、XOR（异或）、SHL（左移位）和 SHR（右移位）。这些逻辑运算符和常量、括号可组成数值表达式。如：

1 SHL 3, 47H AND 0FH, NOT 56H

它们的计算结果分别为：8，7 和 0A9H。

#### 4. HIGH 和 LOW 操作符

HIGH 和 LOW 操作符分别获取表达式计算结果的高 8 位和低 8 位。其使用格式如下：

HIGH    表达式

LOW     表达式

如：

HIGH    8A00H+126H        ;表达式的结果为 8BH

LOW     8A00H+126H        ;表达式的结果为 26H

#### 5. 运算符和操作符的优先级

汇编语言运算符和操作符的优先级按从高到低的排列如下：

汇编语言运算符和操作符的优先级

优先级：高	LENGTH、SIZE
	PTR、SEG、OFFSET、TYPE、THIS、: (用于段超越前缀)
	*, /、MOD、SHL、SHR
	HIGH、LOW
	+, -
	EQ、NE、LT、LE、GT、GE
	NOT
	AND
优先级：低	OR、XOR

强记这些符号及优先序是没有必要的，这是因为汇编语言的运算符、操作符及表达式和高级语言的运算符、操作符及表达式的地位是很不一样，汇编语言编程的核心功能并不依赖它们而是指令语句。学习这部分内容必须要明确这一点，不要在编写程序时出现这样的错误：

```
MOV AX, AL*8
```

### 6.3.2 地址表达式

地址表达式是计算存储单元地址的表达式，它可由标号、变量名和由括号括起来的基址或变址寄存器组成。其计算结果表示一个存储单元的地址，而不是该存储单元的值。

例如有变量定义如下：

```
BVAR DB 1, 2, 3
WVAR DW 1234H, 5678H
```

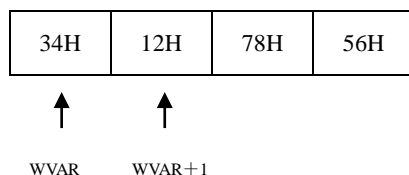
则语句：

```
MOV AL, BVAR+1
```

执行后，AL 的内容为 2，即编译（汇编）后，地址表达式 **BVAR+1** 指向变量 **BVAR** 之后内容为 2 的那个字节；而语句：

```
MOV AX, WVAR+1
```

执行后，AX 的内容不是 5678H，而是 7812H。这是因为地址表达式 **WVAR+1** 没有指向变量 **WVAR** 之后的那个字而是之后的那个字节：



这说明汇编语言变量的属性是不完备的。

请留意下面的变量定义：

```
BVAR DB ?, ?
```



VOFF DW BVAR+1

VPTR DD BVAR+1

似乎定义 VOFF 和 VPTR 的表达式包含了变量，事实上 BVAR 是一个符号地址，经编译（汇编）后已经是一个确定的常量，故表达式 BVAR+1 也是一个常量：变量 BVAR 第二个字节的偏移量，作为变量 VOFF 的初值存放到偏移量为 VOFF 的字单元中了；而当双字变量 VPTR 的初值为包含变量的表达式时，编译（汇编）程序将地址表达式指向的变量的 32 位地址指针赋给 VPTR。编译（汇编）程序这样做让人理解起来有点绕，但当把一个变量作为一个双字变量的初值时，也只能这样编译才可行和有使用价值。

## 6.4 段定义伪指令与源程序框架

本节，我们通过一个完整的实例，介绍汇编语言和程序整体结构相关的内容。通过本节的学习，我们会找到汇编语言编程和其他语言编程的类似的起点；找到本章所学内容的“归属”。

这个程序的功能很简单，就是在屏幕上显示“Hello,World!”:

stack0	segment stack	;堆栈段	Direct	proc	;显示子程序
	dw 40h dup(0)			push bx	
stack0	ends			push di	
				mov bh,ah	
data	segment	;数据段		mov al,dh	
Hello	db "Hello,World!",0			dec al	
data	ends			mov bl,160	
				mul bl	
code	segment	;代码段		mov di,ax	
	assume cs:code,ds:data,ss:stack0			dec al	
main:	mov ax,data	;ds指向数据段		mov al,dl	
	mov ds,ax			mov bl,2	
	mov ax,0b800h			mul bl	
	mov es,ax			add di,ax	
			Next:	mov al,[si]	
	mov dh,10			cmp al,0	
	mov dl,8			jz return	
	mov ah,1eh			mov es:[di],al	
	lea si,Hello			inc di	
	call Direct	;调用子程序		mov es:[di],bh	
				inc di	
	mov ah,0			inc si	
	int 16h			jmp Next	
			return:	pop di	
	mov ah,4ch	;程序结束，退回		pop bx	
	int 21h	;DOS		ret	
			Direct	endp	
			code	ends	
				end main	

为节约篇幅和阅读方便，该程序的下半段“显示子程序”印到了右边，这里特别要提醒读者，汇编语言的源程序是不能这样书写的。

和 C 语言实现相同功能的代码相比，汇编语言的源程序显得冗长许多，但学习到现在我们应该知道，任何高级语言的语句、函数都是机器语言的组合，故 C 语言和汇编语言相比的那种简练是表面的，最终只能以编译（汇编）出来的目标代码作比较。

不过代码效率和程序功能不是我们现在关注的重点，我们现在讨论的是汇编程序的框架结构，是 8088/8086 CPU 的段机制对程序框架的影响及程序结束退回 DOS 的方法等，这也和程序设计方法学或软件工程讨论的一般应用程序的结构框架是有区别的。

### 6.4.1 段定义伪指令

在上面的程序代码中，我们看到成对出现了 3 次 **SEGMENT** 和 **ENDS**，这是汇编语言段定义伪指令。我们可以想象得到，8088/8086 CPU 内存的段管理机制对程序结构的影响是必然的：任何类型存储器的访问，不论是存储变量、堆栈操作还是读取指令，都必然会有段的参与。

段定义伪指令的一般格式如下：

段名     **SEGMENT**    [对齐类型] [组合类型] [类别]  
          [段内语句序列]

段名     **ENDS**

其中，段名是由编程者根据段的用途按标识符的规则定义的，且前后二个段名要相同。和其他写在一行的伪指令语句不同，段定义伪指令和结构定义语句相似，或者也可以看作是一种语句括号：以关键字 **SEGMENT** 开头，以段结束伪指令 **ENDS** 结束。类似的这种伪指令语句还有过程定义语句和宏定义语句等。和其他伪指令语句定义的符号名的另一个不同是，汇编语言程序的段名可以是唯一的，也可以与其它段同名。如果有二个段同名，则后者被认为是前段的后续，它们属同一段。

关键字 **SEGMENT** 之后的“对齐类型”、“组合类型”和“类别”为段属性。段属性必须按如上所述的顺序说明，而是否描述可根据需要选择。如果没有指定某个属性，编译（汇

编) 程序将使用该属性的缺省值。下面对段的几个属性作简要说明。

### 对齐类型

对齐类型包括 **BYTE**、**WORD**、**DWORD**、**PARA** 和 **PAGE** 共 5 个类型, 用于描述当前段对起始地址的要求:

- **BYTE**: 逻辑段从未占用的下一个字节开始。
- **WORD**: 逻辑段从未占用的下一个字开始。有可能空闲 1 个字节; 段的起始地址必须是偶数。
- **DWORD**: 逻辑段从未占用的下一个双字开始。有可能空闲 3 个字节; 段起始物理地址的低 2 位为 0。
- **PARA**: 逻辑段从下一个未占用的节 (16 个字节) 开始, 有可能空闲 15 个字节; 段起始物理地址的低 4 位为 0。当未指定对齐类型时, 节对齐为默认对齐类型。
- **PAGE**: 逻辑段从下一个未占用的页 (256 字节) 开始。有可能空闲 255 个字节, 段的起始物理地址的低 8 位为 0。

段对齐类型 **PARA** 是一个适用于所有类型段的对齐类型, 也是缺省对齐类型。我们给出的例子没有描述对齐类型, 即使用缺省的 **PARA** 节对齐类型。

### 组合类型

组合类型是告诉连接程序如何把不同模块中段名相同的段合并在一起。具体的组合类型如下:

- **NONE**: 表示当前段在逻辑上独立于其它模块, 并有其自己的基地址。**NONE** 是缺省的组合类型。
- **PUBLIC**: 表示当前段与其它模块中同段名的 **PUBLIC** 类型段组合成一个段。组合的先后次序取决于 **LINK** 程序中目标模块排列的次序。在组合时, 后续段的起始地址要按其对齐类型进行定位, 所以, 同名段之间可能有间隔。
- **COMMON**: 表示当前段与其它模块中同名段重叠, 也就是说, 它们的起始地址相同。最终段的长度是同名段的最大长度。由于段覆盖, 所以, 前一同名段中的初始化数据被后续段的初始数据覆盖掉。
- **STACK**: 组合类型 **STACK** 表示当前段是堆栈段, 其组合情况与 **PUBLIC** 相同。
- **AT** 数值表达式: 以数值表达式的值作为前段所的起始地址。

例子中除了堆栈段的组合类型是 **STACK** 外, 其他的段都没有明确描述组合类型, 即为

默认的 NONE 独立组合类型。

### 类别

类别是一个由程序员按标识符规则设定的带单引号的字符串；连接程序将类别名相同的段组合起来在存储器中连续存放。

为突出重点，我们只要求程序框架堆栈段使用 **STACK** 组合类型，并是所以 '**STACK**' 类别。

## 6.4.2 段声明伪指令的段初值

由于段名是自定义的标识符，我们不能依赖他们向编译（汇编）程序传达段定义的类型信息，即那个段是代码段，那个段是堆栈段，那个段是数据段等。那么编译（汇编）程序是如何知道程序定义的段的类型并建立段名和段寄存器的联系的呢？请留意例子程序中的这条语句：

```
assume cs:code,ds:data,ss:stack0
```

这条语句通过建立了段名和段寄存器间的对应关系，让编译（汇编）程序知道了程序的段是如何安排的。**ASSUME** 语句的一般形式是：

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

虽然 **assume** 伪指令指明了某段名和某段寄存器的关系，但并没有确定段寄存器的初值。之所以会这样是因为当典型的 **DOS** 程序运行时，操作系统会根据内存的使用情况把执行程序引导、安放到内存的适当位置，这里所说的“安放”就是通过初始化段寄存器来实现的。这样做不仅让我们在编写单个程序时不用去考虑段间冲突，也不用担心两个执行程序在内存出现冲突。**DOS** 在引导程序进入内存时对各个段寄存器的初始化是有区别的：**CS** 和 **SS**（当对应段的组合类型定义成 **STACK** 时）被直接初始化了，而在装入程序过程中 **DS** 和 **ES** 寄存器有其它用途，所以只能在用户程序中用指令专门对其作初始化，以装入用户数据段和附加段的基址。我们看看程序举例中对 **DS** 赋值的两条语句：

```
mov    ax,data  
mov    ds,ax
```

通过 **ax** 搭桥的原因是由于所有的段寄存器都不能接收立即数；而 **data** 和其他段名一样是一种特殊的立即数：它们的性质一直是立即数，但它们的值是在程序装入时由操作系统确定的。

### 6.4.3 IP 和 SP 的初值

我们先说说给堆栈指针 **SP** 赋初值。如果对应段的组合类型选择 **stack**，栈指针 **SP** 将被设置成该段定义的字节数的总和；如果没有一个段的段组合属性设置成 **stack**，就必须用类似初始化 **DS** 和 **ES** 那样让 **SS** 指向你设置的堆栈段并让 **SP** 指向栈顶。在我们的例子中，使用的是前一种做法。

然后说说如何给 **IP** 赋初值。请注意程序举例中最后一条语句：

```
end    main
```

**end** 是一条伪指令，表示源程序到此为止，编译（汇编）程序不处理该语句之后的任何内容。另外，**end** 后面可以附带一个在代码段中已定义的标号，用以说明程序启动的偏移量。如果源程序是一个独立程序或主模块，**end** 后面一定要带一个如上所述的标号，否则 **end** 之后就不能跟除注释之外的任何东西。

所谓程序的运行，就是程序文件被操作系统引导到内存中，然后让指令指针 **CS:IP** 指向程序的起点，比如 **main**，具体地讲就是给 **CS** 和 **IP** 赋值；而 **IP** 所赋的值，就是由伪指令 **end** 指定的。

### 6.4.4 源程序的基本框架

在我们讨论的例子中将和具体应用需要的内容全部去掉，就是我们看到的下面这个源程序的基本框架：

```

stack0 segment stack      ;堆栈段
dw 40h dup(0)
stack0 ends

data segment              ;数据段
; <变量定义>

data ends

code segment              ;代码段
assume cs:code,ds:data,ss:stack0
main: mov ax,data         ;ds指向数据段
      mov ds,ax

      ; <执行代码>

      mov ah,4ch          ;程序结束，退回DOS
      int 21h

code ends
end main

```

有了这个框架，我们要做的工作就是在数据段定义常量和变量、在代码段编写程序了，这和高级语言编程的套路已相当接近了。本章介绍的伪指令除描述程序框架外，主要集中在数据段定义常量和变量；表达式则根据需要作为指令或伪指令的操作数；操作符也根据功能和需要，出现在各种表达式中。

虽然在这个框架中，只有一个代码段，没有专门的附加段，堆栈容量也是个不大的固定值，但对汇编语言来说，一个段即 64K 的目标代码对应的源程序已经有相当的规模了，完全能满足我们学习的需要。

在阅读他人的程序时，有可能发现程序的代码段是这样构成的：

```

code segment              ;代码段
assume cs:code,ds:data,ss:stack0
myapp proc far
main:  push ds             ;将指向int 20h的指针压栈
      xor ax,ax
      push ax

      mov ax,data         ;ds指向数据段
      mov ds,ax

      ; <执行代码>

      ret                 ;结束程序返回DOS
myapp endp
code ends

```

即把代码段做成一个被 DOS 调用的子程序的形式。由于要弄清为什么可以这样做会涉及到一些 DOS 的具体机理，就不详细讨论了，只要求我们知道这也是程序结束退回 DOS 的一种方式即可。

## 6.5 编制汇编语言程序的完整过程

8088/8086 汇编语言程序编制的完整过程和其他语言是共通的，一般都需要经过编辑、编译（汇编）、连接及调试几个步骤和不可避免的纠错折返。汇编语言的编程工具不具备成熟的集成开发环境（IDE），也考虑 8088/8086 汇编语言的目标代码最终是在 DOS 平台的命令行上运行的，所以我们仍然以命令行方式介绍汇编语言的编程过程。

### 6.5.1 编程工具及经典过程

可使用任何纯文本编辑软件编辑汇编语言的源程序。我们接触较多的 Windows 平台纯文本的编辑软件有 UltraEdit、Edit Plus 等。

编译（汇编）软件，可使用微软的 MASM 或 Borland 公司的 TASM；连接软件可使用微软的 LINK.EXE 或 Borland 公司的 TLINK.EXE；至于调试软件，建议微软的 DEBUG 和 Borland 的 TD 都使用，各有千秋。下面以 Borland 公司的 TASM、TLINK、TD 及前述 Hello.asm 为例作扼要介绍程序编制的基本过程。

假定 TASM.EXE、TLINK.EXE、TD.EXE 及 Hello.asm 都在 D 盘名为“ASMTTest”的文件夹中，则编译、连接及使用 TD 调试的命令行是这样的：

```
D:\ASMTTest>TASM HELLO;
```

```
D:\ASMTTest>TLINK HELLO;
```

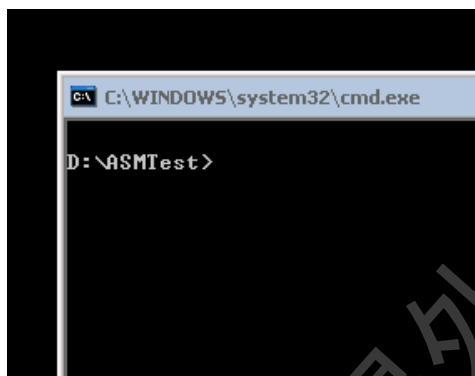
```
D:\ASMTTest>TD HELLO.EXE
```

```
D:\ASMTTest>HELLO
```

现在几乎找不到运行纯 DOS 的桌面系统了，但上述命令及程序执行必须在 DOS 环境下

的命令行进行。Windows 的编辑及文件管理功能是完备方便的，我们也相对熟悉，所以源程序的编辑、修改等工作就在 Windows 平台完成，只将编译（汇编）、连接、运行和调试放在 DOS 环境进行。

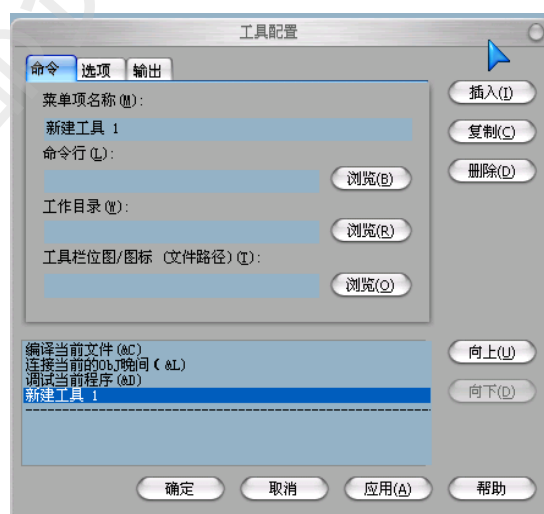
如果在鼠标右键菜单上设置了“DOS 快速通道”功能，则通过操作 ASMTTest 即可方便地进入 DOS 状态及编程文件夹：



## 6.5.2 用 UltraEdit 设置简易的汇编语言编程环境

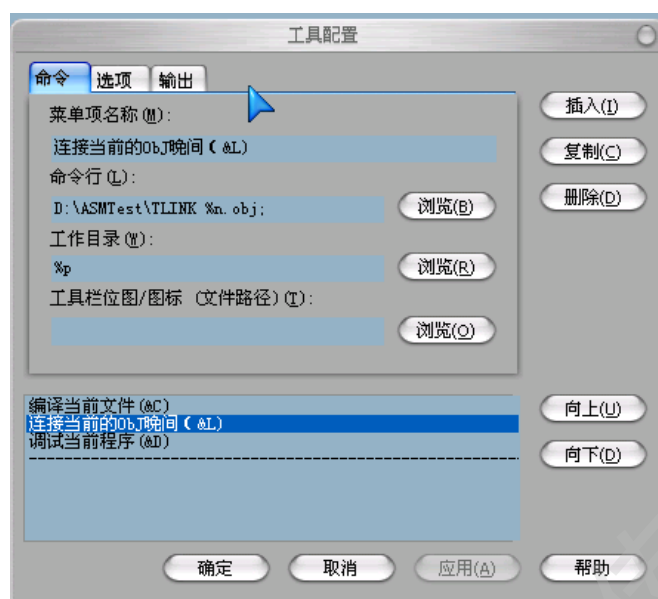
使用 UltraEdit 编辑源程序完全是按 Windows 文本编辑的常规操作进行的，所以在这个环节应该没有什么障碍。这里想说的是，可通过 UltraEdit “高级/工具配置” 专门为汇编语言增加编译、编辑及调试功能，打造一方便实用的汇编语言集成开发环境。

点击 UltraEdit 主菜单的“高级/工具配置”，并在屏幕弹出的对话框中点“插入”：



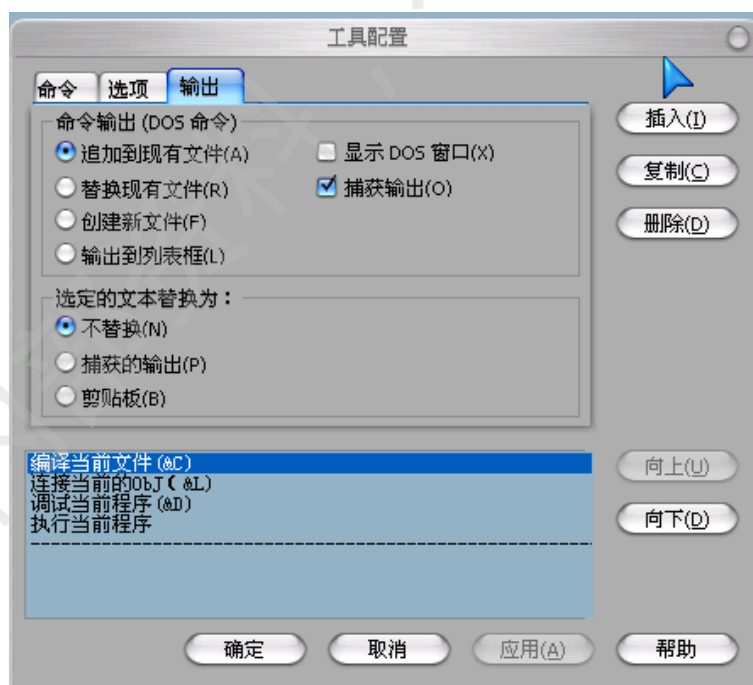
以“连接”为例，设置如下：





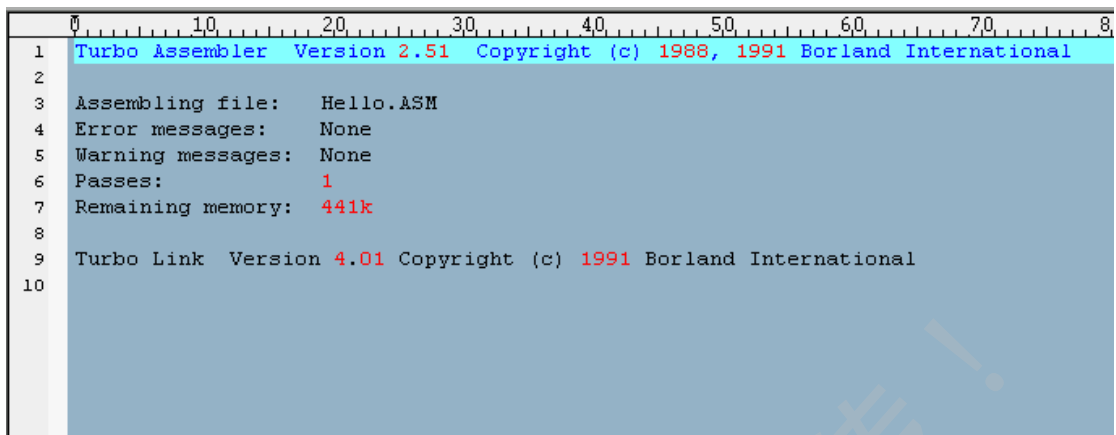
注意设置项中的%n和%p，前者表示当前源文件的主名，后者为当前文件夹。

需要注意的是，设置“编译（汇编）”和“连接”时，配置对话框的“输出”选项卡应选择“捕获输出（O）”选项，而执行或调试设置应该选“显示DOS窗口（X）”：



设置完成后，主菜单“高级”的下拉菜单会增加汇编语言的若干工具，一个简单实用的汇编语言集成开发环境就建立完成了。

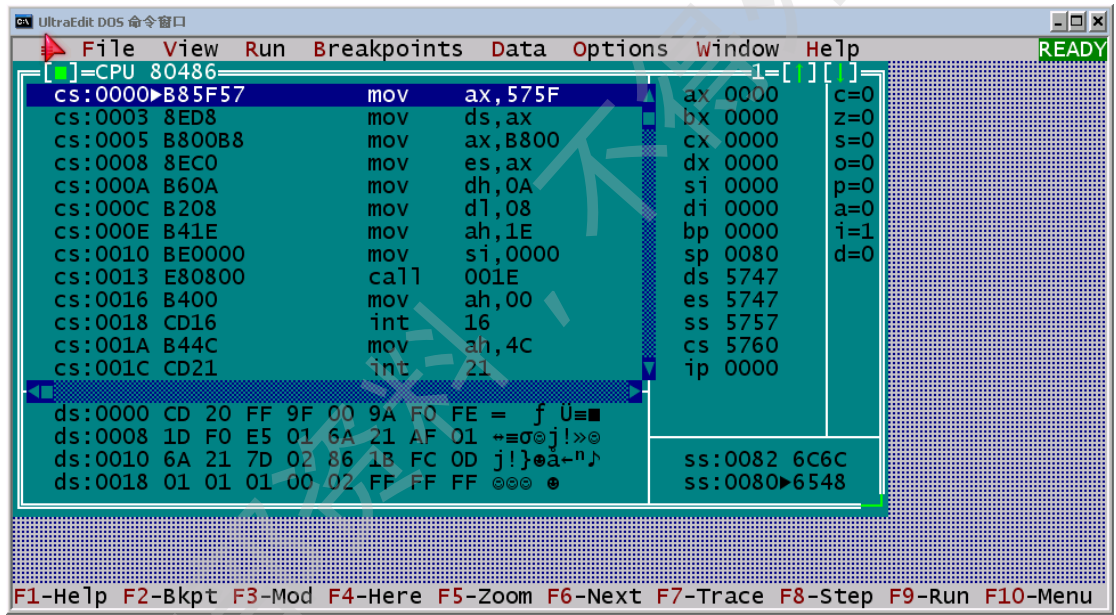
下图是 UltraEdit 编译和连接正确的结果显示：



The screenshot shows the output of Turbo Assembler and Turbo Link in the UltraEdit editor. The text is as follows:

```
1 Turbo Assembler Version 2.51 Copyright (c) 1988, 1991 Borland International
2
3 Assembling file: Hello.ASM
4 Error messages: None
5 Warning messages: None
6 Passes: 1
7 Remaining memory: 441k
8
9 Turbo Link Version 4.01 Copyright (c) 1991 Borland International
10
```

下图是 UltraEdit 调用 TD 调试程序的画面：



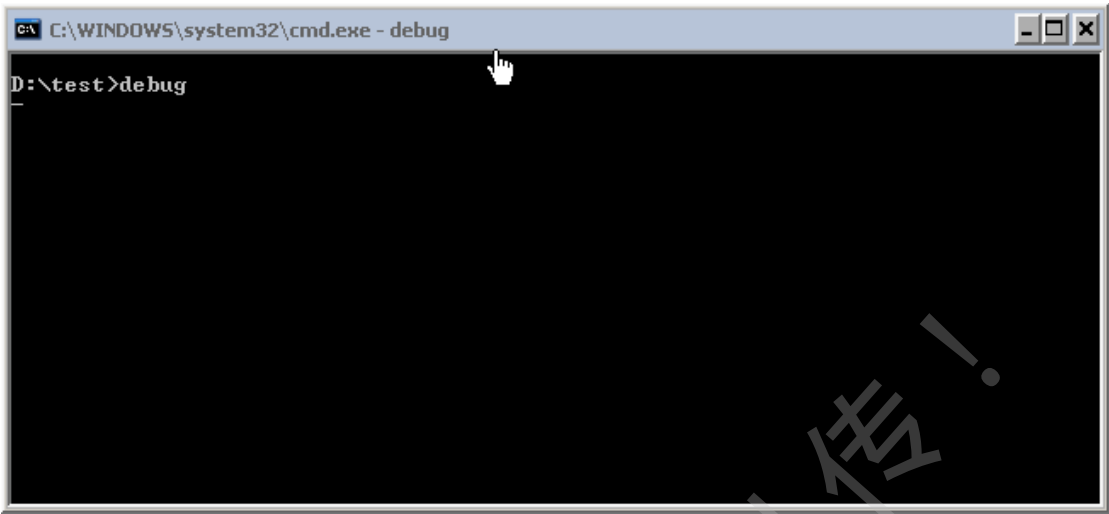
真正掌握上述内容必须通过上机实践，实验课会安排这些内容的，我们就不在这里纸上谈兵了。

### 6.5.3 DEBUG 常用命令简介

DEBUG.EXE 是 DOS 平台命令行方式的调试软件，具有显示和修改寄存器或存储单元内容、跟踪程序执行、观察中间结果等程序调试的基本功能。尽管它早在 DOS 时代就已经不是主流的程序调试工具了，但对我们学习 8088/8086 汇编语言却很有用，而且至今还保留在 Windows 系统中。这里我们仅围绕汇编语言学习的需要作简要介绍。

一. 进入 DEBUG

在 DOS 提示符下键入 DEBUG，然后按 Enter 确认，屏幕显示：



和 DOS 平台完成单纯任务（比如文件复制等）的命令不同，DEBUG 启动后将进入它自己的命令行状态，即“-”状态。这时系统就可以接受 DEBUG 的命令了。

二. 和汇编学习有关的几个 DEBUG 命令

和汇编学习有关的几个 DEBUG 命令

命令字符	功 能
?	显示 DEBUG 命令列表
q	退出 DEBUG
r	显示或设置寄存器值
d	显示默认或指定地址范围内存单元的值
e	从指定地址开始设置各字节单元的值
a	汇编命令
u	反汇编命令
g	运行指定地址开始的代码

和汇编学习相关的 DEBUG 命令如上表所示。DEBUG 命令都是单字符的，其中“?”命令可省去上机时记忆命令的麻烦。

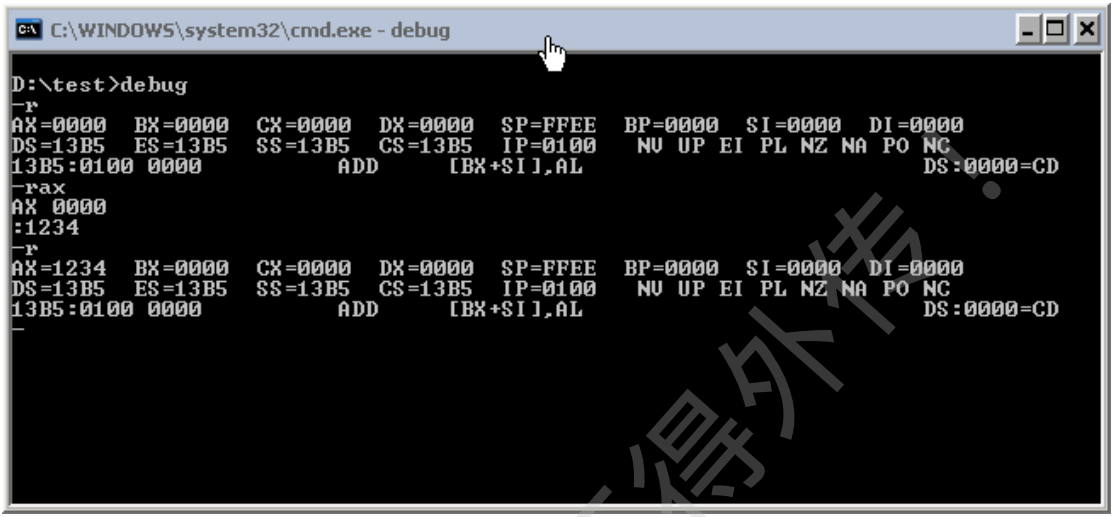
三. 常用 DEBUG 命令的使用

为使介绍更直观、准确，方便同学上机学习，下面的介绍将结合上机进行；另外 DEBUG

只显示和接受 16 进制数，请学习时留意。

在 Windows 平台进入 DOS 可点击“开始”、“运行”，然后在运行对话框的“打开(O)”后键入“cmd”即可，也可键入“DEBUG”直接进入。

R 命令



从上图可见，没有参数的 r 命令将显示所有寄存器的内容；指定寄存器后 DEBUG 先显示指定寄存器的内容，然后用冒号“:”提示等待设置输入。上图记录了将寄存器 AX 设置成 1234H 的过程。

状态寄存器的内容是按状态位逐位显示的，即上图的字符对 NV UP EI 等。由于显示行位置的限制，DEBUG 没有将状态按 ZF=0 的方式显示。下表给出了 DEBUG 双字符状态表示的含义：

DEBUG 双字符状态表示的含义

标志位	含义	FLAG = 1	FLAG = 0
OF	溢出(是/否)	OV OVerflow	NV Not oVerflow
DF	方向(减量/增量)	DN DowN	UP UP
IF	中断(允许/关闭)	EI Enable Interrupt	DI Disable Interrupt
SF	符号(负/正)	NG NeGative	PL PLus
ZF	零(是/否)	ZR ZeRo	NZ Not Zero
AF	辅助进位(是/否)	AC Auxiliary Carry	NA Not Auxiliary
PF	奇偶(是/否)	PE Parity Even	PO Parity Odd
CF	进位(是/否)	CY CarrY	NC Not Carry

## D 命令

```

C:\WINDOWS\system32\cmd.exe - debug
D:\test>debug
-d
13B5:0100  00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00  .....4...
13B5:0110  00 00 00 00 00 00 00 00 00-00 00 00 00 34 00 A4 13  .....
13B5:0120  00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00  .....
13B5:0130  00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00  .....
13B5:0140  00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00  .....
13B5:0150  00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00  .....
13B5:0160  00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00  .....
13B5:0170  00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00  .....
-d100 10f
13B5:0100  00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00  .....

```

从上图可见，没有参数的 D 命令将显示当前单元开始的 128 个字节单元的内容；右边现在有很多“.”的部分按 ASCII 码显示对应单元的内容，不能显示的则用“.”代替。如果在 D 之后给出了地址范围，该命令就只显示地址范围内各字节的内容。

## E 命令

```

C:\WINDOWS\system32\cmd.exe - debug
D:\test>debug
-e100
13B5:0100  00.1    00.2    00.3    00.4    00.
-d100 103
13B5:0100  01 02 03 04  ....

```

E 命令必须指定参数，即要设置新值的字节单元的地址。按 Enter 确认后，DEBUG 在显示指定字节的内容之后用小数点“.”等待设置输入。如果在设置内容后键入空格，DEBUG 将显示下一个字节的内容并继续等待设置；如果不修改当前字节的内容而希望修改后续字节，可键入空格跳过该字节。按 Enter 将结束 e 命令回到“-”状态。

## A 命令、G 命令和 U 命令

```

C:\WINDOWS\system32\cmd.exe - debug
D:\test>debug
-a
13B5:0100 mov al,55
13B5:0102 mov cx,8
13B5:0105 rcl al,1
13B5:0107 rcr ah,1
13B5:0109 loop 105
13B5:010B int 3
13B5:010C
-g=100
AX=AA00 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=13B5 ES=13B5 SS=13B5 CS=13B5 IP=010B OF UP EI PL NZ NA PO NC
13B5:010B CC INT 3
-u100 10b
13B5:0100 B055 MOV AL,55
13B5:0102 B90800 MOV CX,0000
13B5:0105 D0D0 RCL AL,1
13B5:0107 D0DC RCR AH,1
13B5:0109 E2FA LOOP 0105
13B5:010B CC INT 3

```

上图显示了执行 A、G 和 U 命令的情况。A 命令后跟一地址参数，表示将从指定单元键入程序。按 Enter 确认后，屏幕将显示当前地址指针并等待语句输入，即进入“编程状态”。需要明确的是，我们通过 A 命令键入的汇编语言语句并不能被 CPU 直接认识，写入计算机的是被 DEBUG 逐句翻译的机器代码；如果键入错误，DEBUG 将立刻提示你。

在当前地址指针后直接按 Enter 将结束 A 命令。

G 命令和 A 命令的格式是一样的，它通知 CPU 从指定地址开始执行内存单元中的代码。请留意上图偏移量为 010b 的语句 int 3，执行它时会结束程序、显示和没有参数的 r 命令一样的状态并回到 DEBUG。Int 是中断调用，将在后续章节专门介绍。

U 命令可以没有和有 1 个或 2 个参数。比较典型的用法如上图所示使用 2 个参数，其功能是将参数指定地址范围内各单元的机器代码反编译（汇编）成汇编语言代码。和上面使用 A 命令写入程序时的那一段相比，每行汇编代码对应的机器码被显示出来了，其实这才是这些单元中存放的实际内容。

这里不打算介绍 DEBUG 的 T 命令。程序调试的内容我们将在第 6 章介绍使用更方便的 TurboDebug。

尽管 DEBUG 在今天的软件开发中已经没有什么太大的用场，但对我们学习汇编语言却很有帮助：它使我们能用简单的命令观察和设置 CPU 的寄存器和内存单元，可以使用 A 和 G 命令测试单条指令或短小程序，是我们开始结合实践学习汇编语言不可多得的重要工具。

## 习题 6

1. 试说明下述概念的异同:

- ① 指令和伪指令。
- ② 符号常量、变量和标号。
- ③ 表达式的运算符和运算指令。
- ④ ENDS 和 END。

2. 用示意图说明下述为指令语句分配的字节数及各字节内容（假设数据段的段基址是 0ABCDH, 且各变量依序从偏移地址 0 开始定义）。

VARA	EQU	10
VARB	DB	10
VARC	DW	1234H
VARD	DD	12345678H
VARE	DB	2 DUP (1, 2), 0
VARF	DW	VARB
VARG	DD	VARB

3. 说明下面代码中 N1、N2 和 N3 的异同:

N1	EQU	10
N2	DB	10
N3	DW	10

4. 按下述要求在数据段定义变量:

- ① 定义一字符串变量且初值为 “Hello, World!”。
- ② 定义一字节型变量且初值为 0。
- ③ 定义一未指定初值的 100 个单元的字型变量。

④ 定义一值为 1000 的符号常量。

5. 对数据段定义的下述变量：

```
NUM1    DB      60
NUM2    EQU     50
NUM3    DB      20H
```

判断下述语句是否有错并指出是什么错误。

- ① MOV NUM2, AL
- ② MOV AX, NUM3
- ③ MOV NUM3, NUM1
- ④ MOV NUM1, NUM2
- ⑤ MOV BX, AX+NUM2
- ⑥ MOV NUM1, AL and 0F0H
- ⑦ MOV AX, NUM3-NUM1

6. 对于下面的数据定义, 写出各条指令执行的结果：

```
FLDB DW 0A24FH
TABLE DB 32H, 52, 0C2H, 213
TEA EQU WORD PTR TABLE
ARRAY DB 'ABCD'
COUNT EQU $-ARRAY
```

- ① MOV AX, FLDB
- ② MOV BX, TEA
- ③ MOV CH, TABL3+2
- ④ MOV DL, ARRAY



## ⑤ MOV DH, COUNT

本题表达式中出现的\$, 是“编译（汇编）位置指针”，在编译（汇编）时将被替换成数据段的当前地址偏移量。

7. 假设数据段有下属变量定义：

```
INAME    DB    31 DUP(?)
```

```
ADDRESS DB    31 DUP(?)
```

```
CITY     DB    16 DUP(?)
```

```
ID       DB    1, 2, 1, 6, 8
```

① 用一条 MOV 语句将 INAME 的偏移量送入 SI。

② 用一条指令将 ID 的头两个字节的的内容送入 BX。

③ 写出后三个字段的长度表达式。

8. 在下述程序框架中分号“;”后面，同一行代码的功能。

```
stack0    segment stack 'stack'    ;
           dw 40h dup(0)           ;
stack0    ends

data      segment                   ;
           ;
data      ends

code      segment                   ;
           ;在此说明下一句
           assume cs:code,ds:data,ss:stack0
main:     mov ax,data                ;
           mov ds,ax
           ;

           mov ah,4ch                ;
           int 21h

code      ends                      ;
           end main                 ;
```

9. 编写一完整程序，该程序将数据段定义的字符串“Hello, World.”末尾的句号替换成感叹号“!”。

10. 编写一完整程序，分别在两个数据段 DSEG1 和 DSEG2 中各定义一个字型变量 X 和 Y 并计算两数差的绝对值。

内部资料，不得外传！

## 第 7 章 分支与循环程序设计

本章我们将介绍程序转移类指令中的无条件、条件转移指令、循环控制指令，以及以它们为基础的分支与循环程序设计。实质上，循环结构程序是分支结构程序的一种特例，而实现分支结构的核心原理在于条件转移指令能够根据标志位状态决定程序流程。在一章中，读者会深刻体会到 CPU 中的标志寄存器是实现分支与循环结构程序的硬件基础。程序转移类指令，包括下一章将要介绍的子程序调用、返回指令，不会影响任何标志位，部分条件转移指令或循环控制指令只针对现有的标志状态进行判断，从而决定程序的执行流程。换言之，程序转移类指令只可能对标志位实施读操作，而不会实施写操作。因此，在本章中介绍各种转移类指令时，不再给出有关标志位影响的说明。

### 7.1 无条件转移指令

无条件转移指令是指能够无条件改变程序执行流程，改变 CPU 顺序执行下一条指令的模式，而将程序流程转移到代码段指定位置的指令，该指定的转移位置称为**目标地址**。无条件转移指令的功能是通过无条件修改 IP 寄存器的内容或修改 IP、CS 寄存器的内容来实现的。根据无条件转移指令的转移范围以及目标地址的获取方式，可以分为段内直接短转移、段内直接长转移、段内间接转移、段间直接转移、段间间接转移。这些不同类别的无条件转移指令在机器指令系统中是不同的机器指令，但在汇编指令系统中，它们使用相同的指令助记符“JMP”，即 jump 之意。读者也许会认为，无条件转移指令的用途类似于高级语言中的 GOTO 语句。现在高级语言的结构化程序设计中已经不再使用 GOTO 语句了，那么无条件转移指令是否也可以从指令系统中删除掉呢？我们将在分支程序设计一节中回答这个问题，并阐述无条件转移指令存在的必要性。下面我们将分类对无条件转移指令进行介绍。

#### (1) 段内直接转移

**指令格式：**JMP 标号

JMP NEAR PTR 标号

第一种格式中的标号与 JMP 指令位于同一代码段，并且定义的标号类型为 NEAR；第

二种格式中的标号也与 **JMP** 指令位于同一代码段，但定义的标号类型允许为 **FAR**，**PTR** 运算符将标号类型在 **JMP** 指令中暂时的修改为 **NEAR**。

段内直接转移指令的机器指令由操作码字段、位移量字段（**DISP** 字段）构成（机器指令的一般构成请参见第三章针对机器指令的相关说明）。在汇编阶段（注意，指汇编程序将我们的源程序翻译为目标代码的阶段），汇编程序会生成机器指令中的位移量，生成方法为： $DISP = \text{标号的段内偏移量} - \text{JMP指令后那条指令的段内偏移量}$ 。尽管这两个偏移量均是无符号数编码，由于标号的段内偏移量既可能大于 **JMP** 指令后那条指令的段内偏移量，也有可能小于它，因此所计算出的 **DISP** 可能为正数，也可能为负数，其编码一定是带符号数编码，即补码。此运算看起来很奇怪，由两个无符号数相减，得到了一个补码结果，实际上其中的原理是自然的。将 16 位的无符号偏移量看作 17 位的正数补码，最高位即第 16 位是不可见的，我们假定它总为 0，那么两个 17 位正数补码相减得到一个补码运算结果就不再是一件奇怪的事情了。但是，与算术运算指令中的补码加、减运算类似，在计算 **DISP** 过程中也有可能出现补码运算溢出的情况，如果溢出发生，则汇编程序会在屏幕上报告一个语法错误，指出 **JMP** 指令的目标地址已超出其转移范围，程序员必须修改该错误后才能得到目标代码。

当所计算出的 **DISP** 位于 -128~127 范围内时，它可用 8 位补码表示，此时汇编程序生成的机器指令为**短转移指令**，机器指令的操作码与位移量字段总共占据 2 个字节；而当所计算出的 **DISP** 超出 -128~127 范围时，**DISP** 必须使用 16 位补码表示，此时汇编程序生成的机器指令为**长转移指令**，机器指令的操作码与位移量字段总共占据 3 个字节。段内直接转移最大范围为：向低地址端最大转移 -32768 字节（约 32K），向高地址端最大转移 32767 字节（约 32K），若转移范围超出，则不能使用直接转移方式。

**指令功能：** $(IP) + DISP \Rightarrow IP$ ，导致程序流程无条件地转移到目标地址，CPU 下一条即将执行的指令改变为目标地址处的指令。

指令功能中所述操作是在程序执行阶段，即执行 **JMP** 指令时完成的。指令功能中修改（**IP**）的操作是一个加法运算，与生成 **DISP** 的原理类似，在作加法时将（**IP**）理解为 17 位正数补码，并且由于汇编程序生成 **DISP** 时是保证了没有溢出发生的，因此 **JMP** 指令中使用加法来修改（**IP**）同样不会导致溢出产生，我们应将所得到的加法结果重新理解为 16 位的无符号偏移量。

## (2) 段内间接转移

**指令格式:** JMP 16 位通用寄存器名称

JMP 字类型的内存单元

**指令功能:** (16位寄存器)  $\Rightarrow IP$  或 (字类型的内存单元)  $\Rightarrow IP$ , JMP 指令将 16 位通用寄存器或字类型内存单元中的数据解释为目标地址偏移量, 并将其传送至 IP 寄存器保存, 从而使程序流程转移至目标地址。

由于段内转移指令直接使用 16 位无符号偏移量替换 IP 中原有偏移量, 所以段内转移指令可以将程序流程转移至 64K 段内的任意位置, 而一个段的最大空间即为 64K 字节。因此, 段内间接转移中没有目标地址超出转移范围的情况发生。段内间接转移的示例如例 7.1.1 所示。

**例 7. 1. 1** 无条件转移中的段内间接转移指令示例如下。

JMP BX

该指令将 (BX) 解释为目标地址偏移量, 并传送至 IP 保存。

JMP [BX]0100H

该指令将逻辑地址为 (DS): (BX) + 0100H 的字类型内存单元中的 16 位数据解释为目标地址偏移量, 并传送至 IP 保存。

## (3) 段间直接转移

**指令格式:** JMP 标号

JMP FAR PTR 标号

第一种格式中的标号与 JMP 指令不在同一个代码段中, 并且定义类型为 FAR; 第二种格式中的标号与 JMP 指令可能在不同代码段, 也可能在同一代码段, 但其定义类型可能为

NEAR, PTR 运算符将 JMP 指令中的标号类型暂时修改为 FAR。

**指令功能：** 标号的段内偏移量  $\Rightarrow$  IP  
 标号所在段的段基值  $\Rightarrow$  CS

其中，标号的段内偏移量与所在段的段基值都由机器指令提供，由于同时修改了 IP 和 CS，程序流程从当前代码段转移至目标地址所在的代码段。并且，由于使用完整的逻辑地址作为目标地址，段间直接转移没有任何转移范围限制，理论上可在 1M 字节内存空间中任意实施流程转移。

#### (4) 段间间接转移

**指令格式：** JMP DWORD PTR 内存单元逻辑地址

**指令功能：** 将双字内存单元中的低地址字解释为目标地址偏移量，传送至 IP 保存，将双字内存单元中的高地址字解释为目标地址段基值，传送至 CS 保存，从而使程序流程转移至目标地址。与段间直接转移类似，由于使用完整逻辑地址作为目标地址，转移范围在 1M 字节空间内不受限制。

**例 7.1.2** 段间间接转移指令示例如下。

```
.....
ADDR1 DD L1
.....
JMP DWORD PTR ADDR1
.....
```

## 7.2 条件转移指令

在这一小节中，我们将说明各条件转移指令的功能。由于条件转移指令是带条件的实施程序流程转移，那么在执行条件转移指令时，程序流程就有了两种可能的取向：一种是条件不满足，条件转移指令未实施流程转移，那么 CPU 将顺序执行下一条指令；另一种是条件满足，条件转移指令将程序流程转移至目标地址。而这种带条件的转移机制正是分支结构程

序与循环结构程序实现的理论基础。在硬件上，多数条件转移指令所使用的判断条件都来自于状态标志位的当前状态，因此，我们可以称状态标志是实现分支、循环结构程序的硬件基础。读者可以将标志寄存器这一硬件结构推广至其它型号的 CPU，因为在多数 CPU 中，它都是实现分支、循环结构程序的硬件基础。在 8086/8088 CPU 的指令系统中，所有条件转移指令实现的流程转移均为段内短转移，即转移范围向低地址端不超过-128 字节，向高地址端不超过 127 字节。在程序设计中，如果分支结构过于庞大，则应考虑结合无条件转移指令来增大转移范围。下面我们分类对条件转移指令进行介绍。除操作助记符外，条件转移转移指令的指令格式基本一致。

**指令格式：**JX NEAR 类型的段内标号

格式中 JX 表示抽象的助记符，可以使用任意具体的条件转移指令助记符来替换它。

### (1) 单标志位转移指令

单标志位转移指令是指判断条件基于单个标志位状态的条件转移指令。这种转移指令一般成对出现。例如 JC 指令当 CF=1 时转移到目标地址，JNC 指令当 CF=0 时转移到目标地址。单标志位指令的助记符、功能如图 7.2.1 所示。

指令助记符	指令功能
JC	CF=1，转移至目标地址 CF=0，顺序执行
JNC	CF=1，顺序执行 CF=0，转移至目标地址
JO	OF=1，转移至目标地址 OF=0，顺序执行
JNO	OF=1，顺序执行 OF=0，转移至目标地址
JS	SF=1，转移至目标地址 SF=0，顺序执行
JNS	SF=1，顺序执行 SF=0，转移至目标地址

JZ / JE	ZF=1, 转移至目标地址 ZF=0, 顺序执行
JNZ / JNE	ZF=1, 顺序执行 ZF=0, 转移至目标地址
JP / JPE	PF=1, 转移至目标地址 PF=0, 顺序执行
JNP / JPO	PF=1, 顺序执行 PF=0, 转移至目标地址

图 7.2.1 单标志位指令的助记符与功能

## (2) JCXZ 指令

JCXZ 指令为一种单条件转移指令，即判断条件仅有一个，但 JCXZ 指令所判断的条件不是标志位的状态，而是 CX 寄存器中的数据，若  $(CX) = 0$  则转移至目标地址，若  $(CX) \neq 0$  则顺序执行。在 8086/8088 指令系统中，该指令是唯一不以标志位为判断条件的条件转移指令。该指令的用途是配合循环控制指令形成计数循环结构，在进入循环体前用于判断计数器 CX 中的数据是否已经为零，如果已为零则不进入循环体，直接转移至循环出口。请读者结合后面将要学习到的循环控制指令来理解。

## (3) 无符号数条件转移指令

无符号数条件转移指令的功能可以这样解释，将最近一条执行的 CMP（或 SUB）指令中两个操作数解释为无符号数编码，以 CMP 指令所影响的 CF、ZF 标志位来综合判断被减数与减数的大小关系，从而根据判断结果来决定是否实施流程转移。关于使用 CMP 指令判断无符号数大小关系的原理请参见关于第五章关于 CMP 指令的说明。读者应注意，无符号数条件转移指令没有二意性，它固定地将 CMP 指令的操作数解释为无符号数。因此，在程序设计中，需要明确自己所使用的编码是无符号数之后，才能选择无符号数条件转移指令来实现分支或循环结构。无符号数条件转移指令的助记符与功能解释如图 7.2.2 所示。图中的“A 数据”表示被减数，“B”数据表示减数。



指令助记符	指令功能
JA / JNBE	若 $A > B$ ，转移至目标地址 若 $A \leq B$ ，顺序执行
JAE / JNB	若 $A \geq B$ ，转移至目标地址 若 $A < B$ ，顺序执行
JB / JNAE	若 $A < B$ ，转移至目标地址 若 $A \geq B$ ，顺序执行
JBE / JNA	若 $A \leq B$ ，转移至目标地址 若 $A > B$ ，顺序执行

图 7.2.2 无符号数条件转移指令的助记符与功能

#### (4) 带符号数条件转移指令

带符号数条件转移指令的功能可以这样解释，将最近一条执行的 CMP（或 SUB）指令中两个操作数解释为补码，以 CMP 指令所影响的 OF、SF、ZF 标志位来综合判断被减数与减数的大小关系，从而根据判断结果来决定是否实施流程转移。关于使用 CMP 指令判断补码大小关系的原理请参见关于第五章关于 CMP 指令的说明。读者应注意，带符号数条件转移指令没有二意性，它固定地将 CMP 指令的操作数解释为补码。因此，在程序设计中，需要明确自己所使用的编码是补码之后，才能选择带符号数条件转移指令来实现分支或循环结构。带符号数条件转移指令的助记符与功能解释如图 7.2.3 所示。图中的“A 数据”表示被减数，“B”数据表示减数。

指令助记符	指令功能
JG / JNLE	若 $A > B$ ，转移至目标地址 若 $A \leq B$ ，顺序执行
JGE / JNL	若 $A \geq B$ ，转移至目标地址 若 $A < B$ ，顺序执行
JL / JNGE	若 $A < B$ ，转移至目标地址 若 $A \geq B$ ，顺序执行
JLE / JNG	若 $A \leq B$ ，转移至目标地址 若 $A > B$ ，顺序执行

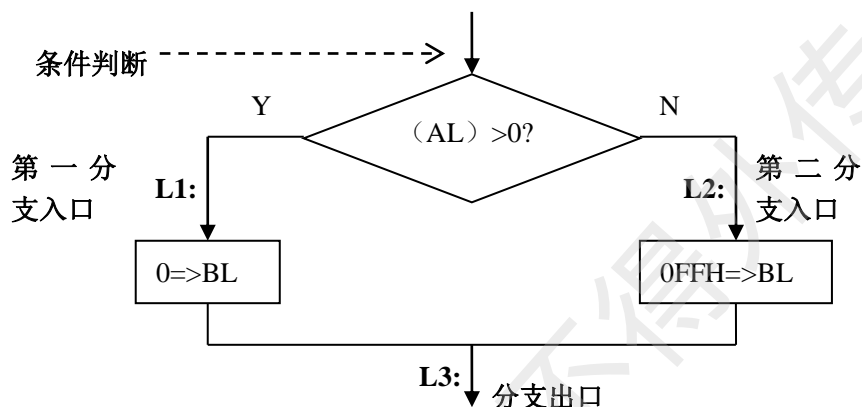
图 7.2.3 带符号数条件转移指令的助记符与功能

## 7.3 分支程序设计

在这一小节中，我们将使用前面学到的条件转移指令、无条件转移指令来进行分支结构程序设计。首先，我们先通过一个简单的例子来说明分支结构程序的组成。

**例 7.3.1** 试设计一个程序片段，实现如下功能：将 (AL) 理解为补码，若 (AL) > 0，则将 BL 寄存器清 0；否则，将 BL 寄存器置为全 1。

流程图如下：



程序片段如下：

```

      CMP  AL, 0          ; 条件判断
      JG   L1
      JMP  L2
L1:   MOV  BL, 0          ; 第一分支入口
      JMP  L3             ; 转移至分支结构出口
L2:   MOV  BL, 0FFH       ; 第二分支入口
L3:   .....             ; 分支结构出口
  
```

在上例的流程图中，我们可以观察到，**基本的双分支结构程序由条件判断模块、第一分支模块、第二分支模块组成**。条件判断模块对应到源程序中，就是最前面的三条指令，包括 CMP、JG、JMP，这三条指令用于对条件实施判断，根据条件决定程序流程转移到哪一个分支模块的入口。在流程图中，我们可以看到，分支结构是一个平面结构，第一分支模块与第二分支模块是平行的，没有顺序概念，但在源程序中，两个分支模块必然有一个在前，有一

个在后，这是源程序的直线形结构限定的。在源程序中，L1 标号对应第一分支模块的入口，L2 标号对应第二分支模块的入口，L3 标号则对应分支结构的出口。分支结构程序中有两处需要注意：

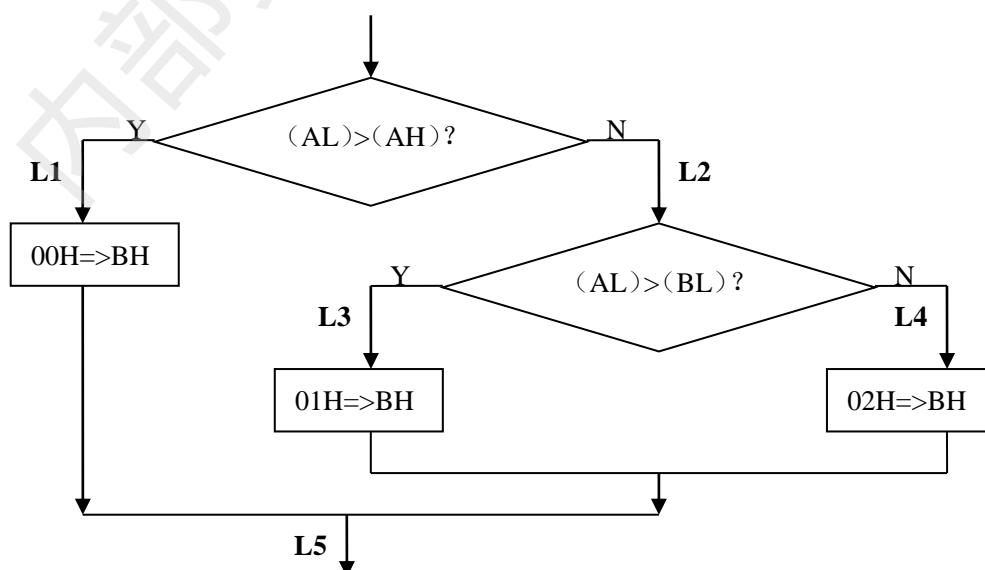
(1) 条件判断模块中 JG 指令转移到 L1 标号，即第一分支入口，而 JMP 指令转移到 L2 标号，即第二分支入口，这样的设计是为了使条件转移指令转移较小的字节距离，因为它的转移范围相对较小。

(2) 第一分支模块的最后有一条 JMP 指令，它的功能非常重要，用于跳过第二分支模块中的指令，转移至分支结构的出口。换言之，**分支结构程序中必须使用无条件转移指令才完整，这也是为什么无条件转移指令仍然在指令系统中存在的重要原因。**事实上，高级语言中的 GOTO 语句在结构化程序设计中被排斥，并不表示它的功能完全被丢弃掉了，它的功能只是被限制在分支、循环结构中，高级语言的程序员看不见它们罢了，在机器底层，它们以无条件转移指令的形式继续存在。

现在，我们已经明确了分支结构程序的组成，那么，我们应该怎样来设计分支结构程序呢？下面，我们再通过一个示例来说明分支结构程序的设计方法。

**例 7.3.2** 试设计一个程序片段，将 (AL)、(AH)、(BL) 中的数据解释为无符号数，如果 (AL) > (AH)，则将数值 00H 传送至 BH 保存；否则，如果 (AL) > (BL)，则将数值 01H 传送至 BH 保存；如果前面两个条件均不满足，则将数值 02H 传送至 BH 保存。

首先，应作流程图，如下所示：



作好流程图后，我们可以发现这是一个嵌套的分支结构。**按照从上至下、从左至右的顺序给各分支结构的入口、出口分配标号**。由于内层分支结构的出口与外层分支结构的出口间无任何功能模块存在，说明它们在源程序中位置相同，只给它们分配了一个标号 L5。接下来，在流程图上按照从上至下、从左至右的顺序梳理程序结构，同时按照条件判断模块、第一分支模块、第二分支模块的顺序将流程图中每个模块都转换为相应的源代码。这样设计出来的源代码如下：

```
        CMP  AL, AH
        JA   L1
        JMP  L2
L1:      MOV  BH, 0
        JMP  L5
L2:      CMP  AL, BL
        JA   L3
        JMP  L4
L3:      MOV  BH, 01H
        JMP  L5
L4:      MOV  BH, 02H
L5:      .....
```

在上例中，我们了解了分支结构程序的设计方法。并且，在例中我们可以看到，在源程序中的标号排列比较整齐，且程序结构也比较清晰。

实际上，循环结构也是双分支结构的一种特例，因为循环控制也具有两种可能的流程取向，一种是循环条件满足，程序流程进入循环入口继续执行循环结构；另一种是循环条件不满足，程序流程转移至循环出口，退出循环结构。因此，循环结构程序也可以使用条件转移指令、无条件转移指令相结合来实现，我们下面通过一个例子来说明。

**例 7.3.3** 假设有一个字类型的变量 VAR1，其中的数据解释为无符号数，试设计一个完整程序，不使用除法指令，求该无符号数除以 3 后的商和余数，商保存在字类型变量 RES1 中，余数保存在字类型变量 RES2 中。

如果不使用除法指令，那么，我们可以使用减法运算来完成除法运算。在循环结构程序中，我们使用 VAR1 中数据不断的减去 3，每减 1 次，就将计数加 1，直到不够减为止。最后，计数值即为商，而循环减 3 剩余的数值则是余数。流程图请读者自己构造，这里仅给出源程序如下：

```

DATA SEGMENT
VAR1 DW 6450
RES1 DW 0
RES2 DW 0
DATA ENDS

CODE SEGMENT
ASSUME CS:CODE, DS:DATA
BEGIN: MOV AX, DATA
        MOV DS, AX
        MOV AX, VAR1
        MOV BX, 0
L1:     SUB AX, 3
        JAE L2
        JMP L3           ; 转移至循环体出口
L2:     INC BX           ; 计数
        JMP L1           ; 转移至循环体入口
L3:     MOV RES1, BX     ; 保存商
        MOV RES2, AX     ; 保存余数
        MOV AH, 4CH
        INT 21H
CODE ENDS
END BEGIN

```

读者可以观察到，本例中的循环结构等价于高级语言源程序中的 **While** 循环结构。

## 7. 4 循环控制指令

通过上一小节的学习，我们已经知道，循环结构也可以通过条件转移指令、无条件转移指令相结合来实现。那么，如果没有专门的循环控制指令，我们也能实现循环结构吗？回答是肯定的，即便只使用我们前面学到的条件转移指令、无条件转移指令，我们也能实现循环结构程序。本小节所介绍的循环控制指令，是专门针对高级语言程序结构中 FOR 循环结构（计数循环）而设计的，可以使程序员更方便的实现计数循环结构。下面，我们先介绍各循环控制指令的功能。由于循环控制指令与前面介绍的无条件、条件转移指令类似，它们可能会判断标志位，但不会影响标志位，因此标志位影响在本小节就不再作分析。与条件转移指令类似，循环控制指令只能实现段内短转移，向低地址端最大转移-128 字节，向高地址端最大转移 127 字节。因此，如果循环结构过于庞大，则需要配合使用无条件转移指令来实现循环结构程序。

### (1) LOOP 指令

指令格式：LOOP NEAR 类型的段内标号

指令功能：首先  $(CX) - 1 \Rightarrow CX$ ，该减法不会影响标志位，然后进行判断，若  $(CX) \neq 0$ ，则转移至目标地址，否则顺序执行。

### (2) LOOPZ / LOOPE 指令

指令格式：LOOPZ NEAR 类型的段内标号

指令功能：首先  $(CX) - 1 \Rightarrow CX$ ，该减法不会影响标志位，然后进行判断，若  $(CX) \neq 0$  并且  $ZF = 1$ ，则转移至目标地址，否则顺序执行。

### (3) LOOPNZ / LOOPNE 指令

指令格式：LOOPNZ NEAR 类型的段内标号

指令功能：首先  $(CX) - 1 \Rightarrow CX$ ，该减法不会影响标志位，然后进行判断，若  $(CX) \neq 0$  并且  $ZF = 0$ ，则转移至目标地址，否则顺序执行。

## 7. 5 循环程序设计

循环结构程序是分支结构程序的一种特例，因此，程序设计方法也是相似的。在本小节中，我们不再讨论程序设计方法，而是通过一些示例来让读者体会循环结构程序的设计过程。

**例 7. 5. 1** 假设有一个字节数组 `ARRY`，数组中的数据解释为学生成绩，为无符号数。试设计一个完整程序，统计这些成绩中大于等于 60 分的人数，统计结果保存在 `NUM` 变量中。

由于数组中每个数据都要判断一次，所以此程序为单重循环结构，并且是计数循环。源程序设计如下：

```
DATA SEGMENT
ARRY DB 75, 82, 64, 50, 70, 45, 90
LENGTH EQU $ - ARRY
NUM DB ?
DATA ENDS

CODE SEGMENT
ASSUME CS:CODE, DS:DATA
BEGIN: MOV AX, DATA
        MOV DS, AX
        XOR AL, AL ; 统计计数器清 0
        LEA BX, ARRY ; BX 指向 ARRY 首地址
        MOV CX, LENGTH ; 初始化循环计数器
        JCXZ L1 ; 若计数值为 0 则退出循环
L2: CMP [BX], 60
        JAE L3
        JMP L4
L3: INC AL ; 若成绩大于等于 60，则计数加 1
L4: INC BX ; BX 指向 ARRY 数组中下一字节
```

```

        LOOP L2                                ; (CX) 减 1 后不为 0 则转移至循环入口
L1:     MOV    NUM, AL                          ; 保存统计结果

        MOV    AH, 4CH

        INT    21H

CODE    ENDS

END     BEGIN

```

**例 7.5.2** 假设有一字符串 STR1，试编写程序查找该字符串中第一个‘A’字符出现的位置，将其位置编号保存在 POS1 字节变量中，若未找到，则将 0FFH 保存至 POS1 变量。

此程序为单重循环结构，可以考虑使用 LOOPNZ 循环控制指令来提高程序设计与执行效率，源程序设计如下：

```

DATA    SEGMENT
STR1    DB    'BBCABCDAC'
LEN      EQU    $ - STR1
POS1    DB    ?
DATA    ENDS

CODE    SEGMENT
ASSUME CS:CODE, DS:DATA
BEGIN:  MOV    AX, DATA
        MOV    DS, AX

        MOV    BX, -1                        ; BX 指向 STR1 第 0 个字符的前一个位置

        MOV    CX, LEN

        JCXZ   L1                            ; 若字符串长度为 0 则退出循环

L2:     INC     BX                            ; BX 指向下一个字符位置

        CMP    STR1[BX], 'A'

        LOOPNZ L2                            ; 字符串查找完毕或找到 'A' 字符时退出循环

L1:     JZ      L3

```



```

        JMP     L4
L3:     MOV     POS1,BL           ; 若找到 'A' 字符, 则保存其位置编号
        JMP     L5
L4:     MOV     POS1,0FFH       ; 若未找到, 则保存 0FFH
L5:     MOV     AH,4CH
        INT     21H
CODE    ENDS
END     BEGIN

```

**例 7. 5. 3** 假设有一字节类型的数组 `ARRY1`, 试编写程序统计 `ARRY1` 数组的各字节数据中含 “1” 数据位的个数, 统计结果保存在 `RES1` 数组中。

此程序为双重循环结构, 因为访问数组中每个字节需要一层计数循环结构, 统计每个字节中 “1” 数据位的个数也需要一层计数循环结构。LOOP 指令使用的计数器只有 `CX` 这一个, 如果出现多层计数循环结构, 要么使用堆栈对各层计数值进行保护和恢复, 要么使用别的寄存器作为计数器。如果不使用 `CX` 作为计数器, 则该层计数循环不能使用 LOOP 指令实现, 而需要换用其它条件转移指令。在本例中, 我们使用堆栈来保护、恢复各层计数值, 两层循环均使用 LOOP 指令来实现。源程序设计如下:

```

DATA    SEGMENT
ARRY1   DB    34H, 37H, 6EH, 0C2H, 55H, 16H, 9FH
LEN      EQU   $ - ARRY1
RES1    DB     LEN DUP(0)
DATA    ENDS

STACK1  SEGMENT  STACK
        DW     40H DUP(0)
STACK1  ENDS

CODE    SEGMENT

```

```
ASSUME  CS:CODE, DS:DATA, SS:STACK1

BEGIN:  MOV  AX, DATA
        MOV  DS, AX
        XOR  BX, BX                ; BX 指向 ARRAY1、RES1 第 0 个字节
        MOV  CX, LEN              ; 初始化外层循环计数器
        JCXZ  EXIT1

LOP1:   PUSH  CX                  ; 保护外层计数值
        MOV  CX, 8                ; 初始化内层循环计数器
        MOV  AL, ARRAY1[BX]

LOP2:   SHR  AL, 1                ; 逐位移至 CF
        JC   L1
        JMP  L3

L1:     INC  RES1[BX]             ; 如果是“1”数据位，则统计计数加 1

L3:     LOOP LOP2                ; 内层循环控制
        POP  CX                  ; 恢复外层计数值
        INC  BX                  ; BX 指向数组中下一个字节
        LOOP LOP1                ; 外层循环控制
        MOV  AH, 4CH
        INT  21H

CODE  ENDS

END  BEGIN
```

## 习题 7

1. 简述无条件转移指令在指令系统中存在的必要性。
2. 简述条件转移指令的分类，以及各类条件转移指令的功能。
3. 按要求分析下面程序片段的执行结果。

```
        MOV  AL, -5
        MOV  BL, -3
        CMP  AL, BL
        JGE  L1
        JMP  L2
L1:     MOV  AH, 00H
        JMP  L3
L2:     MOV  AH, 01H
L3:     .....
```

执行该程序片段后，(AH) = ?

4. 按要求分析下面程序片段的执行结果。

```
        MOV  AL, 6CH
        MOV  CX, 8
        XOR  AH, AH
L1:     ROL  AL, 1
        ADC  AH, 0
        LOOP L1
```

执行该程序片段后，(AH) = ?

5. 试分析并说明下面程序片段的功能。

```
.....
ARY1  DB  35, 23, -6, -21, 90, 65, 73, -54, -7
LEN   EQU  $ - ARY1
.....
        LEA  BX, ARY1
        XOR  AL, AL
```

```
        MOV  CX, LEN
        JCXZ  EX1
L1:     TEST  [BX], 01H
        JNZ   L2
        JMP   L3
L2:     INC   AL
L3:     LOOP  L1
EX1:     .....
```

6. 假设有一字符串 **STR**，试编写一个程序查找 **STR** 字符串中第一个非空格字符的位置，如果找到则将字符位置保存在字节变量 **POS1** 中，如果没有找到则将 **0FFH** 保存至 **POS1** 中。
7. 假设有一字类型的补码数组 **ARY1**，试编写一个程序查找 **ARY1** 数组中的最大值，并将查找结果保存在字类型变量 **RES1** 中。
8. 假设有一字节类型的补码数组 **DAT1**，试编写一个程序统计数组 **DAT1** 中正数（包括 0）和负数分别的数量，正数统计结果保存在字节变量 **RES1** 中，负数统计结果保存在字节变量 **RES2** 中。
9. 假设有两个长度为 8 个字的补码，分别保存在字类型数组 **DA1** 和 **DA2** 中，低地址保存低位，试设计一个程序实现两个长补码的减法，减法结果保存在字类型数组 **RES1** 中，是否溢出用 **OF** 标志反映。
10. 假设有一个长度为 4 个字的无符号数，保存在字类型数组 **DA1** 中，低地址保存低位，试设计一个程序实现这个长无符号数除以 8 的功能，运算结果仍然保存在数组 **DA1** 中。

## 第 8 章 子程序设计 with 系统调用

### 8. 1 子程序调用与返回指令

我们都知道，在高级语言中，子程序的执行需要子程序的调用和返回两个关键步骤，才能实现主程序到子程序、子程序到主程序的流程转移。而这两个关键步骤在机器底层则是分别通过子程序调用指令、子程序返回指令来实现的。子程序调用与返回指令与前面介绍的无条件、条件转移指令、循环控制指令具有类似特征，它们都能实现程序流程的转移，属于程序转移类指令。但与通常程序转移类指令不同，子程序调用指令在实施流程转移前，会使用入栈操作来保存返回点地址，而子程序返回指令则会使用出栈操作恢复返回点地址，并以返回点地址为目标地址实施流程转移。与其它程序转移类指令相似，子程序调用与返回指令不影响标志位，因此在后面对指令的说明中，我们不再针对标志位影响进行说明。与其它程序转移类指令相比较，子程序调用与返回指令更接近于无条件转移指令，因为它们都不会对标志位、寄存器的进行判断，而是无条件的实施转移，不同之处是子程序调用与返回指令会使用堆栈来保存和恢复返回点。与无条件转移指令相似，按照目标地址的获取方式，子程序调用指令可分为段内直接调用、段内间接调用、段间直接调用、段间间接调用四种类型，而相应于段内调用、段间调用，子程序返回指令可以分为段内返回、段间返回两种类型。下面将分别介绍各类子程序调用与返回指令。

#### (1) 段内直接调用指令

指令格式: `CALL PROC_NAME`

`CALL NEAR PTR PROC_NAME`

第一种指令格式中，`PROC_NAME` 为 `NEAR` 类型的子程序，并且与 `CALL` 指令位于同一个代码段；第二种格式中 `PROC_NAME` 既可以为 `NEAR` 类型，也可以为 `FAR` 类型，但它必须与 `CALL` 指令位于同一个代码段，`NEAR PTR` 将它在当前指令中暂时转换为 `NEAR` 类型。

指令功能:

1)  $(SP) - 2 \Rightarrow SP$ ，`SP` 指向空栈顶；

2)  $(IP) \Rightarrow (SP)$ ，CALL 指令后面一条指令的偏移量送入栈顶保存；

3)  $PROC\_NAME$  偏移量  $\Rightarrow IP$ ，将  $PROC\_NAME$  符号位移量（汇编过程中会被替换为相应的数值位移量）作为子程序入口地址的偏移量送入  $IP$  保存，使程序流程向子程序入口转移。

## （2）段内间接调用指令

指令格式：CALL 16 位寄存器名称

CALL 字内存单元的逻辑地址

指令功能：

1)  $(SP) - 2 \Rightarrow SP$ ， $SP$  指向空栈顶；

2)  $(IP) \Rightarrow (SP)$ ，CALL 指令后面一条指令的偏移量送入栈顶保存；

3) (16位寄存器或字内存单元)  $\Rightarrow IP$ ，将 16 位寄存器或字内存单元中的数据作为子程序入口地址的偏移量送入  $IP$  保存，使程序流程向子程序入口转移。

例 8. 1. 1 段内间接调用指令的示例如下。

CALL BX

CALL 0100H[BX]

CALL 0020H[BX][SI]

## （3）不带参数的段内返回指令

指令格式：RET

RETN

第一种格式在 NEAR 类型的子程序中使用，在汇编过程中汇编程序会自动将它解释为第二

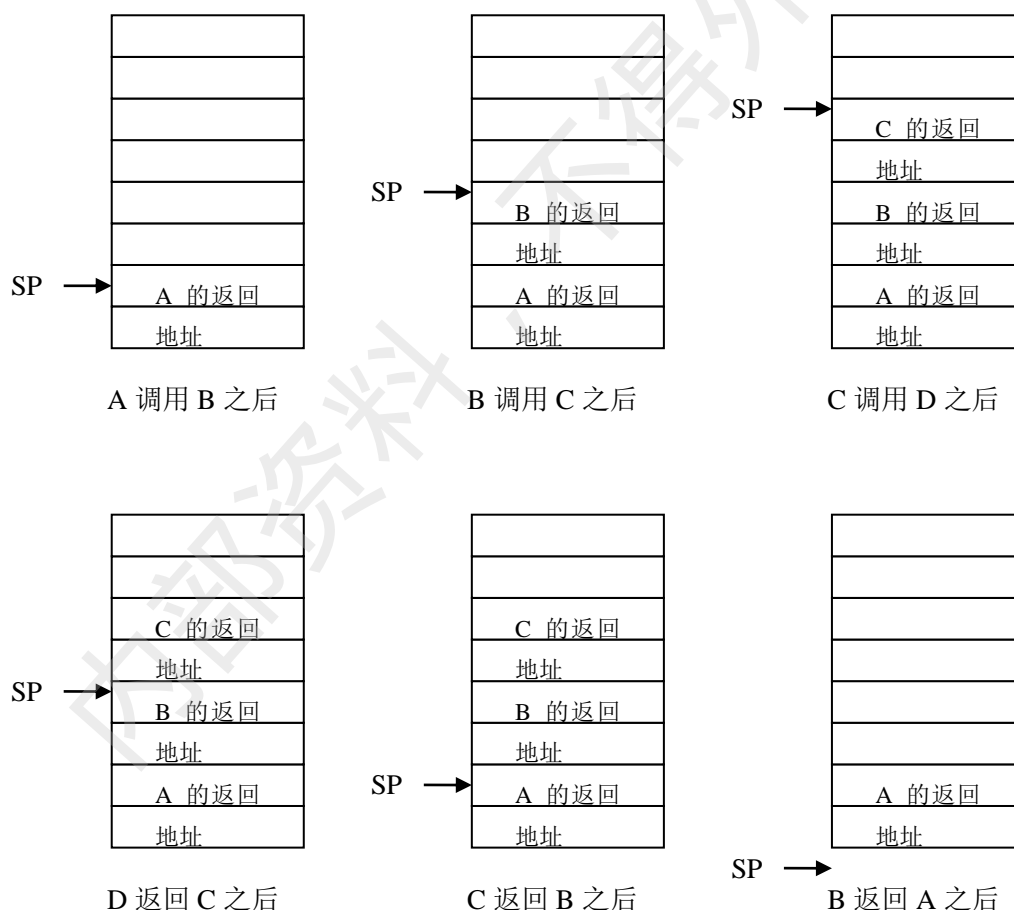
种格式；第二种格式明确指出返回类型为段内返回。

指令功能：

1)  $((SP)) \Rightarrow IP$ ，栈顶单元内容送 IP 保存，使程序流程转向主程序返回点；

2)  $(SP) + 2 \Rightarrow SP$ ，栈顶单元出栈

**例 8.1.2** 假设主程序 A 在执行过程中调用了子程序 B，而子程序 B 在执行过程中调用了子程序 C，子程序 C 又调用了子程序 D，并且假设所有调用与返回都是段内调用与返回，那么，执行各子程序调用、返回指令时，堆栈中的变化如下所示（每个方格表示一个字节）。



在本例中，请读者注意体会使用堆栈保存和恢复返回地址的原因。因为子程序的调用与返回遵循后调用先返回的原则，与堆栈结构后进先出的操作原则一致，因此使用堆栈来保存和恢复返回地址是必要的。

#### (4) 带参数的段内返回指令

指令格式: RET N

RETN N

第一种格式在 NEAR 类型的子程序中使用, 在汇编过程中汇编程序会自动将它解释为第二种格式; 第二种格式明确指出返回类型为段内返回。两种格式中的参数 N 都为整数, 并且必须是偶数。

指令功能:

- 1)  $((SP)) \Rightarrow IP$ , 栈顶单元内容送 IP 保存, 使程序流程转向主程序返回点;
- 2)  $(SP) + 2 \Rightarrow SP$ , 栈顶单元出栈
- 3)  $(SP) + N \Rightarrow SP$ , 从栈顶直接出栈  $N/2$  个字, 此功能用于清除主程序在调用子程序前通过堆栈传递的  $N/2$  个入口参数 (读者可在后面介绍的子程序设计中深刻体会这一功能的用途)。

#### (5) 段间直接调用指令

指令格式: CALL PROC\_NAME

CALL FAR PTR PROC\_NAME

第一种格式中, PROC\_NAME 为 FAR 类型的子程序; 第二种格式中, PROC\_NAME 既可为 NEAR 类型也可为 FAR 类型子程序, FAR PTR 将其在当前指令中的类型暂时指定的 FAR。

指令功能:

- 1)  $(SP) - 2 \Rightarrow SP$ , SP 指向空栈顶;
- 2)  $(IP) \Rightarrow (SP)$ , CALL 指令后面一条指令的偏移量送入栈顶保存;
- 3)  $(SP) - 2 \Rightarrow SP$ , SP 指向空栈顶;
- 4)  $(CS) \Rightarrow (SP)$ , CALL 指令后面一条指令的段基值送入栈顶保存;



5)  $PROC\_NAME$  偏移量  $\Rightarrow IP$ ,  $PROC\_NAME$  段基值  $\Rightarrow CS$ , 将  $PROC\_NAME$  的逻辑地址作为子程序入口地址, 偏移量送入  $IP$  保存, 段基值送入  $CS$  保存, 使程序流程向子程序入口转移。

#### (6) 段间间接调用指令

**指令格式:**  $CALL$  双字内存单元的逻辑地址

**指令功能:**

- 1)  $(SP) - 2 \Rightarrow SP$ ,  $SP$  指向空栈顶;
- 2)  $(IP) \Rightarrow (SP)$ ,  $CALL$  指令后面一条指令的偏移量送入栈顶保存;
- 3)  $(SP) - 2 \Rightarrow SP$ ,  $SP$  指向空栈顶;
- 4)  $(CS) \Rightarrow (SP)$ ,  $CALL$  指令后面一条指令的段基值送入栈顶保存;
- 5) (双字内存单元的低地址字)  $\Rightarrow IP$ , (双字内存单元的高地址字)  $\Rightarrow CS$ , 将指定的双字内存单元中的低地址字解释为偏移量送入  $IP$  保存, 高地址字解释为段基值送入  $CS$  保存, 使程序流程向子程序入口转移。

#### (7) 不带参数的段间返回指令

**指令格式:**  $RET$

$RETF$

第一种格式在  $FAR$  类型的子程序中使用, 在汇编过程中汇编程序会自动将它解释为第二种格式; 第二种格式明确指出返回类型为段间返回。

**指令功能:**

- 1)  $((SP)) \Rightarrow IP$ , 栈顶单元内容送  $IP$  保存;
- 2)  $(SP) + 2 \Rightarrow SP$ , 栈顶单元出栈;

3)  $((SP)) \Rightarrow CS$ ，栈顶单元内容送 CS 保存，使程序流程转向主程序返回点；

4)  $(SP) + 2 \Rightarrow SP$ ，栈顶单元出栈；

读者应注意，与段内调用、返回不同，段间调用入栈时是入栈段基值、偏移量两个字，相应的，段间返回时出栈也是两个字。读者可参考例 8.1.2 自行构造段间调用、返回的示例，并观察堆栈中的变化。

### (7) 带参数的段间返回指令

指令格式：RET N

RETF N

第一种格式在 FAR 类型的子程序中使用，在汇编过程中汇编程序会自动将它解释为第二种格式；第二种格式明确指出返回类型为段间返回。两种格式中的参数 N 都为整数，并且必须是偶数。

指令功能：

1)  $((SP)) \Rightarrow IP$ ，栈顶单元内容送 IP 保存；

2)  $(SP) + 2 \Rightarrow SP$ ，栈顶单元出栈；

3)  $((SP)) \Rightarrow CS$ ，栈顶单元内容送 CS 保存，使程序流程转向主程序返回点；

4)  $(SP) + 2 \Rightarrow SP$ ，栈顶单元出栈；

5)  $(SP) + N \Rightarrow SP$ ，从栈顶直接出栈  $N/2$  个字，此功能用于清除主程序在调用子程序前通过堆栈传递的  $N/2$  个入口参数（读者可在后面介绍的子程序设计中深刻体会这一功能的用途）。

这里，我们给出一个简单的总结。子程序调用、返回指令都必须使用堆栈来保存和恢复返回地址，堆栈操作对于子程序机制的实现是必不可少的。因此，CPU 中的堆栈机制是实现子程序调用与返回的硬件基础，此原理可以向任意型号的 CPU 作推广理解。

## 8. 2 子程序设计

### 8. 2. 1 子程序设计的一般规范

上一小节中，我们对子程序调用、返回指令做了较全面的说明。在本小节中，我们将说明如何使用指令与伪指令来构造完整的子程序，以及在子程序设计中需要遵循的一般规范。

#### (1) 子程序框架

在源程序中，子程序的框架如下所示。

```
PROC_NAME  PROC    [NEAR / FAR]
.....
RET
.....
PROC_NAME  ENDP
```

其中，PROC\_NAME 为程序员定义的子程序名称，PROC、ENDP 伪指令用于定义子程序，子程序的主体位于这两个伪指令语句之间，主要由指令序列构成。子程序中至少需要一条返回指令 RET，它不一定位于子程序语句的最后，但执行子程序时，最后一条被执行的指令必然是 RET 指令。PROC 伪指令后可以选择性的指定子程序的类型，有 NEAR 和 FAR 两种类型，如果不加以指定，则汇编程序默认解释为 NEAR 类型。源程序中 RET 指令的具体格式由子程序的类型确定，如果子程序的类型为 NEAR，那么汇编程序在汇编过程中会将 RET 指令解释为 RETN 指令；如果子程序的类型为 FAR，那么汇编程序在汇编过程中会将 RET 指令解释为 RETF 指令。在源程序中，当使用 CALL 指令对子程序进行调用时，如果不使用 PTR 作类型转换，那么针对 NEAR 类型的子程序，汇编程序会将相应的 CALL 指令解释为段内直接调用指令；针对 FAR 类型的子程序，汇编程序会将相应的 CALL 指令解释为段间直接调用指令。在程序设计中，读者**需要注意针对同一子程序的 CALL 指令与 RET 指令搭配使用，要么同为段内操作，要么同为段间操作**。调用子程序时最好不作类型转换，而使用子程序类型来作统一解释。如果 CALL 指令与 RET 指令搭配不当，将导致子程序返回时不能转移到正确的返回点，同时堆栈操作也会出现错误。

#### (2) 子程序的独立性与通用性

在子程序设计中，需要考查待设计的功能是否适合编制为子程序。一般而言，子程序具有独立性与通用性。独立性是指子程序的功能相对独立，例如，在指定字符串中查找指定字符所在位置、完成两个指定长操作数的加减法等等，如果一个子程序的独立性差，那么就涉及到大量的参数传递，使程序设计变得非常复杂；通用性是指子程序的代码可重用，因为子程序设计的目的就是提高代码重用率，减少代码的重复编写。因此，当某个功能模块具有较好的独立性，且经常会被使用时，我们认为它适合设计为子程序形式。当然，子程序设计是否良好，取决于设计者对功能模块的划分经验，这是需要长时间实践来逐步积累的。

### (3) 子程序的参数传递

无论是高级语言程序设计，还是汇编语言程序设计，在多数情况下，主程序在调用子程序时都需要向子程序传递入口参数，而子程序也需要向主程序传递出口参数。在高级语言中，参数传递是简单的，因为程序员仅需要考虑参数的类型，不需要考虑参数传递的方式和过程，甚至在多数情况下都不考虑参数传递的顺序，因为这些复杂的功能是由高级语言的编译系统帮助程序员完成的。在汇编语言程序设计中，程序员必须考虑参数的类型、传递方式、传递顺序，并且在程序设计中加以实现。由于高级语言中的子程序参数传递方式在机器底层就是汇编语言中使用的参数传递方式，因此通过对子程序参数传递方式的学习，读者能够深入理解参数传递在机器底层的实现原理。

在汇编语言程序设计中，参数传递方式主要有三种：**寄存器传递方式、堆栈传递方式、数据区传递方式**。除寄存器传递方式外，其余两种参数传递方式都需要使用内部存储器。虽然寄存器传递方式最快捷，但由于 CPU 中寄存器的数量有限，在高级语言实现的程序中所使用的参数传递方式多为堆栈传递方式与数据区传递方式。**在汇编语言程序中，程序员必须在主程序与子程序间建立参数传递方式的约定**，主程序按照约定向子程序传递入口参数，并按照约定取得出口参数；子程序按照约定取得入口参数，并按照约定向主程序传递出口参数。**约定内容包括参数类型（字节、字等）、参数传递方式（寄存器、堆栈、数据区）、参数顺序（参数在堆栈、数据区中的顺序）**。读者在进行程序设计时需要注意，主程序与子程序必须按照约定进行参数传递，否则必然导致参数传递的混乱，导致程序执行中的逻辑错误。在后面给出的子程序设计示例中，请读者注意体会参数传递的约定。

### (4) 子程序的现场保护

主程序调用子程序时，寄存器、标志位当时的状态我们称为 **CPU 现场**。在高级语言程序设计中，程序员并不会考虑到 CPU 现场的保护，因为这些操作是由编译系统生成的代码帮助程序员完成的，但是这些操作在机器底层是客观存在的，在汇编语言程序设计中，我们不仅不能对它视而不见，而且需要自己在程序设计中来完成它。子程序与主程序都可能会使用 CPU 中的寄存器，也都有可能因为完成某些运算而影响标志位。但是，**我们通常希望所设计的子程序对主程序而言具有透明性，即子程序返回后，在主程序看来，除约定的参数传递外，CPU 现场是没有改变的。**因为如果子程序任意破坏 CPU 现场，而主程序对子程序的造成的现场破坏置之不理，那么主程序认为没有变化的寄存器内容、标志位状态实际上很可能已经改变，必然造成主程序执行的逻辑错误。

因此，现场保护与现场恢复对于子程序调用是必要的，那么我们如何来实施现场保护和恢复呢？一般习惯的方法是使用 **PUSH** 指令将现场保护到堆栈中，使用 **POP** 指令将堆栈中的预先保护的现场加以恢复。理论上，在主程序和子程序中实施现场保护和恢复都是可行的。但如果在主程序中实施现场保护，那么在每个子程序调用指令的位置前都要增加 **PUSH** 指令来实施现场保护，在调用指令位置后都要增加 **POP** 指令来实施现场恢复，使代码设计出现严重的冗余。因此，现场保护与恢复通常是在子程序中实现的，当程序流程进入子程序后，还未执行子程序主体前，使用 **PUSH** 指令保护现场，在子程序主体执行完毕后，返回主程序之前，使用 **POP** 指令恢复现场。并且，由于子程序的设计者更清楚子程序会破坏 CPU 现场中的哪一个部分，因此所施加的现场保护和恢复往往是局部的。读者可在后面给出的子程序设计示例中进一步体会现场保护与恢复的方法。

## 8. 2. 2 子程序设计示例

在本小节中，我们通过一个子程序设计的示例来说明子程序的设计方法、子程序参数传递的不同方式、子程序中的现场保护。

**例 8. 2. 1** 试设计一程序，完成两个长补码的加法，其中两个长补码的长度是相同的，并且采用低地址存放低位，高地址存放高位的方式存储。加法功能使用子程序实现，两个长补码的起始地址、长补码的长度（以字为单位）、运算结果的起始地址均由主程序以入口参数的形式传递给子程序；运算是否溢出使用一个字节标志来表示，00H 表示运算无溢出，0FFH

表示运算溢出，子程序将此溢出标志以出口参数的形式传递给主程序。本例中参数传递方式采用寄存器传递方式。

### 1) 设计参数传递约定

题目已指明参数传递为寄存器传递方式，这里只需要具体约定寄存器与参数间的对应关系。在本例中，我们约定主程序通过 SI、DI 寄存器传递两个长补码的起始偏移量（认为两个长补码位于同一数据段，并且段基值保存到 DS 中），通过 CX 寄存器传递补码的长度，通过 BX 寄存器传递运算结果的起始偏移量（认为段基值已保存在 DS 中）；并且，我们约定子程序通过 DL 寄存器传递溢出标志。

### 2) 源程序设计

参数传递约定设计完成后，我们开始源程序设计，主程序与子程序的相应代码如下所示。

```
DATA SEGMENT
VAR1 DW 5482H, 669EH, 02C7H, 14B2H, 0C254H
VAR2 DW 8C2BH, 0C24CH, 0AB12H, 357AH, 41A5H
LEN EQU $-VAR2
SUM DW LEN DUP(0) ; 用于保存运算结果
OVR DB ? ; 用于保存溢出标志
DATA ENDS

STACK1 SEGMENT STACK ; 子程序设计必须使用堆栈段
DW 40H DUP(0)
STACK1 ENDS

CODE SEGMENT
ASSUME CS: CODE, DS: DATA, SS: STACK1
BEGIN: MOV AX, DATA
MOV DS, AX
LEA SI, VAR1 ; 传递入口参数
LEA DI, VAR2
```

```

MOV  CX, LEN
LEA  BX, SUM
CALL LONGADD                ; 调用子程序
MOV  OVR, DL                ; 保存出口参数
MOV  AH, 4CH
INT  21H

LONGADD PROC

    PUSHF                    ; FR、AX 入栈，保护 CPU 现场
    PUSH  AX
    CLC                      ; 清除 CF 标志，保证第一次执行 ADC 时等同于 ADD
L1:  MOV  AX, [SI]
      ADC  AX, [DI]          ; 完成当前字的加法，引入了上次加法的进位
      PUSHF                  ; 保护 OF 标志
      MOV  [BX], AX          ; 当前字的计算结果保存到指定位置
      ADD  SI, 2              ; 修改 SI、DI、BX，使它们指向下一个字
      ADD  DI, 2
      ADD  BX, 2
      POPF                    ; 恢复 OF 标志
      LOOP L1                ; 计数循环，完成所有字单元的加法后退出循环
      JO   L2                ; 判断最后一次运算是否溢出
      MOV  DL, 0              ; 无溢出则将 0 送至 DL 保存，传递出口参数
      JMP  L3
L2:  MOV  DL, 0FFH            ; 有溢出则将 0FFH 送至 DL 保存，传递出口参数
L3:  POP  AX                  ; 恢复 CPU 现场
      POPF
      RET                    ; 返回主程序

LONGADD ENDP

CODE  ENDS

END  BEGIN

```

读者在上例中应注意，首先，主程序向子程序传递入口参数、取得出口参数都是按照约定来

进行的；其次，读者应注意子程序中的现场保护与恢复，它们分别在子程序的首位部分用 PUSH 和 POP 指令来实现；第三，由于上例中采用寄存器来传递参数，当执行子程序时，各参数已在寄存器中就位，使我们在子程序中没有找到“取入口参数”的动作，但实际上入口参数在子程序中已经可以直接使用。

**例 8.2.2** 试设计一程序，完成例 8.2.1 中指定的功能，但本例中参数传递方式采用堆栈传递方式。

### 1) 设计参数传递约定

题目中已指定参数传递方式为堆栈传递方式，这里只需约定入口参数入栈的顺序与出口参数的出栈顺序。我们约定主程序按照被加数起始偏移量、加数起始偏移量、被加数与加数的长度、运算结果起始偏移量的顺序将入口参数入栈，子程序则按照相应顺序来获取入口参数；并且，我们约定子程序的出口参数覆盖最先入栈的入口参数，因为只有一个出口参数，返回主程序后，主程序只需将出口参数出栈即可。

### 2) 源程序设计

参数传递约定设计完成后，我们开始源程序设计，主程序与子程序的相应代码如下所示。

```
DATA SEGMENT
VAR1 DW 5482H, 669EH, 02C7H, 14B2H, 0C254H
VAR2 DW 8C2BH, 0C24CH, 0AB12H, 357AH, 41A5H
LEN EQU $-VAR2
SUM DW LEN DUP(0) ; 用于保存运算结果
OVR DB ? ; 用于保存溢出标志
DATA ENDS

STACK1 SEGMENT STACK ; 子程序设计必须使用堆栈段
DW 40H DUP(0)
STACK1 ENDS
```



CODE SEGMENT

ASSUME CS: CODE, DS: DATA, SS: STACK1

BEGIN: MOV AX, DATA

MOV DS, AX

LEA SI, VAR1 ; 通过堆栈传递入口参数

PUSH SI

LEA SI, VAR2

PUSH SI

MOV SI, LEN

PUSH SI

LEA SI, SUM

PUSH SI

CALL LONGADD ; 调用子程序

POP DX ; 出口参数出栈

MOV OVR, DL ; 保存出口参数

MOV AH, 4CH

INT 21H

LONGADD PROC

PUSH BP ; BP 作为现场加以保护

MOV BP, SP ; 获取当前栈顶位置

PUSHF ; 保护 CPU 现场

PUSH AX

PUSH BX

PUSH CX

PUSH SI

PUSH DI

MOV BX, [BP+4] ; 取得保存运算结果的起始偏移量

MOV CX, [BP+6] ; 取得被加数、加数得长度

MOV DI, [BP+8] ; 取得加数的起始偏移量

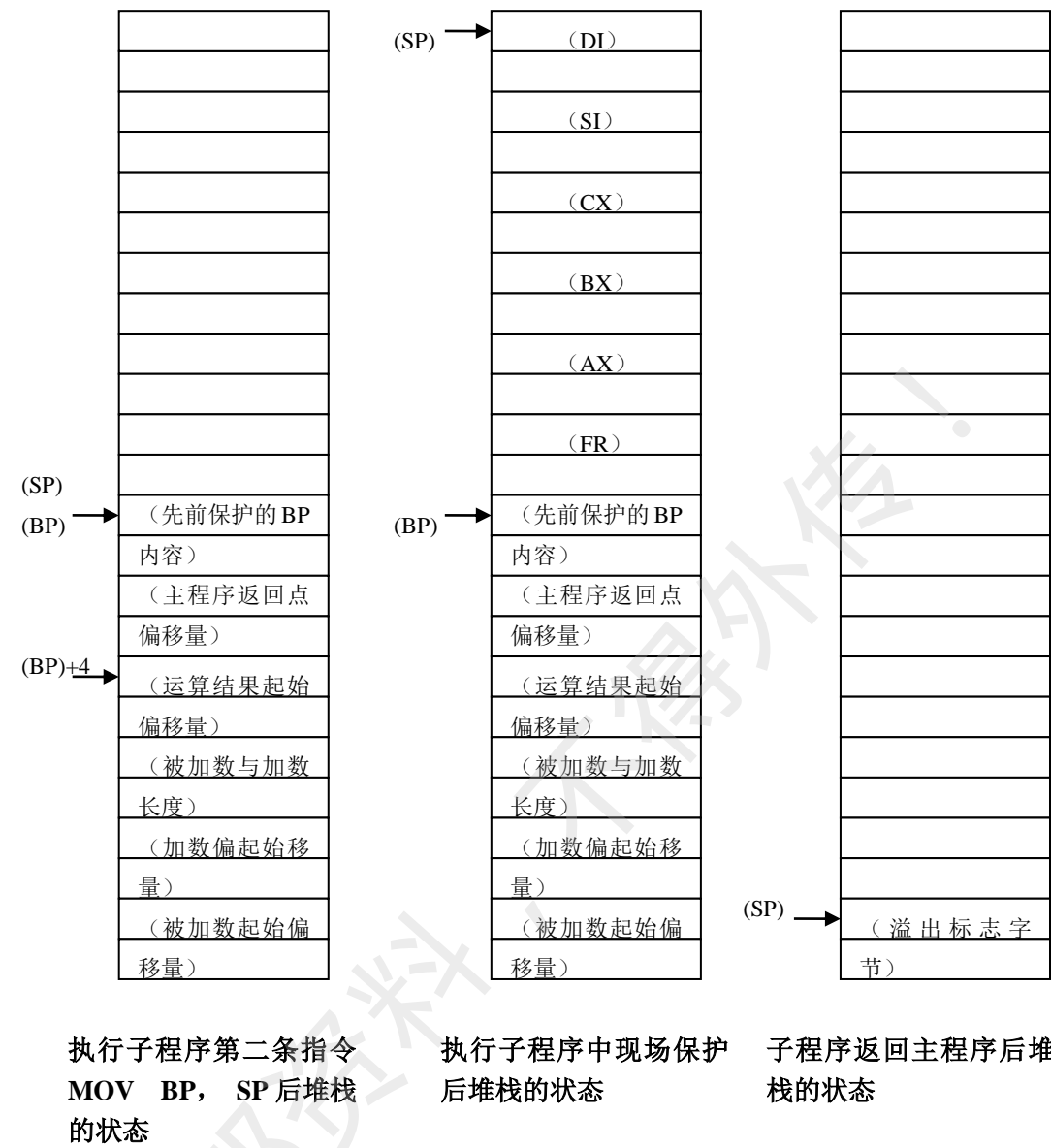
MOV SI, [BP+10] ; 取得被加数的起始偏移量

```

        CLC                                ; 清除 CF 标志, 保证第一次执行 ADC 时等同于 ADD
L1:     MOV  AX, [SI]
        ADC  AX, [DI]                      ; 完成当前字的加法, 引入了上次加法的进位
        PUSHF                             ; 保护 OF 标志
        MOV  [BX], AX                     ; 当前字的计算结果保存到指定位置
        ADD  SI, 2                        ; 修改 SI、DI、BX, 使它们指向下一个字
        ADD  DI, 2
        ADD  BX, 2
        POPF                             ; 恢复 OF 标志
        LOOP L1                          ; 计数循环, 完成所有字单元的加法后退出循环
        JO   L2                          ; 判断最后一次运算是否溢出
        MOV  [BP+10], 0                   ; 无溢出则将 0 送至堆栈保存, 传递出口参数
        JMP  L3
L2:     MOV  [BP+10], 0FFH                 ; 有溢出则将 0FFH 送至堆栈保存, 传递出口参数
L3:     POP  DI                          ; 恢复 CPU 现场
        POP  SI
        POP  CX
        POP  BX
        POP  AX
        POPF
        POP  BP
        RET  6
        ; 返回主程序, 仅清除 3 个入口参数, 因最先入栈的入口参数已被出口参数覆盖
LONGADD ENDP
CODE  ENDS
END   BEGIN

```

在本例中, 堆栈中重要的状态如下图所示 (每个方格表示一个字节)。



读者在本例中应注意，首先，主程序向子程序传递参数时，是通过入栈操作完成的，并且遵循了先前设计的约定顺序；其次，子程序从堆栈中取得入口参数是使用 `BP` 寄存器来完成的，这是由于入口参数位于比返回地址更高的地址单元中，不便使用出栈操作来完成；第三，子程序向堆栈传递出口参数时，覆盖了主程序最先入栈的入口参数，因此返回时仅使用 `RET` 指令清除了位于低地址端的三个入口参数。另外，在程序设计中，如果采用堆栈传递参数，务必记得使用 `RET N` 指令，在返回主程序的同时清除堆栈中的入口参数，如果这些入口参数不加以清除，就会越积越多，直至堆栈溢出。

**例 8.2.3** 试设计一程序，完成例 8.2.1 中指定的功能，但本例中参数传递方式采用数据区传递方式。

### 1) 设计参数传递约定

题目中已指明采用数据区传递方式，在这里只需指明各参数在数据区中的顺序、占据的空间。我们这里约定主程序按照被加数起始偏移量、加数起始偏移量、被加数与加数的长度、运算结果起始偏移量的顺序在入口参数数据区中存放入口参数，子程序也按照此顺序在数据区中获取入口参数，各参数均占据一个字的空间。因为出口参数仅有一个，所以出口参数数据区中仅存一个字的空间。

### 2) 源程序设计

参数传递约定设计完成后，我们开始源程序设计，主程序与子程序的相应代码如下所示。

```
DATA SEGMENT
VAR1 DW 5482H, 669EH, 02C7H, 14B2H, 0C254H
VAR2 DW 8C2BH, 0C24CH, 0AB12H, 357AH, 41A5H
LEN EQU $-VAR2
SUM DW LEN DUP(0) ; 用于保存运算结果
OVR DB ? ; 用于保存溢出标志
IPAR DW 4 DUP(0) ; 入口参数数据区
OPRA DW 1 ; 出口参数数据区
DATA ENDS

STACK1 SEGMENT STACK ; 子程序设计必须使用堆栈段
DW 40H DUP(0)
STACK1 ENDS

CODE SEGMENT
ASSUME CS: CODE, DS: DATA, SS: STACK1
BEGIN: MOV AX, DATA
```

```
MOV DS, AX

LEA SI, IPAR ; SI、DI 分别指向入、出口参数数据区

LEA DI, OPAR

MOV WORD PTR [SI], OFFSET VAR1 ; 传递入口参数

MOV WORD PTR 02H[SI], OFFSET VAR2

MOV WORD PTR 04H[SI], LEN

MOV WORD PTR 06H[SI], OFFSET SUM

CALL LONGADD ; 调用子程序

MOV DL, BYTE PTR [DI] ; 保存出口参数

MOV OVR, DL

MOV AH, 4CH

INT 21H

LONGADD PROC

    PUSHF ; 保护 CPU 现场

    PUSH AX

    PUSH BX

    PUSH CX

    PUSH DX

    PUSH SI

    PUSH DI

    MOV DI, 02H[SI] ; 从入口参数数据区按照约定获取入口参数

    MOV CX, 04H[SI]

    MOV BX, 06H[SI]

    MOV SI, [SI]

    CLC ; 清除 CF 标志, 保证第一次执行 ADC 时等同于 ADD

L1: MOV AX, [SI]

    ADC AX, [DI] ; 完成当前字的加法, 引入了上次加法的进位

    PUSHF ; 保护 OF 标志

    MOV [BX], AX ; 当前字的计算结果保存到指定位置

    ADD SI, 2 ; 修改 SI、DI、BX, 使它们指向下一个字
```

```
    ADD    DI, 2
    ADD    BX, 2
    POPF                                ; 恢复 OF 标志
    LOOP   L1                          ; 计数循环，完成所有字单元的加法后退出循环
    JO     L2                          ; 判断最后一次运算是否溢出
    MOV    DL, 0                       ; 无溢出则将 0 送至 DL 保存
    JMP    L3
L2:    MOV    DL, 0FFH                 ; 有溢出则将 0FFH 送至 DL 保存
L3:    POP    DI                       ; 恢复 (DI)，使之指向出口参数数据区
    MOV    [DI], DX                   ; 向数据区传递出口参数
    POP    SI                         ; 恢复 CPU 现场
    POP    DX
    POP    CX
    POP    BX
    POP    AX
    POPF
    RET                                ; 返回主程序
LONGADD ENDP
CODE    ENDS
END     BEGIN
```

在本例中读者应注意，主程序仅通过 SI、DI 寄存器分别向子程序传递了入口参数数据区、出口参数数据区的起始偏移量，而子程序则是按照约定位置在数据区中读取入口参数、写入出口参数。

## 8. 3 系统调用

### 8. 3. 1 系统调用的概念

除了上一小节介绍的自定义子程序外，操作系统还会提供大量的系统调用。这些系统调用通常用于完成常规外部设备的输入、输出功能，以减轻程序员的负担，使他们能够从繁琐的输入、输出程序设计中解脱出来，集中精力于应用程序的设计。系统调用是使用中断调用指令 **INT** 来完成的，而系统调用返回主程序是使用中断返回指令 **IRET** 来完成的，这两种指令的格式与功能如下所示。

#### (1) 中断调用指令

指令格式：INT 中断类型号

指令功能：

- 1)  $(SP) - 2 \Rightarrow SP$   
 $(FR) \Rightarrow (SP)$
- 2)  $(SP) - 2 \Rightarrow SP$   
 $(CS) \Rightarrow (SP)$
- 3)  $(SP) - 2 \Rightarrow SP$   
 $(IP) \Rightarrow (SP)$
- 4)  $0 \Rightarrow IF$   
 $0 \Rightarrow TF$
- 5)  $(0000 : \text{中断类型号} \times 4) \Rightarrow IP$
- 6)  $(0000 : \text{中断类型号} \times 4 + 2) \Rightarrow CS$

中断调用指令的执行过程继承了硬件中断过程中的各个步骤。首先，**INT** 指令将 **FR**、**CS**、**IP** 按顺序入栈保存，保护 **FR** 是因为 **INT** 指令会修改 **IF**、**TF** 标志位，因此先将其作为 CPU 现场保护起来，保护 **CS**、**IP** 是为中断服务程序返回主程序提供必要条件。然后，**IF**、**TF** 标志被清零，禁止其它可屏蔽中断与单步中断打断服务程序的执行（中断服务程序不是应用程序员所编写，无单步调试的必要），如果中断服务程序本身允许更高级别的可屏蔽中

断打断自己，可以在中断服务程序中重新将 IF 标志置 1。最后，从中断向量表中获取中断服务程序入口地址。中断向量表为内部存储器中最低地址端一块存储区域，其中每 4 个字节对应一个中断服务程序的入口地址，低地址字保存入口地址的偏移量，高地址字保存入口地址的段基值，使用中断类型号乘以 4 就可以得到该中断类型在中断向量表中的对应地址，从该地址指定的单元中，可以获取服务程序的入口地址。有关硬件中断过程的说明请读者参见输入 / 输出程序设计一章中的内容。

## (2) 中断返回指令

指令格式：IRET

指令功能：

- 1)  $((SP)) \Rightarrow CS$   
 $(SP) + 2 \Rightarrow SP$
- 2)  $((SP)) \Rightarrow IP$   
 $(SP) + 2 \Rightarrow SP$
- 3)  $((SP)) \Rightarrow FR$   
 $(SP) + 2 \Rightarrow SP$

中断返回指令在中断服务系统中使用，用于返回主程序。与子程序返回指令 RET 不同，IRET 指令会多出栈一个字，用于恢复 INT 指令或硬件中断过程所入栈的标志寄存器内容。

## 8. 3. 2 常用的系统调用

与子程序调用类似，系统调用要求主程序按照约定为自己传递入口参数，并且系统调用也会遵从约定向主程序传递出口参数。因此，在程序设计中，在我们进行系统调用前，我们应先按照系统调用的约定向它传递入口参数，并在系统调用返回后，按照它的约定来获取出口参数。在这里，仅举出少数几种示例中经常使用的系统调用，说明它的参数传递方式。

### (1) 单个字符输入

调用格式：MOV AH, 01H

INT 21H



其中，AH 寄存器可以理解一种固定的入口参数，它的内容表示系统调用的子功能号。由于 21H 号系统调用中有很多种子功能，而每次调用只能使用其中一个子功能，正是 (AH) 所中的子功能号所指定的。上面格式中的系统调用，可以解释为调用第 21H 号系统调用的第 01H 号子功能。

**功能：**等待键盘输入，当键盘上输入一个字符时，在屏幕的当前光标位置回显这个字符，光标自动向右移动一个字符位置，如果已到达行尾则自动换行，并且，系统调用将输入字符保存在 AL 寄存器中。

**入口参数：**无（子功能号不视为通常的入口参数）

**出口参数：**(AL) = 输入字符的 ASCII 码

## (2) 单个字符输出

**调用格式：**MOV AH, 02H

INT 21H

**功能：**在屏幕的当前光标位置显示 (DL) 指示的字符，光标自动向右移动一个字符位置，如果已到达行尾则自动换行。

**入口参数：**(DL) = 输出字符的 ASCII 码

**出口参数：**无

## (3) 字符串输入

**调用格式：**MOV AH, 0AH

MOV DX, 缓冲区起始偏移量

INT 21H

**功能：**从键盘接收一个字符串输入，输入的字符串保存在逻辑地址 (DS): (DX) + 2 起始的缓冲区中，并在屏幕的当前光标位置回显字符串，光标自动向右移动，如果已到达行尾则自动换行。逻辑地址 (DS): (DX) 指示的字节单元解释为主程序传递的入口参数，含义为字

字符串最大字符数，逻辑地址 (DS): (DX) + 1 指示的字节单元解释为系统调用向主程序传递的出口参数，为实际从键盘接收的字符数。当输入字符数已达最大时，系统调用停止接收输入，并通过扬声器发出提示音，但无论输入字符数有没有达到最大，仅当用户在键盘上输入回车键时，系统调用才返回主程序，返回前将实际输入的字符数保存至逻辑地址 (DS): (DX) + 1 指示的字节单元。回车字符 0DH 不计入实际输入字符数中，但会被存入逻辑地址 (DS): (DX) + 2 指示的缓冲区中。

#### 入口参数:

(DS): (DX) = 缓冲区首字节地址

((DS): (DX)) = 最大字符数

#### 出口参数:

((DS): (DX) + 1) = 实际输入字符数

#### (4) 字符串输出

调用格式: MOV AH, 09H

MOV DX, 字符串起始偏移量

INT 21H

**功能:** 在屏幕的当前光标位置显示逻辑地址 (DS): (DX) 指定的字符串，光标自动向右移动，如果已到达行尾则自动换行。系统调用逐个显示字符，直到遇到 “\$” 字符为止，否则显示过程不会停止。因此，字符串必须使用 “\$” 字符结尾。

#### 入口参数:

(DS): (DX) = 字符串首字节地址

字符串以 “\$” 字符结尾

#### 出口参数: 无

### 8.3.3 系统调用示例

在本小节中，我们将举出一些系统调用的示例，这些示例将使用上一小节中介绍的系统调用完成一些常用的字符、字符串的输入、输出功能。

**例 8.3.1** 设计一程序，完成如下功能：将光标移至新行行首，从键盘接收单个字符，在屏幕当前光标位置回显输入字符，若输入的是大写英文字符则转换为小写字符，若输入为非大写英文字符则不做任何处理，将处理后的字符在新行上显示，循环执行上述过程，直到接收到回车字符为止。

```

STACK1  SEGMENT  STACK
            DW  40H  DUP(0)
STACK1  ENDS

CODE  SEGMENT
ASSUME  CS: CODE, SS: STACK1
BEGIN: CALL  NEWLINE          ; 光标置于下一行行首
            MOV  AH, 01H
            INT  21H
            CMP  AL, 0DH        ; 判断是否为回车字符
            JNZ  L1
            JMP  L2              ; 若是回车字符则退出循环
L1:  CMP  AL, 'A'                ; 判断输入字符是否为大写字符
            JAE  L3
            JMP  L4
L3:  CMP  AL, 'Z'
            JBE  L5
            JMP  L4
L5:  SUB  AL, 20H                ; 若是大写字符则转换为小写字符
L4:  CALL  NEWLINE              ; 光标置于下一行行首

```

```

        MOV    DL, AL                ; 显示转换后的字符
        MOV    AH, 02H
        INT     21H
        JMP     BEGIN                ; 循环控制
L2:     MOV    AH, 4CH
        INT     21H
NEWLINE PROC                ; 实现回车换行的子程序
        PUSH   AX
        PUSH   DX
        MOV    DL, 0DH
        MOV    AH, 02H
        INT     21H
        MOV    DL, 0AH
        MOV    AH, 02H
        INT     21H
        POP    DX
        POP    AX
        RET
NEWLINE ENDP
CODE    ENDS
END     BEGIN

```

在例 8.3.1 中，使用 21H 号调用的 01H 号子功能接收单个字符，使用 02H 号子功能显示单个字符。NEWLINE 子程序中显示了 0DH 与 0AH 两个字符，分别为回车字符与换行字符，这两个字符属于控制字符，只改变光标位置，不显示具体字符。回车字符不改变光标所在行，只是将光标置于第 0 列（首列）；换行字符则不改变光标所在列，只是将光标置于下一行同列位置，若光标已到屏幕底部，则使文本上滚一行。因此，NEWLINE 子程序的功能是将光标置于下一行的首列位置。程序中的字符转换方法是将大写字母的 ASCII 码减去 20H，这是因为大写字母的 ASCII 码范围为 41H 至 5AH，小写字母的 ASCII 码范围为 61H 至 7AH，对应的大小写字母间相差 20H。

**例 8.3.2** 设计一程序，完成如下功能：从键盘接收一个字符串，并在接收时回显输入的字符串，输入字符串最大允许 20 个字符（不计回车字符），将字符串中的数字字符全部删除，并在新行上显示处理后的字符串。

```

DATA    SEGMENT

MAXLEN  DB   20                      ; 0AH 子功能入口参数，指定最大字符数
INPTLEN DB   ?                      ; 0AH 子功能出口参数，返回实际输入的字符数
STR1     DB  21  DUP(0)             ; 0AH 子功能的字符串缓冲区
STR2     DB  21  DUP(0)             ; 09H 子功能的字符串缓冲区
DATA     ENDS

STACK1   SEGMENT  STACK
          DW  40H  DUP(0)
STACK1   ENDS

CODE     SEGMENT
ASSUME   CS: CODE, DS: DATA, SS: STACK1
BEGIN:   MOV  AX, DATA
          MOV  DS, AX
          LEA  DX, MAXLEN            ; 输入字符串
          MOV  AH, 0AH
          INT  21H
          XOR  CX, CX
          MOV  CL, INPTLEN           ; 循环次数为实际输入的字符数
          LEA  SI, STR1
          LEA  DI, STR2

LOP1:    CMP  [SI], 30H              ; 判断当前字符是否为数字字符
          JAE  L1
          JMP  L2

L1:      CMP  [SI], 39H

```

```
        JA     L2
        JMP    L3
L2:     MOV    AL, [SI]           ; 若为非数字字符则保存至 STR2
        MOV    [DI], AL
        INC    DI
L3:     INC    SI                 ; 若为数字字符则不保存至 STR2
        LOOP   LOP1
        MOV    [DI], '$'         ; STR2 以 '$' 字符结尾
        CALL   NEWLINE          ; 回车换行
        LEA    DX, STR2          ; 显示处理后的字符串
        MOV    AH, 09H
        INT    21H
        MOV    AH, 4CH
        INT    21H
NEWLINE PROC                      ; 实现回车换行的子程序
        PUSH   AX
        PUSH   DX
        MOV    DL, 0DH
        MOV    AH, 02H
        INT    21H
        MOV    DL, 0AH
        MOV    AH, 02H
        INT    21H
        POP    DX
        POP    AX
        RET
NEWLINE ENDP
CODE    ENDS
END     BEGIN
```

## 习题 8

1. 试说明子程序调用指令与返回指令分别实现返回地址保存与恢复的原理。
2. 子程序参数传递方式可以分为哪几类？简述各类参数传递方式的原理。
3. 试说明在子程序中保护 CPU 现场的必要性。
4. 试设计一子程序，实现如下功能：接收主程序传递的字符串起始偏移量、字符串长度等入口参数，将指定字符串中的小写英文字符转换为大写英文字符。参数传递方式可自行选择，但应对所选方式加以文字说明。
5. 试设计一子程序，实现如下功能：接收主程序传递的待处理数据地址、待处理数据类型（字或字节）、位编号等入口参数，将待处理数据中指定编号的二进制位清零，其它数据位则不受影响。参数传递方式可任选。
6. 试设计一子程序，实现如下功能：接收主程序传递的两个长补码地址、补码长度（以字为单位）、运算类型标识（区分加法与减法）、运算结果保存地址等入口参数，按照运算类型标识完成长补码的加法或减法。参数传递方式可任选。
7. 试设计一子程序，将主程序指定的两个字符串合并为一个字符串，请自行定义入、出口参数及其传递方式。
8. 试设计一子程序，在主程序指定的字符串中查找指定字符，将字符位置返回给主程序。请自行定义入、出口参数及其传递方式。
9. 试设计一子程序，在主程序指定的无符号数数组（字节类型）中统计奇数的个数，并将统计结果返回给主程序。请自行定义入、出口参数及其传递方式。

## 第 9 章 数值运算程序设计

80x86 汇编语言提供的算术运算指令有加（包括加 1）、减（包括减 1）、乘、除，它们可方便地对数值型数据进行二进制运算。但必须牢记，这些数据都是二进制编码的数值型数据。

对于汇编语言程序来讲，所面对的不仅仅只是二进制数据，还可能是其它编码格式的数据。比如在日常生和中，我们接触最多的、最熟悉计算的是十进制数及其计算。又如我们通过键盘键入十进制数据，但 CPU 所得到的却是这个数字的 ASCII 码。只有通过转换处理后，才能将其变换成对应的十进制数的二进制编码。为此，80x86 汇编语言提供了一组丰富的 BCD 码（Binary-Coded-Decimal）运算指令。

在前面我们已经学习了二进制加、减指令，在本章首先学习数值型数据的二进制乘、除指令，其后再学习数值型数据的 BCD 码运算指令。

### 9.1 二进制乘除法运算指令

#### 9.1.1 乘法运算指令

我们知道，在汇编语言中有两种数值型整数，一种是有符号整数（Signed Integer），另一种是无符号整数（Unsigned Integer）。针对它们，80x86 汇编提供了两种整数乘法运算指令：无符号整数乘法指令 MUL 和有符号整数乘法指令 IMUL。虽然它们所处理的整数类型不同，但确具有相同的指令格式。

指令格式：

MUL 寄存器/内存操作数 ;: 无符号整数乘法指令

IMUL 寄存器/内存操作数 ;: 有符号整数乘法指令

这两条指令具有以下特点：

1. 它们都是单操作数指令。在指令中的操作数是源操作数，它可以是寄存器操作数或内存操作数。累加寄存器（AL、AX 和 EAX）是隐含操作数。
2. 指令中源操作数的大小（Size）说明了本次操作是字节（Byte）乘、字（Word）乘还是双字（DWord）乘。其中双字乘只有 80386 及以上 CPU 支持。注意，今后凡是“双字”操作的，都应遵循这一约定。



3. 对于“字节乘”来说，积存放在寄存器 AX 中；对于“字乘”来说，积存放在寄存器 DX: AX 中；而对于“双字乘”，积则是存放在 EDX: EAX 中。正因为如此，在乘操作前无论 AH、或 DX、或 EDX 中存放的什么数据，都会因存放乘积而被破坏。

乘法指令中的操作数和乘积关系，见表 9.1。

表 9.1

乘法类型	隐含（目的）操作数	源操作数	乘积
字节乘 (Byte)	AL	8 位寄存器名 8 位数据地址	16 位数 存于 AX 中
字乘 (Word)	AX	16 位寄存器名 16 位数据地址	32 位数 存于 DX: AX 中
双字乘 (DWord)	EAX	32 位寄存器名 32 位数据地址	64 位数 存于 EDX: EAX 中

指令应用举例：

1. MUL CL ;; 无符号整数字节乘：(AL) x (CL)，积存放在 AX 中。
2. IMUL BYTE PTR [BX] ;; 有符号整数字节乘：(AL) x ([BX])，  
;; 积存放在 DX: AX 中。
3. MUL DI ;; 无符号整数字乘：(AX) x (DI)，积存放在 DX: AX 中。
4. IMUL EBX ;; 有符号整数双字乘：(EAX) x (EBX)，  
;; 积存放在 EDX: EAX 中。
5. MUL DWORD PTR[ECX] ;; 无符号整数双字乘：(EAX) x ([ECX])，  
;; 积存放在 EDX: EAX 中

80386 及以上 CPU 除支持上述 IMUL 指令的寻址格式外，还支持 IMUL 的以下特殊格式。

#### 1) 双操作数指令 IMUL Reg16, IMM

目的操作数 Reg16 是 16 位寄存器，源操作数为 16 位的立即数 IMM。16 位积存放在 Reg16 中。如果乘积超过 16 位，CF 和 OF 将置位。本条指令，既可用于有符号数乘，也可用于有符号数乘。比如指令 IMUL DX, 456。

### 2) 三操作数指令 `IMUL Reg16, Mem16, IMM`

其中的第一个操作数必须是 16 位寄存器, 第二个操作数必须是 16 位内存操作数 `Mem16`, 第三个操作数是 16 位立即数 `IMM`。它可实现将第二个操作数和第三操作数相乘, 乘积存放在第一操作数 (16 位寄存器) 中。

比如指令 `IMUL BX, [SI], 35` 则完成由 `SI` 指示内存单元字和立即数 35 相乘, 结果存入 `BX` 中。

同样, 当乘积大于 16 位整数范围时, `CF` 和 `OF` 将被置位。本指令也适用于无符号整数乘。

问, 当出现 32 位或 64 位乘时, 你应该选用什么指令?

### 3) 指令 `IMUL Reg, Reg/Mem`

其中, 第一个操作数是寄存器, 它是目的操作数; 第二个操作数是源操作数, 它可以是寄存器操作数或内存操作数。目的操作数可以是 16 位或 32 位寄存器, 源操作数必须与目的操作数的大小 (Size) 一致, 即同为 16 位或 32 位操作数。乘积存放在目的操作数 (第一操作数寄存器) 中。比如:

`IMUL DX, 25`; 16 位数相乘,  $(DX) \times 25 \rightarrow DX$

`IMUL ECX, MUL, 25`; 32 位数相乘,  $(MUL) \times 25 \rightarrow ECX$

`IMUL BX, CX`; 16 位数相乘,  $(BX) \times (CX) \rightarrow BX$

## 9.1.2 除法运算指令

同乘法指令相类似, 80x86 汇编同样有无符号整数除指令 (`DIV`) 和有符号整数除指令 (`IDIV`)。由于是整数除法, 除法结果包括了商和余数。

### 1. 指令格式

`DIV Regs/Mem`

`IDIV Regs/Mem`

除数是指令中的操作数, 可以是寄存器操作数 (`Regs`) 或内存操作数 (`Mem`)。它的大小 (Size), 表明了本指令是进行“字节除法”、“字除法”或“双字除法”。

### 2. 被除数、除数和商

除法中的被除数为隐含操作数, 它与除数、商和余数的关系, 见表 9.2。

表 9.2

除法类型	被除数	除数	商和余数
字节除法	AX	8 位寄存器或内存操作数	商: AL , 余数: AH
字除法	DX: AX	16 位寄存器或内存操作数	商: AX , 余数: DX
双字除法	EDX: EAX	32 位寄存器或内存操作数	商: EAX , 余数: EDX

从表中可见, 在字节除法中, AH 最好不要作为除数; 在字除法中, DX 和 AX 最好不要作为除数; 在双字除法中, EDX 和 EAX 最好不要作为除数, 因为在除法操作结束后, 它们会被占用。

3. 对于无符号整数除法来说, 在除法操作之前, 首先应按表 9.2 的要求, 准备好被除数。对于除数是 8 位 (字节除)、16 位 (字除) 或 32 位 (双字除) 整数时, 相应的被除数必需是 16 位、32 位或 64 位。若不符合这个要求, 可采将被除数高位字节、字或双字填零的办法, 生成 16 位、32 位或 64 位的被除数。比如无符号整数除:  $(AL) = 124$  除以 23 可用以下指令完成:

```
XOR AH, AH      ; 0 → AH, 使被除数为 AX
```

```
MOV BL, 23
```

```
DIV BL
```

执行结果:  $(AL) = 5$  (商),  $(AH) = 9$  (余数)

4. 对于有符号整数除法来说, 若被除数是 8 位、16 位或 32 位整数时, 应在执行除法指令前, 用指令 CBW、CWD、CWDE 或 CDQ 扩展符号位, 分别将它们填充成 16 位、32 位或 64 位有符号被除数。有关符号位扩展指令功能, 请见表 9.3。

表 9.3

指令	功能说明
CBW	将 AL 中符号位扩展到 AH, AX 成为有符号的字
CWD	将 AX 中符号位扩展到 DX, DX: AX 成为有符号的双字
CWDE	扩展 AX 中符号位, EAX 变成有符号的双字
CDQ	将 EAX 中符号位扩展到 EDX, EDX: EAX 成为有符号 64 位数

比如, 有符号整数除:  $(AX) = -8124$  除以 23 可用以下指令完成:

CDW           ；扩展有符号数符号位到 DX，形成有符号双字 DX: AX

MOV CX, 23

IDIV CX

执行结果：(AX) = -353 (商)，(DX) = -5 (余数)

5. 需要特别说明的是，无论在执行 DIV 或 IDIV 指令中，是除数等于 0，或是商太大或太小，超过存放商的单元所能表示范围，都会产生除法溢出中断。比如，(AX) = 1321 除以 5，商为 264，显然超过 AL 所能存放（表示）的最大整数 255，故将发生除法溢出中断。

又如，被除数 6553 除以 -5，若使用字节除，商为 -1310，余 3。显然 -1310 超过了一个字节所能表示的最小的负数（-128），故也会发生除法溢出中断。

## 9.2 BCD 码加减法指令

### 9.2.1 BCD (Binary-Coded Decimal) 码

我们已经知道数值型数据的二进制编码和它们的算术运算指令，下面我们将学习 80x86 汇编所支持的无符号整数的 BCD 编码，及其加、减运算指令。

一个十进制数位 (0 ~ 9)，可用 4 位二进制位表示，称为十进制数的二进制编码 (BCD)，它们之间的对应关系如表 9.4 示。为了表述，以下我们将由 4 个二进制位表示的一个 BCD 码称为一位 BCD 码（或称一个 BCD 码位）。

表 9.4 BCD 码和 4 位二进制间的关系

BCD 码	4 位二进制	BCD 码	4 位二进制
0	0000	5	0101
1	0001	6	0110
2	0010	7	0111
3	0011	8	1000
4	0100	9	1001

对于 BCD 码，有非组合型 (Unpacked) 和组合型 (Packed) 两种表示和存储方式：

## 1. 非组合型 (Unpacked) BCD 码

一位非组合型 BCD 码占用一个字节的低 4 位 (D3 D2 D1 D0)，高 4 位用 0 填充。也就是说，一位 BCD 码占用一个字节。如要存放一个两位十进制数，需占用 2 个字节。通常，高地址字节存放十位数，低地址字节存放个位数。

## 2. 组合型 (Packed) BCD 码

一个组合型 BCD 码由两位 BCD 码组成，占用一个字节。其中，一个 BCD 码位占用字节的低 4 位 (D3 D2 D1 D0)，另一个 BCD 码位则占用字节的高 4 位 (D7 D6 D5 D4)。如果我们将一个组合型 BCD 码看成是一个十进制数，那么高 4 位可以认为是十位数，而低 4 位则可以认为是个位数。这也就是说，一个字节可存放 0 — 99 间的 BCD 码值，或十进制数 0 — 99。显然，若要表示 4 位十进制数，就要占用 2 个字节，低地址字节则存放十位和个位，相邻的高地址字节存放千位和百位。

我们如果要在数据段中定义非组合型 BCD 和组合型 BCD 格式的十进制数 89, 67, 23, 9 时，分别使用以下语句：

UnBCD DB 09, 08, 07, 06, 03, 02, 09, 00; 定义非组合型 BCD

BCDA DB 89H, 67H, 23H, 09H; 定义组合型 BCD

又如，将 65 以组合型 BCD 码形式存入 CL，以非组合型 BCD 码形式存入 DX，使用指令：

```
MOV CL, 65H
```

```
MOV DX, 0605H
```

## 9.2.2 BCD 码加减法指令

BCD 码的加减法指令有：

非组合型 BCD 码加法调整指令 AAA (ASCII Adjust after Addition)

非组合型 BCD 码减法调整指令 AAS (ASCII Adjust after Subtraction)

组合型 BCD 码加法调整指令 DAA (Decimal Adjust after Addition)

组合型 BCD 码减法调整指令 DAS (Decimal Adjust after Subtraction)

这些指令都有一个共同的特点：它们都是用在加指令 (ADD、ADC) 和减指令 (SUB、

SBB) 之后, 对存放在 AL 中的加、减结果进行调整 (或校正), 使结果仍为 BCD 码, 即在 AL 中存放一位非组合型 BCD 码 (0—9), 或两位组合型 BCD 码 (0—99)。

### 1. 非组合型 BCD 码加法调整指令 AAA

指令格式: AAA

AAA 又称为 ASCII 加法调整指令。我们知道, 数字字符 ‘0—9’ 的 ASCII 值是 30H—39H, 就其低 4 位来讲, 与一位 BCD 码相同。若不关心高 4 位, 它就是一个非组合型 BCD 码。若将两个 1 位数字的 ASCII 码相加, 比如 39H+35H=6EH, 和是没有任何数值意义的数值。若用 AAA 指令对这两个 1 位数字的 ASCII 相加的和进行调整, 将会得到十进制值 14, 即 (AX)=0104。若再将 AX 加上 3030H, 便会得到 1 和 4 的 ASCII (即 3134H)。

比如: 计算 BCD 码 4+9:

MOV AX, 9

ADD AL, 4 ;: 完成加后, (AL) = 0DH

AAA ;: 调整 AL 的值: 由于 (AL) > 9, 所以 (AL)+6 → AL,  
;: (AH) + 1 → AH, AL 中高 4 位清 0, CF 置 1。

最后 (AH) = 1, (AL) = 3, CF=1, 即存放在 AX 中的非组合型 BCD 码 0103H, 应看作是十进制数 13。

由此可见, 两个 1 位 BCD 码 (或数字的 ASCII) 在完成相加 (比如 ADD、ADC 指令) 操作后, 再用 AAA 指令对 AL 中的值进行调整, 便可将结果调整为 BCD 码 (0—9)。调整的方法是: 若 (AL) > 9 或 AF 置 1, 则 AH 加 1, CF 和 AF 置 1, AL 高 4 位清 0; 否则, CF 和 AF 将清 0。相加和以非组合型 BCD 格式存放在 AX 中。

例如:

XOR AH, AH ;: AH 必须事先清 0, 保存相加后向高位的进位。

MOV AL, 37H ;: AL 中存放被加数 37

ADD AL, 38H ;: (AL) = 6FH

AAA ;: 调整 AL: (AL) + 6 → AL, 1 → AF, 1 → CF  
;: (AH) + 1 → AH, AL 高 4 位清 0,  
;: 相加和 (AX) = 0105H

又例如:

MOV AX, 0209H ;: 注意 (AH) = 2

ADD AL, 5

AAA ;: 调整后, (AX) = 0304H, 可以认为是计算 29+5 的和。

## 2. 组合型 BCD 码加法调整指令 DAA

指令格式: DAA

DAA 用于对两个组合型 BCD 码进行相加的加法指令之后, 对存放在 AL 中的和进行调整。调整后的 BCD 码仍保存在 AL 中。注意, 加法指令中的目的操作数必须是寄存器 AL。

调整准则:

- 1) 若 AL 的低 4 位 > 9 或向高 4 位有进位 (AF=1), 则 (AL) + 6 → AL, 且 1 → AF;
- 2) 若 AL 的高 4 位 > 9 或向更高位 (比如百位) 有进位 (CF=1), 则 (AL) + 60H → AL, 且 1 → AF (高 4 位 > 9 时), 1 → CF;
- 3) 若不属于 1)、2) 情况, 则 0 → CF, 0 → AF。

比如实现十进制数 4534+287 的指令段如下, 和存放在 DX 中, 程序未考虑相加可能向更高位进位的情况。

```
MOV AL, 34H ;: 计算低字节和 (十位和个位)
ADD AL, 87H ;: (AL) = BBH
DAA ;: 调整: (AL) = 21H, (CF) = 1, (AF) = 1
MOV DL, AL ;: 保存低位 BCD 码 → DL
MOV AL, 45H ;: 计算高字节和 (千位和百位)
ADC AL, 02H ;: 低 2 位向高位有进位, (AL) = 48H
DAA ;: 调整, (CF) = 0, (AF) = 0
MOV DH, AL ;: 保存高位 BCD 码 → DH
;: (DX) = 4821H
```

## 3. 非组合型 BCD 码减法调整指令 AAS

非组合型 BCD 码减法调整指令 AAS, 同样又称为 ASCII 减法调整指令。它不但可以用于对 1 位 BCD 码相减后的结果进行调整, 也可用于两个数字的 ASCII 相减后的结果进行调整。

指令格式: AAS

AAS 用在对两位非组合型 BCD 码相减的减法指令后, 通过对存放在 AL 中的差的调整,

使 AL 中的值为非组合型 BCD (0—9)。调整准则是：

- 1) 若  $(AL) > 9$  或  $AF=1$  (向高位有借位)，则  $(AL) - 6 \rightarrow AL$ ,  $(AH) - 1 \rightarrow AH$ ,  $1 \rightarrow CF$  和  $1 \rightarrow AF$ , AL 高 4 位清 0;
- 2) 若不符合 1) 的条件，则 CF 和 AF 清 0。

例如：

```
MOV AX, 0103H
```

```
SUB AL, 4 ;:  $(AL) - 4 = -1$  (即 FFH)
```

```
AAS ;: 调整 AL:  $(AL) - 6 \rightarrow AL$ ,  $(AH) - 1 \rightarrow AH$ , AL 高 4 位清 0,  $1 \rightarrow CF$ ,  $1 \rightarrow AF$ , 差  $(AX) = 0009$ 
```

本例类似于  $13 - 4 = 9$ 。

例如：

```
MOV AX, 04
```

```
SUB AL, 08
```

```
AAS
```

$(AX) = FF06H$ ,  $AF=1$ ,  $CF=1$ 。由于  $CF=1$ ，表示是向高位有借位，或结果为负。

#### 4. 组合型 BCD 码减法调整指令 DAS

指令格式： DAS

DAS 用在对两个组合型 BCD 进行相减操作的减法指令之后，通过对 AL 中的结果 (差) 进行调整，使 AL 中的值为组合型 BCD (00—99)。调整准则是：

- 1) 若 AL 中低 4 位  $> 9$  或  $AF=1$  (向高 4 位有借位)，则  $(AL) - 6 \rightarrow AL$ ，且  $1 \rightarrow AL$ ;
- 2) 若 AL 中高 4 位  $> 9$  或  $CF=1$  (向更高位有借位)，则  $(AL) - 60H \rightarrow AL$ ，且  $1 \rightarrow CF$ ;
- 3) 若不属于 1) 和 2)，则  $0 \rightarrow CF$  和  $0 \rightarrow AF$ 。

比如计算  $83 - 38$

```
MOV AX, 3883H
```

```
SUB AL, AH ;:  $83H - 38H = 4BH$ ，即  $(AL) = 4BH$ 
```

```
DAS ;: 对 AL 进行调整，形成组合型 BCD,  $(AL) = 45H$ ,  
;:  $CF = 1$ ,  $AF = 1$ 。表示差为 45。
```



## 9.3 BCD 码加减法程序设计原理与实现

在 80x86 汇编语言程序中，使用 BCD 码运算时，应注意以下几点：

1. 非组合型 BCD 码和组合型 BCD 码在数据段、寄存器中的表示形式。比如在内存单元中定义非组合型 BCD 码时，最好将高 4 位置为 0；同时还应遵循 PC 机在内存单元中存放数据的规则，即低地址存放低字节（低位），相邻的下一个高地址存放高位字节。

2. 在数据段或寄存器中，定义组合型 BCD 码时，字节的低 4 位存放低位 BCD 码；字节的高 4 位存放高位 BCD 码；每一个组合型 BCD 码都必须定义成 Hex。

例如，定义组合型 BCD 码数据 35，78，97，45 等时，用以下伪指令：

```
DB 35H, 78H, 97H, 45H
```

```
MOV AL, 97H
```

若用指令 DB 35，78，97，45 和 MOV AL，45，在汇编后，存放在内存中的数据实际上是 23H，4EH，61H，2DH，它们显然不是我们所需要的组合型 BCD 值。

3. 对于非组合型 BCD 码加减操作，一次只能完成一位数加减；对于组合型 BCD 码加减操作，一次只能完成两位数加减。也就是说，它们每次进行只能字节加减运算。

4. 无论是非组合型还是组合型 BCD 码，加减运算后，被调整的对象总是 AL 中的内容。

对于非组合型 BCD 码的调整中，进位或借位（CF=1），都会使（AH）受影响，即（AH） $\pm 1 \rightarrow AH$ 。

对于组合型 BCD 码的调整中，高位 BCD 码的进位或借位，只会使  $1 \rightarrow CF$ ，不会影响 AH 中的值。

5. 在进行多位非组合型或组合型 BCD 的加减运算，要特别注意进位和借位的处理，结果数符（正数或负数）的确定。

例：写程序段，用组合 BCD 码加，实现十进制数  $57836 + 169427$

```
BCDA DB 36H, 78H, 05H
```

```
BCDB DB 27H, 94H, 16H
```

```
BCDSUM DB 0, 0, 0
```

```
MOV SI, OFFSET BCDA
```

```
MOV DI, OFFSET BCDB
```

```
MOV BX, OFFSET BCDUM
```

```

        CLC
        MOV  CX, 3
        XOR  AH, AH
NEXT:    MOV  AL, [SI]
        ADC  AL, [DI]
        DAS
        MOV  [BX], AL
        INC  SI
        INC  DI
        INC  BX
        LOOP NEXT

```

## 9.4 BCD 码乘法调整指令

80x86 汇编语言，除支持 BCD 码的加减调整指令外，还支持 BCD 码的乘法调整指令（ASCII Adjust after Multiplication）和除法调整指令 AAD（ASCII Adjust before Division）。

### 9.4.1 BCD 码乘法调整指令 AAM

BCD 码乘法调整指令 AAM 用在两个一位 BCD 码乘操作指令 MUL 或 IMUL 之后，实现对 AX 中的积进行调整，使之成为非组型 BCD 码，并保存在 AX 中。实际上，两个 1 位 BCD 码相乘，其积的最大值是 81（即 51H）。AAM 将保存在 AL 中 Hex 值变换成两位 BCD 码，十位存放在 AH 中，个位存放在 AL 中。

由于数字字符 ASCII 码的低 4 位就是其数字值，与非组合型 BCD 码相同，因此 AAM 又称为 ASCII 乘法调整指令。在对数字字符 ASCII 进行乘操作（字节乘）时，应首先使其高 4 位清 0，从而保证参与乘的两个数都是 BCD 码。

例如：计算 ASCII 码 38H（‘8’）和 36H（‘6’）的 BCD 码乘

```

MOV  AL, 38H
AND  AL, 0FH      ;; AL 高四位清 0
MOV  BL, '6' AND 0FH ;; 6 → BL

```

```

    MUL    BL           ;; (AX) = 30H
    AAM                ;; 将 (AX) 调整成 BCD, (AX) = 0408H

```

## 9.4.2 BCD 码除法调整指令 AAD

BCD 码除法调整指令 AAD，同样也被称为 ASCII 除法调整指令。在实施一位 BCD 码的除法 (DIV 或 IDIV) 操作之前，它对保存在 AX 中的两位非组合型 BCD 码被除数进行调整，使其成为 Hex，并保存 AX 中，为使用除法指令 (二进制除) 作准备。除数只能是一位 BCD 码值 (1~9)。在完成除操作后，商存于 AL 中，余数存于 AH 中。

例如，计算两位非组合 BCD 码除法：56÷6

```

    MOV    AX, 0506H    ;; 非组合型 BCD 码被除数送 AX
    AAD                    ;; 调整后, (AX) = 38H
    MOV    CL, 6
    DIV    CL           ;; 商 (AL) = 9, 余数 (AH) = 2
    AAM                ;; 将商调整成非组合 BCD, 存于 AX 中

```

此例中，(AX) = 0009H，余数被丢弃。

又如：用 BCD 码除法调整指令，

```

    MOV    AX, 0506H    ;; 非组合型 BCD 码 56 被除数送 AX
    AAD                    ;; 调整后, (AX) = 38H
    MOV    CL, 3
    DIV    CL           ;; 商 (AL) = 12H, 余数 (AH) = 2
    MOV    CL, AH       ;; 保存余数到 CL
    AAM                ;; 将商 12H 调整成非组合 BCD, 存于 AX 。

```

此例中，(AX) = 0108H，即商等于 18；余数 2 存于 CL 中。

从例子可见，首先应使用除法调整指令 AAD，将 AX 中非组合型 BCD 码调整成二进制数，其后再用除法指令完成除操作，保存余数。最后再使用乘法调整指令 AAM，对 AL 中的值调整成非组合型 BCD 码，存放到 AX 中。

## 习题 9

1. 用数据定义指令 `NUM DB 89H`, 问我们可以将它看作一个什么数?
2. 试使用加法指令, 编写一个子程序, 实现两个字相乘的功能。
3. 试使用减法指令, 编写一个子程序, 实现两个字相除的功能。
4. 试编写子程序, 实现指令 `AAA`、`AAD` 的功能。
5. 试编写子程序, 实现指令 `AAS`、`DAS` 的功能。
6. 在 `UnPKBCD` 中, 定义有 20 个两位非组合型 `BCD` 码整数, 编写一个计它们和的程序。  
假定最终和 < 999, 且存放在 `SUM` 开始的单元中。比如数 89, 34 在 `UnPKBCD` 中定义成:  
09, 08, 04, 03。根据此约定自行定义其余数据。
7. 在 `PKBCD` 中, 定义有 20 个两位组合型 `BCD` 码整数, 编写一个计它们和的程序。假定  
最终和 < 999, 且存放在 `SUM` 开始的单元中。比如数 89, 34 在 `PKBCD` 中定义成: 89H,  
34H。根据此约定自行定义其余数据。
8. 设有两个由 6 个组合型 `BCD` 码组成的多精度数, 试编写程序计算它们的和。

## 第 10 章 非数值处理程序设计

我们在第八章，学习了汇编语言的数值计算程序设计，在那里我们所处理的数据都是数值型数据，它们可以是二进制，十进制，16 进制或 BCB 编码格式表示的，可以方便地实现加、减、乘、除等基本运算。

在汇编语言程序中，除对数值型数据的处理（加、减、乘、除和比较等）外，还有对非数值型数据的处理（比如比较、查找、排序和编码变换等）。非数值型数据处理也是汇编语言中数据处理的重要部分。在本章，我们将学习 80x86 汇编语言的非数值处理指令和相关的程序设计技术。

串操作是非数值处理的一个重要内容。为此，80x86 CPU 提供了一组功能强大的串操作指令。需要特别要说明的是，在这里的“串”（String）并不仅仅是由一个或多个字符（字节）组成的、指通常意义下的字符串，而且还可能是由若干字节（Byte）、字（Word）或 80386 及其以上 CPU 支持的双字（Dword）组成的数据的集合。它们总是存放在一片连续的内存单元中。

串操作指令的基本格式如下，它包括可选的语句标号（LABEL）和重复操作前缀（Repeat Prefix）、串操作指令三部分。

**[LABEL: ] [Repeat Prefix] 串操作指令**

指令的操作单位可以是一个字节（Byte）、一个字（Word）或一个双字（Dword）。80x86 汇编语言提供的串操作指令有：

指令	功能描述
LODS	取串指令。从串（内存）中取一个字节、字或双字，并存入累加寄存器 AL、AX 或 EAX。
STOS	存串指令。将累加寄存器 AL、AX 或 EAX 中的值存放到串中。
MOVS	串传送指令。从一个串（内存区域）中取一个字节、字或双字，传送（搬移）到另一个串存储区中。
CMPS	串比较指令。以一个字节、字或双字为操作单位，对两个串中的数据进行比较。

SCAS	串搜索指令。在指定串中搜索，查看是否含有累加寄存器 AL、AX 或 EAX 中的内容。
INS	从口（PORT）地址读取串指令。将从口地址中读取的串数据（字节、字或双字），存放到内存单元中。
OUTS	写串数据到口地址指令。将内存单元中的串数据（字节、字或双字）写入口地址。

需说明的是，由于串操作指令 INS 和 OUTS 与口地址操作有关，故不在本章介绍它们。

在程序中，有时需要对存放在一片连续内存区域中的串数据重复处理，这就需要重复执行串操作指令。汇编语言提供的串重复操作前缀指令 REP、REPZ/REPE 和 REPNZ/REPNE，在满足重复操作的条件下，重复执行串操作指令。

串操作指令在格式上类似，特别是串操作指令功能强大。一条指令的功能可能需要用多条以前我们所学习过的指令来实现，这一点将在以后的串操作指令学习中体会到。

## 10.1 串操作指令

### 10.1.1 串操作指令的特点

串操作指令具有以下特点：

#### 1. 串操作指令的基本格式

[LABEL: ] [Repeat Prefix] 串操作指令

其中串操作指令是一条单独的功能强大的操作指令，这一点将在其后的深入学习中，逐步加深认识。重复操作前缀[Repeat Prefix]是可选的，它与串操作指令写在一行，共同形成一条可重复执行的串操作指令。但是，在指令的应用中，选用何种重复前缀，是 REP、REPZ/REPE 还是 REPNZ/REPNE，或者不使用，应根据所使用指令和要实现的功能来确定。有关它们的使用，将在以后介绍。

#### 2. 串操作单位

串操作单位就是指在一次串操作中所处理的串数据的大小（SIZE）。我们知道，在 80x86 汇编语言中的“串”（String）数据，实际上是存放在一片（连续）内存单元中的数据的集合。可以

根据需要，对这些数据按字节（Byte）为单位，字（Word）为单位，或双字（DWord）为单位进行处理，即是说，在指令执行中，被处理的数据可以是字节、字或双字，它们就是一个操作单位。

### 3. 源串数据地址和目的串数据地址

在串操作指令中，通常会涉及到待操作的串数据地址。对于源串数据地址由地址指针 DS:[SI] 默认指定。在需要特别指明它所在的段时，也可以使用段前缀，比如 CS、SS 和 ES 段。目的串数据地址只能由地址指针 ES:[DI] 指定。

### 4. 操作方向

在串数据处理时，以某一个地址为首地址，从内存单元中读取第一个串数据，改变地址，再读取第二个串数据，再改变地址，读取第三个串数据等。操作方向的含意是说在操作中，它们的地址的变化方式。很显然，对于连续的串数据处理中，从首地址开始，读取第一个串数据，然后地址增加一个操作单位大小（字节为 1，字是 2，双字是 4），指向下一个串数据首地址，便可取得第二个串数据。按照这样，以地址依次增加的形式处理数据，称为正向操作方式；也有从最后一个数据单位的首地址开始，地址依次递减，获取串数据的，称为反向操作方式。

通过设置标志寄存器中的方向标志位 DF 来决定串操作方向。当方向标志位 DF=0，则为正向操作；当 DF=1 则为反向操作。在程序中，我们用指令 CLD 使 DF=0（清 0），用指令 STD 使 DF=1（置 1）。通过这一对指令，就可事先规定串处理操作的方向。当 DF=0，这时采用正向操作方式。当 DF=1，这时采用反向操作方式。使用 CLD 指令，可以清零 DF 位（DF=0）；使用 STD 指令，可以置位 DF 位（DF=1）。

### 5. 源串和目的串地址的自动修改

每执行一次串操作指令，源串和目的串的地址（SI 和 DI）将会按照操作方向、操作单位的大小自动修改，指向下一个单位的首地址。

操作方向	操作单位	源串偏移地址的修改	目的串偏移地址的修改
DF=0	BYTE	(SI) + 1 → SI	(DI) + 1 → DI
	WORD	(SI) + 2 → SI	(DI) + 2 → DI
	DWORD	(SI) + 4 → SI	(DI) + 4 → DI
DF=1	BYTE	(SI) - 1 → SI	(DI) - 1 → DI
	WORD	(SI) - 2 → SI	(DI) - 2 → DI

DWORD                      (SI) -4 → SI                      (DI) -4 → DI

## 6. 重复次数（或重复前缀）

在需要多次重复执行串操作指令时，应使用重复前缀。在任何一个处理中，每个串操作指令最大可能的重复次数必须事先确定，并且事前存放在 CX 寄存器中。每执行一次操作指令，CX 的值会自动减 1。除 CX 中的值用来控制指令的重复执行次数外，ZF 标志也可作为控制指令的重复执行次数的另一个条件。重复前缀和指令的配合使用情况如下。

重复前缀	可使用指令
REP	MOVS、STOS、INS 和 OUTS
REPZ/REPE	CMPS、SCAS
REPNZ/REPNE	CMPS、SCAS

有关各重复前缀的重复条件和指令的使用，将在下节详细介绍。

## 10.1.2 串操作指令

在上一节中，我们已经了解了串操作指令特点，下面我们将介绍 6 种串操作指令。它们尽管看起来有些复杂，但实际上很简单，规律性强。

### 1. 取串指令 LODS (Loads String)

指令格式：LODS 源串

源串地址由 DS:[SI] 指定。它从源串中读取一个字节、字或双字，存放在 AL、AX 或 EAX 中，并按照 DF 值和操作单位大小修改 SI。通常，在程序中常使用以下简化的无操作数指令格式，实际上这是一种隐含操作数指令。

LODSB    ····· 从源串中读取一个字节，存入 AL，(SI) ±1 → SI  
 LODSW    ····· 从源串中读取一个字，存入 AX，(SI) ±2 → SI  
 LODSD    ····· 从源串中读取一个双字，存入 EAX，(SI) ±4 → SI

指令的执行后，对标志位无影响。由于取串指令每次只是完成从源串中取出一个操作单位的数



据，并存放在累加寄存器中，一般没有使用重复前缀的必要。

有时，源串也可使用符号地址的格式表示。比如，若符号 LIST 定义为 WORD 属性，则指令：

```
LODS LIST
```

同样实现从 LIST 中读取一个字，并存入 AX 中，同时按 DF 值，修改 SI，即完成  $(SI) \pm 2 \rightarrow SI$  操作。

在用地址指针表示源操作数地址时，应使用 PTR 操作符，明确操作数的大小 (SIZE)。其形式如下：

```
LODS WORD PTR [SI]
```

## 2. 存串指令 STOS (Store String)

指令格式：STOS 目的串

目的串的地址由 ES: [DI] 指定。它实现将 AL、AX 或 EAX 中的内容存放到目的串指示的存储单元中的功能，并按照 DF 值和操作单位大小修改 DI。简化的无操作数指令格式是：

[Repeat Prefix] STOSB ..... 将 AL 中的值存入目的串， $(DI) \pm 1 \rightarrow DI$

[Repeat Prefix] STOSW ..... 将 AX 中的值存入目的串， $(DI) \pm 2 \rightarrow DI$

[Repeat Prefix] STOSD ..... 将 EAX 中的值存入目的串， $(DI) \pm 4 \rightarrow DI$

指令的执行，对标志位无影响。由于其功能是将累加寄存器中的内容写入目的串单元。显然，这条指令可用来实现对内存区域的填充。

REP 是可选用的重复前缀，重复次数存放在 CX 中。

比如，用空格 (20H) 填充在 DATA 段中 DISP 开始的 100 个字节单元程序段如下。在程序段中，我们采用反向操作方式和 DWORD 操作单位。

```
.....
```

```
MOV AX, DATA
```

```
MOV ES, AX
```

```
MOV DI, OFFSET DISP
```

```
STD
```

```
ADD DI, 100-4 ;: 反向操作中，第一个操作单元的首地址
```

```
MOV CX, 25
```

```
MOV EAX, 20202020H
```

```
REP STOSD
```

执行完成后,  $(CX) = 0$ ,  $(DI) = \text{OFFSET DISP}$ 。

同样, 用符号地址表示目的地址时, 可写成:

```
REP STOS DWORD PTR DISP
```

或 `REP STOS DWORD PTR [DI]`

### 3. 串传送指令 (Moving String)

指令格式: `MOVS 目的串, 源串`

源串地址由 `DS: [SI]` 指定, 目的串地址由 `ES: [DI]` 指定。指令按指定的串操作单位, 从源串中取出一个操作单位的串数据, 存放到目的串中, 并按照 `DF` 值同时修改 `SI` 和 `DI`。通常, 在程序中也可使用以下无操作数指令格式:

`[Repeat Prefix] MOVSB` ..... 从源串中取一个字节, 写入目的串, 修改地址指针:  $(SI) \pm 1 \rightarrow SI$ ,  $(DI) \pm 1 \rightarrow DI$

`[Repeat Prefix] MOVSW` ..... 从源串中取一个字, 写入目的串, 修改地址指针:  $(SI) \pm 2 \rightarrow SI$ ,  $(DI) \pm 2 \rightarrow DI$

`[Repeat Prefix] MOVSD` ..... 从源串中取一个双字, 写入目的串, 修改地址指针:  $(SI) \pm 4 \rightarrow SI$ ,  $(DI) \pm 4 \rightarrow DI$

显然, 该指令可实现对串数据的搬移或传送。指令的执行对标志位无影响。同样可选用的重复前缀是 `REP`。

### 4. 串比较指令 CMPS (Compare String)

指令格式: `CMPS 源串, 目的串`

其中源串地址由 `DS: [SI]` 指定, 目的串地址由 `ES: [DI]` 指定。指令按指定的串数据操作单位, 从源串中取出一个操作数, 并与目的串中一个操作单位数据进行比较 (比较结果是它们相等或不相等), 比较结果反映在标志寄存器的 `ZF` 标志位上。比较后, 源和目的串均不受影响, 并按照 `DF` 值, 修改 `SI` 和 `DI`。通常, 在程序中, 使用的无操作数指令格式是:

`[Repeat Prefix] CMPSB` ..... 对源串和目的串, 按字节比较, 修改地址指针:  $(SI) \pm 1 \rightarrow SI$ ,  $(DI) \pm 1 \rightarrow DI$

[Repeat Prefix] CMPSW .....对源串和目的串,按字比较,修改

地址指针: (SI)  $\pm 2 \rightarrow$  SI, (DI)  $\pm 2 \rightarrow$  DI

[Repeat Prefix] CMPSD .....对源串和目的串,按双字比较,修

改地址指针: (SI)  $\pm 4 \rightarrow$  SI, (DI)  $\pm 4 \rightarrow$  DI

由于指令的执行影响 ZF 标志,所以应选用含有 ZF=0 或 ZF=1 为条件的重复前缀。当选用重复前缀 REPZ/REPE 时,即 ZF=1 (意为相等),表示相等则继续执行串比较指令。当选用重复前缀 REPNZ/REPNE 时,即 ZF=0,表示为不相等时继续执行串比较指令。当不满足重复前缀的条件时,则结束串操作指令的执行,而执行其后的下一条指令。此外,重复条件同样受 CX 的值的约束。

## 5. 搜索指令 SCAS (Scan String)

指令格式: SCAS 目的串

其中,目的串的地址由 ES: [DI]指定。指令按指定的串操作单位,从目的串中取出一个操作单位数据,与累加寄存器 (AL、AX 或 EAX) 的中内容进行比较 (其结果为相等或不相等),比较结果影响 ZF 标志。比较后,目的串中的内容不受影响,同时按照 DF 值修改 DI。也就是说,该指令是在目的串中搜索,确认是否包含有累加寄存器 (AL、AX 或 EAX) 中指定的串。在程序中,可使用的无操作数指令格式是:

[Repeat Prefix] SCASB .....比较目的串中是否与 AL 内容相同或不同,

修改地址指针: (SI)  $\pm 1 \rightarrow$  SI, (DI)  $\pm 1 \rightarrow$  DI

[Repeat Prefix] SCASW .....比较目的串中是否与 AX 内容相同或不同,

修改地址指针: (SI)  $\pm 2 \rightarrow$  SI, (DI)  $\pm 2 \rightarrow$  DI

[Repeat Prefix] SCASD .....比较目的串中是否与 EAX 内容相同或不同,

修改地址指针: (SI)  $\pm 4 \rightarrow$  SI, (DI)  $\pm 4 \rightarrow$  DI

由于指令的执行影响 ZF 标志,所以应选用重复前缀 REPZ/REPE, 或 REPNZ/REPNE。当选用重复前缀 REPZ/REPE 时,即 ZF=1(相等),则继续执行串搜索指令。当选用重复前缀 REPNZ/REPNE 时,则 ZF=0,表示为不相等时继续执行串搜索指令。同样,当不满足重复前缀的条件时,则结束串操作指令的执行,而执行其后的下一条指令。此外,应注意的是重复条件还要受 CX 值的约束。

## 6. 重复前缀 (Repeat Prefix)

在需要重复使用串操作指令时，有 REP, REPZ/REPE 和 REPNZ/REPNE 三种重复前缀可供选用。

### 1) 重复前缀 REP

重复前缀 REP 的功能是重复执行串操作指令，一直到重复执行到重复次数 (CX) = 0 为止。在每执行一次串操作指令后，都会自动完成 (CX) - 1 → CX 操作。这就是说，REP 判断的条件是 CX 的内容是否为 0。若 (CX) ≠ 0，则继续执行串操作指令，否则结束串操作指令的执行，顺次执行其后的下一条指令。

显然，REP 只能作为 STOS 和 MOVS 指令的重复前缀。比如，用字符 'A' 填满 100 个连续的内存单元程序段：

```
MOV CX, 100
MOV AL, 'A'
REP STOSB
```

又如，将源串中的 100 个字节的数据 55AA 存放到目的串中的程序段：

```
MOV CX, 50
MOV AX, 55AAH
REP MOVSW
```

注意，在上述两个程序段中，假定对源串和目的串的地址都已经正确设置。

### 2) 重复前缀 REPZ/REPE

使用重复前缀 REPZ/REPE，则在每一次执行串操作指令前，要先检查条件“(CX) ≠ 0 且 ZF=1”是否满足，若条件满足，则执行一次串操作指令。若条件不满足，则结束串操作指令的执行，而顺次执行其后的指令。

重复前缀 REPZ/REPE 条件中，包含有两个条件：一个条件是受限于 CX 的值，一个条件是 CF 的值。CX 的初值是串操作指令可能执行的最多次数，每执行一次操作，CX 会自动减 1，直到 (CX) = 0 为止；在每执行一次串操作指令时，ZF 标志位会受到影响，而 ZF=1 则表示有“相同或相等”的含义。这样一来，同时受限 CX 和 ZF 的重复前缀 REPZ/REPE，则表示在最多重复次数（即 CX 的初值）的限制下 (CX 的初值 ≠ 0)，且相等（即 ZF=1），则重复执行串操作指令。结束串操作指令执行的条件是，执行完受限的次数（即 (CX) = 0），或者在执行过程中出现有不相同 (ZF=0) 的情况。

由于 REPZ/REPE 中带有对标志位 ZF 的进行判断的条件，所以它可作为影响 ZF 标志位的串

操作指令 CMPS 和 SCAS 的重复前缀。

### 3) 重复前缀 REPZ/REPNE

重复执行串操作指令的条件是“(CX) ≠ 0 且 ZF=0”。也就是说，前缀 REPZ/REPNE 表示，在最多重复次数的限制下（即 (CX) ≠ 0），且不相等（ZF=0），才重复执行串操作指令。否则，结束串操作指令的执行，顺序执行其后的下一条指令。

由于 REPZ/REPNE 中带有对 ZF 标志位进行判断的条件，所以它同样可作为影响标志位的串操作指令 CMPS 和 SCAS 的重复前缀。

REPZ/REPNE 有相等循环/不相等循环的意思，类似于指令 LOOPZ、LOOPNZ 的功能。

通过对上述串操作指令的学习，我们加深了对串操作指令是“复合性指令”的理解。一个串操作指令，不象 ADD, CMP 等指令的功能单一性，它会完成以下的几主要个操作：按照 DF 标志和操作单位的大小，修改源和目的地址，修改 CX（执行串操作指令一次，CX 减 1 一次），影响标志位。

请分析程序段的功能：

```
        REPZ  CMPSB
        JNZ   OTHER
NEXT:    MOV   AL, 0FH
        JMP   OK
OTHER:   MOV   AL, 0
OK:      NOP
```

执行本段程序后，在什么条件下 (AL) = 0FH？，又在什么条件下 (AL) = 0？

## 10.2 串操作指令的应用

### 10.2.1 串操作指令在程序中的使用要点

通过上一节我们对串操作指令的学习，在程序中使用串操作指令时，应注意以下几点：

1. 根据要完成的操作，选取合适的串操作指令。比如，取串指令 LODS，往往用于将源串中的数据读入累加寄存器中，其后我们再对累加寄存器中的数据进行处理；存串指令 STOS 是将累加寄存器中的数据存入目的串最方便的指令。若加上 REP 前缀，则可实现对目的串所指示的内存单元区进行填充；通过 REP 前缀，串传送指令 MOVS 可将源串中的数据顺序地依次搬移到目

的串中；在确认源串和目的串对应位置上的数据单位是否相同或不相同时，可选用串比较 **CMPS** 指令。必要时，可以使用 **REPZ**、**REPZ** 重复前缀。串搜索 **SCAS** 指令可查询目的串，确认它是否有与累加寄存器中的内容是否相同或不同，同样可以使用 **REPZ**、**REPZ** 前缀，进行重复搜索。

2. 选用 **CLD** 或 **STD** 指令，设置操作方向。

3. 设置串源地址首地址 **DS: SI** 和目的串首地址 **ES: DI**。注意，这里的首地址意指操作首地址。特别是在反向操作时，操作首地址应是存放的最后一个操作单位数据的首地址。

4. 在确定串操作的操作单位是字节、字或是双字后，进而应确定应该使用的指令。比如在串传送中，是选用 **MOVS**、**MOVSW** 还是使用 **MOVSD**。

5. 若要重复操作，应选取适当的重复操作前缀，且将最多可能的重复操作次数送入 **CX**。

6. 注意使用 **REPZ/REPE**、**REPZ/REPNE** 后的结束处理。

上述几点，将在其后的例子中给予说明。

例如，以字节为操作单位，查找在两个串中对应位置上是否有相同的字符。程序段格式如下：

```
;; 设置 DS 和 ES, SI 和 DI。假定有一个数据段 DATA, SRC_STRING 为源串符
;; 号地址, DEST_STRING 为目的串符号地址。
ASSUME DS: DATA, ES: DATA
MOV AX, DATA
MOV DS, AX ;; 设置 DS 和 ES 初值
MOV ES, AX
MOV SI, OFFSET SRC_STRING ;; 设置 SI 和 DI 初值
MOV DI, OFFSET DEST_STRING
;; 设置操作次数
MOV CX, NUM ;; 最多可能的重复查询次数。
;; 设置操作为方向，比如正向操作
CLD
;; 串操作指令，比如串比较（以字节为操作单位，找两串中在对应位置上是否相同数据）
REPZ CMPSB
;; 比较结果处理
JNZ Not_Match
;; At least One_Matched
MOV AL, 80H
```

.....

Not\_Match: MOV AL, 00H

## 10.2.2 程序设计举例

1. 查找串数据块 BLK1 中是否含有 ‘A’，若有则统计出 ‘A’ 的个数。假定字符串长度 $\leq 255$ 。

**分析：**根据题目要求，我们选用字节型取串指令 LODSB，即每次处理一个字符；采用正向操作；计数值存放在 CX 中；查找结果（‘A’ 的个数）存放在 BL 中。

```
Data SEGMENT

BLK1 DB 'SxyBA, MMKKAA%, ABCDEA'

LEN EQU $-BLK1

Data ENDS

Stack SEGMENT STACK 'STACK'

    DW 20 DUP (0)

Stack ENDS

CODE SEGMENT

    ASSUME CS: CODE, DS: DATA, SS: STACK

MAIN:  MOV AX, DATA

        MOV DS, AX

        MOV CX, LEN

        MOV SI, OFFSET BLK1

        XOR BL, BL

        CLD

NEXTA: LODSB

        XOR AL, 'A'

        JNZ NEXTB

        INC BL

NEXTB:  DEC CX

        JCXZ OK

        JMP NEXTA
```

```

OK:    MOV  AH, 4CH
        INT  21H

CODE    ENDS

        END  MAIN

```

如果我们选用 SCASB 指令，采用反向操作方式，相关的程序段如下：

```

        MOV  CX, LEN
        MOV  SI, OFFSET  BLK1
        ADD  SI, CX
        DEC  SI          ;; 置反向处理中的第 1 个数据单位首址
        XOR  BL, BL
        MOV  AL, 'A'
        STD
NEXTA:  REPNZ SCASB
        JNZ  OK
        INC  BL
        JCXZ OK
        JMP  NEXTA
OK:     MOV  AH, 4CH
        INT  21H

```

2. 在待传送数据块的 Data\_BLK 中，有 NUM 个字节数据，求其校验和，最后将其存放在 AL 中。

**分析：**采用“字节型”指令 LODSB，将每一个字节数据送入 AL 后，再计算校验和。所谓校验和是指不考虑 NUM 个数据相加的进位位的累加结果。下面的程序段采用反方向操作方式。

```

... ..

MOV  SI, OFFSET  Data_BLK
MOV  CX, NUM    ;; 字节个数送入 CX
ADD  SI, CX     ;; 操作首地址 = 数据首址 + 数据字节数 - 1
DEC  SI
STD
XOR  DL, DL    ;; (DL) = 相加部分和

```



```

NEXT:  LODSB

        ADD  DL, AL

        DEC  CX

        JCXZ  CHK_SUM

        JMP  NEXT

CHK_SUM: MOV  AL, DL

        . . . . .

```

## 10.3 其它非数值处理程序设计实例

例1. 试编写程序，利用模板型数据串 ‘|\*\*\*\*\*’，生成以下图示格式的字符串数据：

```
|*****|*****|*****|*****|*****|*****|
```

**分析：**从图示格式的字符串可见，由模板型数据串重复 6 次，再在其后加上 ‘|’，便可生成所需的字符串。

在实现中，我们如果将源串（模板型数据串）和目标串（生成的字符串）地址相邻安排。这样，就可以使用字符串传送指令，第一次将原串传送到目的串；第二次则将第一次刚生成的目的串，传送到下紧接的下一串，这样传送 6 次，共 48 个字符。

```

Data Segment

Pattern DB  '|*****'

NewStr  DB  49  DUP ( '|')  ;; 事先全部初始化为 '|.....|'

Data  ENDS

Stack Segment Stack

DW   20  DUP (0)

Stack ENDS

Code Segment

        ASSUME  CS: Code, DS: Data, ES: Data, SS: Stack

Main:  MOV  AX, Data

        MOV  DS, AX

        MOV  ES, AX

        MOV  SI, OFFSET Pattern

```

```

MOV ES, OFFSET NewStr

MOV CX, 24    ;; 传送“字”的个数→CX

CLD

REP MOVSW

MOV AX, 4C00H

INT 21H

CODE ENDS

END Main

```

例2. 设有 40 个组合型 BCD 码数据，试编写程序，对其按升序排列。

分析：

采用“每次找寻一个最小的数，并将其存放在本次找寻中的首位”的方法。具体地说，第一次从第 1—40 个 BCD 码中找出一个最小的，将其与第一个位置的数据交换；第二次从第 2—40 个 BCD 码中找出一个最小的，将其与第二个位置的数据交换；如此这样，进行到第 39 次时，从第 39 和第 40 这两个 BCD 码中找出一个最小的，将其与第 39 的数据交换。注意，程序中数据序号是从 0 开始的。

根据上述分析，查找中涉及两重循环：一是控制最小值查找次数的外循环，另一是从待查找数据中找出一个最小值的内循环。在外循环中，用 J 表示查找的次数（J 的初值为 0，终值为数据长度-1），对于每一次找出的最小值，与本次查找中的第一个数据交换。在内循环中，用 I 表示在，从这些数据中找出一个最小数的次数（I 从 J 到数据结束止）。用 L 记录在第 J 次查找中找到的最小值序号。

```

DATA SEGMENT

PBCD DB 89H, 34H, 56H, 78H, ....., 76H, 88H, 97H, 56H

LGTH EQU $ - PBCD    ;; 组合 BCD 码数据个数

DATA ENDS

STACK SEGMENT STACK

    DW 20 DUP (0)

STACK ENDS

CODE SEGMENT

```

```

        ASSUME  CS: CODE, DS: DATA, ES: DATA, SS: STACK

MAIN:   MOV  AX, DATA

        MOV  DS, AX

        MOV  CX, 0      ; ; 第 J 次外循环中, CX 是本次循环中第一个
                        ; ; 数据的序号。

NEXT:   PUSH  CX

        MOV  BX, CX     ; ; 预置当前最小值序号 L 于 BX

        MOV  SI, CX     ; ; 找出一个最小值的内循环 I 于 SI

LOP:    MOV  AL, PBCD[SI]

        CMP  AL, PBCD[BX]

        JNB  LOP1

        MOV  BX, SI      ; ; 保存当前最小值序号

LOP1:   INC  SI

        CMP  SI, LGTH

        JB   LOP         ; ; 在第 J 次中, 找出最小数值的次数控制。

        CMP  BX, CX      ; ; 确定本次查找中的最小数值的序号。

        JZ   LOP2

        MOV  SI, CX

        MOV  AL, PBCD[SI] ; ; 最小值交换

        MOV  AH, PBCD[BX]

        MOV  PBCD[SI], AH

        MOV  PBCB[BX], AL

LOP2:   POP  CX

        INC  CX

        CMP  CX, LGTH-1  ; ; 控制外循环次数

        JNZ  NEXT

; ; 结束循环

        MOV  AX, 4C00H

        INT  21H

CODE   ENDS

```

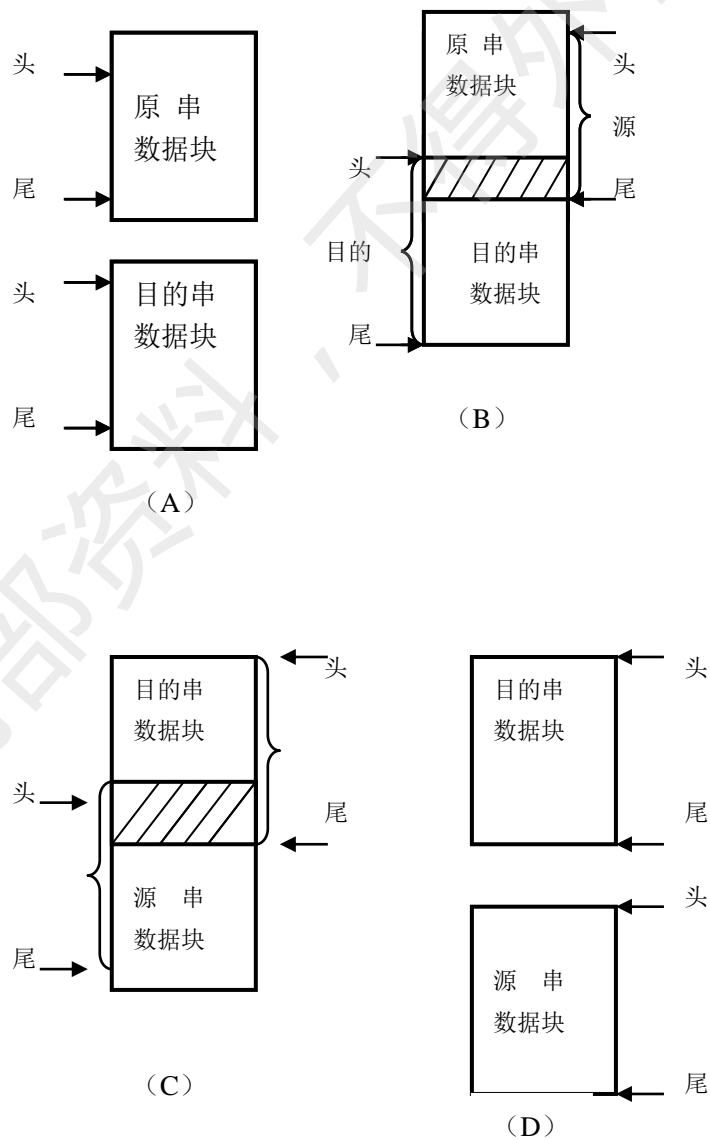
END MAIN

例 3：将 SRCBLK 中的偶数个字节数据，搬移到 DSTBLK 开始的内存单元中。

分析：

1. 由于被搬移的数据为偶数字节，故可以“字”为串操作单位，即一次搬移一个字，这样比一次搬一个字节的操作，要节约时间。
2. 操作方向问题

如下图示，原数据块和目的数据块间有以下 4 种关系。这些关系确定了可以采取的搬移方式（正向搬移或反向搬移），以确保在搬移过程中原始数据不受损害。



在图（A）和图（D）中，源串和目的串的地址不交叠（或分离），故无论正向或反向操作都不会在数据搬移过程中使数据受损。在图（B）和图（C）中，源串和目的串的地址有部分相互交叠（斜线部分）。对于图（B），如果采用反向操作方式，则在数据搬移过程中不会使数据受损。对于图（C），如果采用正向操作方式，则在数据搬移过程中不会使数据受损。这就是说，若目的串首地址低于源串首地址，则应选用正向操作方式；若目的串首地址高于源串首地址，则应选用反向操作方式。

3. 对于反向字操作来说，操作首地址=串首值+字节数-2，

搬移次数=字节数/2

DATA SEGMENT

SRCBLK DB '128976598WS JKOPB:?65#RSTwVWI234'

LGTH EQU \$-SRCBLK

BLANK DB LGTH DUP(0) ;; 空白数据块区

DATA ENDS

STACK SEGMENT STACK

DW 10 DUP(0)

STACK ENDS

CODE SEGMENT

ASSUME CS: CODE, DS: DATA, SS: STACK

MOV AX, DATA

MOV DS, AX

MOV ES, AX

MOV SI, OFFSET SRCBLK

MOV DI, OFFSET SRCBLK+10 ;; 源串和目的串相互交叠

CLD ;; 设置为正方向

CMP SI, DI ;; 比较源串和目的串首地址，确定操作方向

JC SETCX

STD ;; (SI) ≤ (DI)，反向处理

ADD SI, LGTH

SUB SI, 2

```
        ADD    DI, LGTH
        SUB    SI, 2
SETCX:  MOV    CX, LGTH
        SHR    CX, 1          ;; 字数送 CX, 假定 LGTH 是偶数
        REP    MOVSW
        MOV    AX, 4C00H
        INT    21H
CODE    ENDS
        END    MAIN
```

## 习题 10

1. 试用其它指令集模拟串操作指令 `REP STOSW`、`REP MOVSB` 的功能。
2. 试用其它指令集模拟串操作指令 `REPZ SCASW`、`REPNZ CMPSB` 的功能。
3. 设有由两位非组合型 BCD 构成的数据，存放在内存单元中（十位数存放在高地址）。用串比较指令，找出最大值，并给出它在串中的位置（用从 0 开始的序列号表示）。
4. 试编写一个将串中的空隔（20H）删除的程序。
5. 试编写一个程序，在一个指定字符前插入空隔（20H）的程序。

## 第 11 章 输入/输出程序设计

输入/输出设备是实现计算机系统与人或其它设备、系统之间进行信息交换的装置。又叫 I/O 设备、外部设备、外围设备、外设。例如：键盘、扫描仪、数码相机、鼠标、光笔、操纵杆、显示器、打印机、绘图仪、声卡、磁盘存储器等。

CPU 和主存称为主机，其它称为外设。

信息送入主机称为信息输入，简称输入，对应外设称输入设备；反之为输出。

通常外围设备通过一个转接电路与主机连接，这个转换电路即 I/O 接口电路，又称为输入/输出接口、I/O 接口、外围接口。

接口电路中的寄存器称为 I/O 端口，当 CPU 要与外设交换信息或对外设进行某种控制时，便借助于访问端口来完成。一个外设可以有多个 I/O 端口。8086/8088 的一个端口可以是 8 位或 16 位。

输入/输出接口是计算机的重要组成部分；汇编语言是输入/输出接口编程最有效的编程语言。

### 11.1 输入/输出指令

#### 11.1.1 I/O 端口的编址方法

##### 1、与存储器统一编址方法

I/O 端口与存储器共用一个地址空间，把一个 I/O 端口等同于一个存储单元。对 I/O 端口的访问等同于存储单元的访问。

##### 2、与存储器独立编址方法

I/O 端口与存储器分别使用相互独立的地址空间。这时，为了访问 I/O 端口，就必须使用专门的指令：输入指令和输出指令。

8086/8088 采用这种编制方式。其端口不分段，可寻找的端口范围为 64K（8 位端口）或 32K（16 位端口）。

## 11.1.2 输入/输出指令

### 1. 输入指令

指令格式: `IN      ACC , PORT`

其中, ACC 为 AL (字节) 或 AX (字) 寄存器, PORT 为输入端口地址。

### 2. 输出指令

指令格式: `OUT  PORT ,  ACC`

其中, PORT 为输出端口地址, ACC 为 AL (字节) 或 AX (字) 寄存器。

这两条指令都指定使用累加器 AL 或 AX, 它们均属于数据传输类指令。输入指令 IN 是把 PORT 所指定端口地址的一个字节或字传送给 AL 或 AX。而输出指令 OUT, 正好相反, 把 AL 或 AX 中的内容输出到 PORT 所指定的端口。使用这两条指令时, 有下面两种寻址方式来确定端口地址 PORT。

## 11.1.3 I/O 端口的寻址方式

### 1、直接 I/O 寻址方式

指令中直接给出端口地址。

`IN    AL , n`

`IN    AX , n`

`OUT    n , AL`

`OUT    n , AX`

其中  $n = 0 - 255$

### 2、间接 I/O 寻址方式

用寄存器 DX 的内容来指定端口地址。

`IN    AL , DX`

`IN    AX , DX`

`OUT    DX , AL`

`OUT    DX , AX`

在 IBM-PC 系统中, 实际上我们只可能使用间接 I/O 寻址方式去访问外部设备端口, 因为低地址 I/O 端口已被系统使用了。



## 11.2 主机与外设数据传送的方式

CPU 与一个外设交换的信息可分为状态信息 (Status)、控制信息(Control)和数据(Data)。状态信息和控制信息与数据是不同性质的信息,必须要分别传送。但在 8086/8088 中,只有通用的 IN 和 OUT 指令,因此,外设的状态也必须作为一种数据输入;而 CPU 的控制命令,也必须作为一种数据输出。为了使它们相互之间区分开,它们必须有自己的不同端口地址。所以,一个外设往往有几个状态端口、控制端口和数据端口。

主机与外设数据传送的不同方式需要相应硬件接口电路支持。

### 11.2.1 无条件传送方式

这种传送方式又称同步传送方式或立即传送方式。它只有在外部控制过程的各种动作时间是固定的,且是已知的条件下才能够应用。在传送信息时,已知外部设备是准备好的,所以不查询外设的状态。在输入时,就只给出 IN 指令;而在输出时,就只给出 OUT 指令。这种传送方式的优点是程序简单,硬件和软件都很省。但这种传送方式必须已知且确信外设已准备好的情况下才能使用,否则要出错。

### 11.2.2 程序查询传送方式

一般情况下,主机和外设的工作是不同步的,为了确保数据传送的可靠性,在作数据传送前,先查询外设的当前状态。在确认外设可以进行输入/输出操作之后,才用输入/输出指令完成相应的输入/输出操作;若外设未准备好,则 CPU 就等待。

### 11.2.3 中断传送方式

在上述的程序查询传送方式中,CPU 要不断地询问外设,当外设没有准备好时,CPU 要等待,不能做别的操作,这样就浪费了 CPU 的时间。而且许多外设输入或输出一个数据的速度很慢,在这个过程中,CPU 可以执行大量的指令。为了提高 CPU 的效率,可才用中断传送方式。CPU 不查询外设的状态,当外设准备好传送数据时,通过控制总线向 CPU 发出中断请求,CPU 停下当前的程序,处理中断事件;处理完成后又继续执行中断前被中断了的程序。

查询方式和中断方式比较，前者适合 CPU 与外设数据交换频繁的场所，而后者适合外设与 CPU 数据交换相对偶然的情况。

## 11.2.4 直接存储器传送方式(DMA)

从输入/输出指令中我们知道，惯常的输入和输出是需要 CPU 中转的。顾名思义，直接存储器存取方式是不通过 CPU 中转的传送方式。

DMA (Direct Memory Access) 是通过专门的 DMA 器件来实现的。在 DMA 操作期间，DMA 器件接替 CPU 控制总线。由于不通过 CPU 中转，DMA 方式适用于大量数据快速地输入/输出。

## 11.3 中断

### 11.3.1 中断的一般概念

CPU 正在执行一个程序时，系统有突然“事件”发生，若条件许可，CPU 就停止当前执行的程序；停止的地方叫“断点”，对“断点”做好标记和保护；然后去执行用于处理该突发“事件”的程序，即“中断处理程序”；“中断处理程序”执行完毕后又回到“断点”处继续执行原来的程序，这个过程就叫“中断”。使用中断控制技术可以及时处理计算机中的突发故障和偶然事件，可以协调主机与外设的工作速度矛盾，可以及时处理实时控制系统中的控制点现场信息。

### 11.3.2 中断源及中断类型码

引起中断的原因或来源叫中断源。8088/8086CPU 可以处理 256 个不同的中断类型（即中断源），给每个中断源一个编号，这个编号就叫中断类型码，用 N 表示， $N=0, 1, 2, \dots, 255$ 。

#### 1. 内部中断

由 CPU 内部引起的中断，称为内部中断，又称为软件中断。内部中断不受状态标志寄存器中 IF 标志位的影响。

(1) 由中断指令 INT 引起的内部中断

CPU 执行完一条  $INT\ n$  指令后, 会马上产生中断, 并调用系统中相应的中断处理程序来完成该中断功能, 该中断指令是一个双字节指令, 其操作码为  $CDH$ , 操作数  $n$  指出中断类型, 即  $N=n$ ;  $n=00H, 01H, 02H, 03H, \dots, FFH$ 。

## (2) 由 CPU 的某些错误引起的中断

除法出错中断: 在执行除法指令时, 若发现除数为 0 或商超过了寄存器所能表达的范围, 则立即产生一个  $N=0$  的中断, 执行的中断处理程序与中断指令  $INT\ 00H$  的中断处理程序相同。

溢出中断指令  $INTO$ : 当执行该指令时, 根据前面的运算, 若  $OF=1$ , 则立即产生一个  $N=4$  的中断, 执行的中断处理程序与中断指令  $INT\ 04H$  的中断处理程序相同; 若  $OF=0$ , 便不产生中断, CPU 继续运行原程序。溢出中断指令  $INTO$  是一个单字节指令, 操作码为  $CEH$ 。

## (3) 为调试程序而设置的中断

单步中断: 当状态标志寄存器中  $TF$  标志位置为 1 时, 每条指令执行后, CPU 自动产生  $N=1$  的中断, 即单步中断, 执行的中断处理程序与中断指令  $INT\ 01H$  的中断处理程序相同。产生单步中断时, CPU 自动地将  $PSW$  压入堆栈, 然后置  $TF$  为 0, 于是, 当进入单步中断处理程序后, CPU 就不是处于单步方式了, 它将按正常方式运行中断处理程序。在单步处理程序结束时, 原来的  $PSW$  从堆栈中取回, 又把 CPU 重新设成了单步方式。

使用单步中断可以一条指令一条指令地跟踪程序的流程, 观察 CPU 每执行一条指令后, 各个寄存器及有关存储单元的变化, 从而找出和确定产生错误的原因。

断点中断指令  $INT\ 3$ : 当执行该指令时, 产生一个  $N=3$  的中断, 执行的中断处理程序与中断指令  $INT\ 03H$  (操作码为  $CDH$ ) 的中断处理程序相同。断点中断指令  $INT\ 3$  是一个单字节指令, 操作码为  $CCH$ 。

使用断点中断可以一段程序一段程序地跟踪程序的流程, 每执行完一段程序后, 检查各个寄存器及有关存储单元的内容, 从而确定错误位置, 分析产生错误的原因。

## 2. 外部中断

非 CPU 内部原因引起的中断, 统称为外部中断。

### (1) 非屏蔽中断 NMI

由非屏蔽中断线  $NMI$  引起的中断, 称为非屏蔽中断。该中断不受状态标志寄存器中  $IF$  标志位的影响, 产生一个  $N=2$  的中断, 执行的中断处理程序与中断指令  $INT\ 02H$  的中断处理程序相同。

## (2)可屏蔽中断 INTR

由可屏蔽中断线 INTR 引起的中断，称为可屏蔽中断。该中断受 状态标志寄存器中 IF 标志位的影响。若 IF=1（即允许中断，开中断），可屏蔽中断的请求可被 CPU 响应；若 IF=0（即禁止中断，关中断），可屏蔽中断的请求就不能得到 CPU 的响应。中断允许标志位 IF 的状态可用指令 STI 和 CLI 设置为开中断和关中断。CPU 响应可屏蔽中断时，从数据总线上获取其中断类型码 N。

### 11.3.3 中断向量表

每一个中断类型都对应一个中断，每一个中断都有相应的中断服务程序，中断向量表就是中断服务程序的入口地址表。中断向量表放在主存储器最低 1KB 的地址空间中（即 0000:0000~0000:03FFH）。在 PC 中规定中断服务程序为 FAR 型，每个中断向量占用四个字节，其中两个低字节存放入口地址的段内偏移量，两个高字节存放入口地址的段基值。这样，一个断类型码为 N 的中断，其中断服务程序入口地址的偏移量就存放在  $4 \times N$  和  $4 \times N + 1$  存储单元中，段基值就存放在  $4 \times N + 2$  和  $4 \times N + 3$  存储单元中。

### 11.3.4 中断优先级

在中断服务程序中再次调用其它中断服务程序称为中断嵌套。为了解决中断能否嵌套等问题，将中断源进行优先级排序，从高到低的顺序为：除法中断、INT n、INTO、NMI、INTR、单步中断。

在同一时刻有多个中断源向 CPU 发送中断请求时，CPU 首先响应优先级最高的中断请求。

当 CPU 正在执行中断服务程序时，若有新的中断请求发生，CPU 则根据中断优先级的高低来确定是否响应这些中断。若新的中断请求的级别比当前执行的中断请求的级别要高，且满足响应条件，那么 CPU 便中断目前的中断服务程序，转而去执行新的中断服务程序；当新的中断服务程序执行完之后，再回过头来执行原来的中断服务程序。若新的中断请求的级别比当前执行的中断请求的级别要低或相等，那么 CPU 则继续执行原来的中断服务程序。

## 11.3.5 中断过程

### 1. 中断请求

中断源在满足其中断请求条件时，向 CPU 发送中断请求。每一个中断源都有其自己的中断请求条件。

### 2. 中断响应

CPU 每执行一条指令会自动地检测有无中断发生；如有中断发生，且满足中断响应条件时，CPU 将中断当前执行的程序并使用堆栈记忆最必要的返回信息，然后转到相应的中断处理程序。

当中断发生时，由硬件自动完成如下工作：

- a、获取中断向量 N
- b、标志寄存器（PSW）内容入栈
- c、当前代码段寄存器（CS）内容入栈
- d、当前指令计数器（IP）内容入栈
- e、禁止外部中断和单步中断（IF=0，TF=0）
- f、从中断向量表中取  $4 \times N$  的字内容送 IP，取  $4 \times N + 2$  的字内容送 CS
- g、转中断处理程序

### 3. 中断处理

中断处理由中断处理程序来完成，不同的中断源有不同的中断处理程序。

中断处理程序首先将其相关寄存器内容压栈，然后对相应的偶发事件作处理，在中断返回指令之前按后进先出的顺序恢复寄存器内容。

### 4. 中断返回

中断返回指令：IRET

IRET 使程序回到被中断程序的断点处继续执行原来的程序。执行 IRET 指令时，由硬件自动完成如下工作：

- a、被中断程序指令计数器（IP）出栈
- b、被中断程序代码段寄存器（CS）出栈
- c、被中断程序标志寄存器（PSW）出栈

## 11.4 几个常用 BIOS 中断调用

基本输入/输出系统 BIOS 提供了对主要 I/O 接口的控制功能。如键盘、显示器、打印机、串行通信等。这些功能通过软中断指令 INT 来调用，故称为中断调用，每个调用中又含有多个子功能调用。

### 11.4.1 键盘中断调用（INT 16H）

#### 1、从键盘读字符（00H 号功能）

调用参数：

AH=00H

返回参数：

AH=键盘扫描码

AL=字符的 ASCII 码

#### 2、读键盘缓冲区字符（01H 号功能）

调用参数：

AH=01H

返回参数：

若已有键按下：

ZF=0

AH=键盘扫描码

AL=字符的 ASCII 码

若无键按下：

ZF=1

#### 3、读键盘状态字节（02H 号功能）

调用参数：

AH=02H

返回参数：

AL=键盘标志单元的内容

bit0:Right-Shift 键按下

bit1:Left-Shift 键按下

bit2:Ctrl 键按下

bit3:Alt 键按下

bit4:Scroll Lock 键打开

bit5:Num Lock 键打开

bit6:Cap Lock 键打开

bit7:Insert 键打开

## 11.4.2 显示中断调用 (INT 10H)

### 1、设置光标位置 (02H 号功能)

调用参数:

AH=02H, BH=页号, DH=行, DL=列

返回参数:

无返回参数。

### 2、读光标位置 (03H 号功能)

调用参数:

AH=03H, BH=页号

返回参数:

CH=光标开始行, CL=光标结束行, DH=行, DL=列

### 3、屏幕初始化或上滚 (06H 号功能)

调用参数:

AH=06H, BH=滚入行属性

AL=上滚行数, AL=0 全屏幕为空白

CH=左上角行号, CL=左上角列号

DH=右下角行号, DL=右下角列号

返回参数:

无返回参数。

### 4、屏幕初始化或下滚 (07H 号功能)

调用参数:

AH=07H, BH=滚入行属性

AL=下滚行数, AL=0 全屏幕为空白

CH=左上角行号, CL=左上角列号

DH=右下角行号, DL=右下角列号

返回参数:

无返回参数。

## 5、读当前光标位置的字符和属性 (08H 号功能)

调用参数:

AH=08H, BH=显示页

返回参数:

AH=属性, AL=字符

## 6、在当前光标位置上写字符和属性 (09H 号功能)

调用参数:

AH=09H, BH=显示页

AL=字符, BL=属性, CX=字符重复次数

返回参数:

无返回参数。

## 7、在当前光标位置上只写字符 (0AH 号功能)

调用参数:

AH=0AH, BH=显示页

AL=字符, CX=字符重复次数

返回参数:

无返回参数。

# 11.4.3 打印中断调用 (INT 17H)

## 1、打印一个字符并回送状态字节 (00H 号功能)

调用参数:

AH=00H, AL=字符, DX=打印机号

返回参数:



AH=打印机状态字节

## 2、初始化打印机并回送状态字节（01H 号功能）

调用参数：

AH=01H, DX=打印机号

返回参数：

AH=打印机状态字节

## 3、回送状态字节（02H 号功能）

调用参数：

AH=02H, DX=打印机号

返回参数：

AH=打印机状态字节

# 11.4.4 串行通信中断调用（INT 14H）

## 1、初始化串行通信口（00H 号功能）

调用参数：

AH=00H, AL=初始化参数, DX=通信口号（0, 1）

返回参数：

AH=通信口状态, AL=调制解调器状态

## 2、向串行通信口写字符（00H 号功能）

调用参数：

AH=01H, AL=字符, DX=通信口号（0, 1）

返回参数：

写成功：(AH)<sub>7</sub>=0；写失败：(AH)<sub>7</sub>=1

(AH)<sub>0-6</sub>=通信口状态

## 3、从串行通信口读字符（02H 号功能）

调用参数：

AH=02H, DX=通信口号（0, 1）

返回参数：

读成功：(AH)<sub>7</sub>=0, AL=字符；写失败：(AH)<sub>7</sub>=1

#### 4、取串行通信口状态（03H 号功能）

调用参数：

AH=03H, DX=通信口号（0, 1）

返回参数：

AH=通信口状态, AL=调制解调器状态

### 11.4.5 时间中断调用（INT 1AH）

#### 1、读取时间计数器当前值（00H 号功能）

调用参数：

AH=00H

返回参数：

CX=时间计数器的高位字

DX=时间计数器的低位字

AL=0: 未计满 24 小时

=1: 已计满 24 小时

时间计数器计数一次的时间约为 55.072ms 。

#### 2、设置时间计数器的当前值（01H 号功能）

调用参数：

AH=01H

CX=时间计数器当前值的高位字

DX=时间计数器当前值的低位字

返回参数：

无返回参数。

### 11.5 几个常用的 DOS 系统功能调用（INT 21H）

DOS 的中断调用为程序开发提供了涉及系统和接口的常用功能，使用这些功能调用，可以简化在 PC 机上的编程。这里只介绍几个常用功能调用。

DOS 功能调用的基本格式：

MOV AH, <功能号>

## INT 21H

### 1、有回显单字符键盘输入（01H 号功能）

调用参数：

AH=01H

返回参数：

AL=输入字符，并在屏幕上显示该字符。

### 2、无回显单字符键盘输入（07H 号功能）

调用参数：

AH=07H

返回参数：

AL=输入字符

### 3、键盘输入到缓冲区（0AH 号功能）

调用参数：

AH=0AH, DS:DX=缓冲区首地址

(DS:DX)=缓冲区最大字符数

返回参数：

(DS:DX+1)=实际输入的字符数

(DS:DX+2)=输入的首字符

因输入字符串的长度应该是任意的,输入串不可能存放在空间很有限的寄存器里,所以在使用本功能前需要准备输入串的存储空间。这块存储区用 DS:DX 指向,且前两个字节用于存放编程者指定的最大串长度和功能调用后实际输入的串长度。

### 4、显示单字符（02H 号功能）

调用参数：

AH=02H, DL=输出字符

返回参数：

无返回参数。

### 5、显示字符串（09H 号功能）

调用参数：

AH=09H, DS:DX=输出字符串的地址,该字符串用 '\$' 结束。

返回参数:

无返回参数。

## 6、取日期（2AH 号功能）

调用参数:

AH=2AH

返回参数:

CX=年, DH=月, DL=日

## 7、设置日期（2BH 号功能）

调用参数:

AH=2BH, CX=年, DH=月, DL=日

返回参数:

无返回参数。

## 8、取时间（2CH 号功能）

调用参数:

AH=2CH

返回参数:

CH=时, CL=分, DH=秒, DL=1/100 秒

## 9、设置时间（2DH 号功能）

调用参数:

AH=2DH, CH=时, CL=分, DH=秒, DL=1/100 秒

返回参数:

AL=00H: 成功; AL=FFH: 无效

## 10、取中断向量（35H 号功能）

调用参数:

AH=35H, AL=中断类型码

返回参数:

ES:BX=中断向量

## 11、设置中断向量（25H 号功能）

调用参数:

AH=25H, AL=中断类型码, DS:DX=中断向量

返回参数：

无返回参数。

## 11.6 应用举例

**例 1：**编写通用发声子程序 MYSOUND。利用 PC 机系统板上的 2 号定时器和扬声器驱动系统及 BIOS 的时间中断调用（INT 1AH）来产生指定频率和指定时间长度的声音。

I/O 端口 61H 的第 0 位为 1 时，允许 2 号定时器工作；为 0 时，禁止 2 号定时器工作。

I/O 端口 61H 的第 1 位为 1 时，允许 2 号定时器驱动扬声器；为 0 时，禁止 2 号定时器驱动扬声器。

I/O 端口 43H 为 2 号定时器的控制寄存器，I/O 端口 42H 为 2 号定时器的计时常数寄存器。

为了方便以后的应用，现将 MYSOUND 编写成一个共用子程序，在程序的开始用伪指令 PUBLIC 说明。

假定发声的频率存放在 BX 寄存器中，由于定时器使用 1.19MHz 的基准频率，所以送往定时器 2 的计数值是 122870H 除以 BX 的商。这个商分两次送往定时器 2 的计时常数寄存器，先送低字节，后送高字节。当 (BX) = 0 时，表示不发声。

由于使用时间中断 INT 1AH 作延时，它的 1 个延时单位约为 55.072ms，所以发声持续的时间是 55.072ms 的倍数。假定这个倍数存放在 DX 寄存器中。

```

PUBLIC MYSOUND
PROCEDURE SEGMENT
    ASSUME CS:PROCEDURE
MYSOUND PROC    FAR
    PUSH    AX
    PUSH    CX
    PUSH    SI
    PUSH    DX          ; 暂存延时倍数
    CMP     BX, 0
    JZ      MYS1        ; 不发声转到 MYS1 处

```

```
MOV    DX, 12H

MOV    AX, 2870H

DIV    BX

MOV    BX, AX        ; 计数值暂存

IN     AL, 61H       ; 取 61H 端口状态值

MOV    AH, AL        ; 暂存状态值

OR     AL, 00000011B

OUT    61H, AL       ; 打开定时器 2 和扬声器

MOV    AL, 0B6H      ; 给定时器 2 送控制字

OUT    43H, AL

MOV    AL, BL        ; 给定时器 2 送计数值低字节

OUT    42H, AL

MOV    AL, BH        ; 给定时器 2 送计数值高字节

OUT    42H, AL

JMP    MYS2

MYS1:  IN     AL, 61H

        MOV    AH, AL

        AND    AL, 11111101B ; 关闭扬声器，表示不发声

        OUT    61H, AL

MYS2:  POP    BX        ; 取出延时倍数

        PUSH   AX

        MOV    AH, 00H    ; 读取时间计数器初值

        INT    1AH

        MOV    SI, DX

MYS3:  MOV    AH, 00H    ; 读取时间计数器当前值

        INT    1AH

        SUB    DX, SI    ; 计算时间计数器差值

        CMP    DX, BX    ; 与延时倍数比较

        JB     MYS3      ; 延时未到转 MYS3

        POP    AX
```

```

        MOV     AL, AH        ; 发声结束恢复 61H 断口状态
        OUT     61H, AL
        POP     SI
        POP     CX
        POP     AX
        RETF
MYSOUND ENDP
PROCEDURE ENDS
END

```

**例 2：**编写声音报警程序 MYBEEP。利用通用发声子程序 MY SOUND 按下面的要求发声：先产生频率为 600HZ 的声音持续 0.5 秒，后产生频率为 1000HZ 的声音持续 0.5 秒，再产生频率为 400HZ 的声音持续 0.5 秒，然后停顿 1 秒；按前面的规律重复 5 次。

每重复 1 次需要调用四次 MY SOUND 来发出四段不同的声音，在程序中把停顿处理成发频率为 0 的声音。由于 MY SOUND 的一个延时单位约为 55.072ms，所以 0.5 秒约为 9 个延时单位。

在程序的开始处用伪指令 EXTRN 说明 MY SOUND 是一个共用的外部子程序。

```

EXTRN   MY SOUND: FAR
STACK   SEGMENT PARA STACK 'STACK'
        DB      1024 DUP('STACK...')
STACK   ENDS
DATA    SEGMENT
FREQ     DW 600, 1000, 400, 0        ; 频率表
TIME     DW 09H, 09H, 09H, 12H     ; 延时倍数表
COUNT   DW 05H                    ; 重复次数
DATA     ENDS
CODE     SEGMENT
        ASSUME  CS: CODE, DS: DATA, ES: DATA, SS: STACK
MYBEEP   PROC     FAR

```

```

        PUSH    DS

        XOR     AX, AX

        PUSH    AX

        MOV     AX, DATA

        MOV     DS, AX

        MOV     ES, AX

        MOV     CX, COUNT      ; 置重复次数值

LOP1:   MOV     AL, 4            ; 每次发声的段数

        MOV     SI, OFFSET FREQ ; 频率表首地址

        MOV     DI, OFFSET TIME ; 延时倍数表首地址

LOP2:   MOV     BX, [SI]

        MOV     DX, [DI]

        CALL    MYSOUND

        ADD     SI, 2

        ADD     DI, 2

        DEC     AL

        JNZ     LOP2

        LOOP    LOP1

        RETF

MYBEEP ENDP

CODE    ENDS

        END     MYBEEP

```

### 例 3：编写演奏乐曲的程序 MYMUSIC。

在音乐体系中，音名与频率的关系是固定不变的。下表列出了大字组、小字组、小字 1 组、小字 2 组和小字 3 组各音名与频率的对应关系，其它音名与频率的关系可由此表推出（相邻两个音名的频率之比为固定常数  $1.0594631$ ，即  $2^{1/12}$ ）。C 调的 1 音的频率为  $261.626\text{HZ}$ ，2 音的频率为  $293.665\text{HZ}$ ，3 音的频率为  $329.628\text{HZ}$ ，4 音的频率为  $349.228\text{HZ}$ ，余类推，它们满足十二音律关系。



音名与频率对照表

大字组	音名	C	$\sharp C$	D	$\sharp D$	E	F
	频率(HZ)	65. 406	69. 296	73. 416	77. 782	82. 407	87. 307
	音名	$\sharp F$	G	$\sharp G$	A	$\sharp A$	B
	频率(HZ)	92. 497	97. 999	103. 826	110. 000	116. 541	123. 471
小字组	音名	c	$\sharp c$	d	$\sharp d$	e	f
	频率(HZ)	130. 813	138. 591	146. 832	155. 564	164. 814	174. 614
	音名	$\sharp f$	g	$\sharp g$	a	$\sharp a$	b
	频率(HZ)	184. 997	195. 998	207. 652	220. 000	233. 082	246. 942
小字1组	音名	$c^1$	$\sharp c^1$	$d^1$	$\sharp d^1$	$e^1$	$f^1$
	频率(HZ)	261. 626	277. 183	293. 665	311. 127	329. 628	349. 228
	音名	$\sharp f^1$	$g^1$	$\sharp g^1$	$a^1$	$\sharp a^1$	$b^1$
	频率(HZ)	369. 994	391. 995	415. 305	440. 000	466. 164	493. 883
小字2组	音名	$c^2$	$\sharp c^2$	$d^2$	$\sharp d^2$	$e^2$	$f^2$
	频率(HZ)	523. 251	554. 365	587. 330	622. 254	659. 255	698. 457
	音名	$\sharp f^2$	$g^2$	$\sharp g^2$	$a^2$	$\sharp a^2$	$b^2$
	频率(HZ)	739. 989	783. 991	830. 609	880. 000	932. 328	987. 767
小字3组	音名	$c^3$	$\sharp c^3$	$d^3$	$\sharp d^3$	$e^3$	$f^3$
	频率(HZ)	1046. 502	1108. 731	1174. 659	1244. 508	1318. 510	1396.913
	音名	$\sharp f^3$	$g^3$	$\sharp g^3$	$a^3$	$\sharp a^3$	$b^3$
	频率(HZ)	1479. 978	1567. 982	1661. 219	1760. 000	1864. 655	1975.533

在一首乐曲中，每个音符的音高由对应的音名来确定，而音长则由节拍来确定。假定在4/4拍中，十六分音符（1/4拍）持续1个延时单位，则八分音符（1/2拍）持续2个延时单位，四分音符（1拍）持续4个延时单位，二分音符（1拍）持续8个延时单位，全分音符（1拍）持续16个延时单位。

将音符的频率值和持续时间构成一个频率时间表，乐曲程序就根据这个表来发声。表中每个音符占3个字，分别表示频率、延时单位数、分隔符，频率为0表示休止，-1表示乐曲结束。在程序中设置一个速度调节因子来调节乐曲的演奏速度。

下面是演奏“卖报歌”的乐曲程序。

```

EXTRN    MYSOUND:FAR

STACK    SEGMENT PARA STACK 'STACK'
          DB      1024 DUP('STACK...')

STACK    ENDS

DATA     SEGMENT

MYTAB    DW 523, 2, ?, 523, 2, ?, 523, 4, ?, 523, 2, ?, 523, 2, ?, 523, 4, ?
          DW 440, 2, ?, 523, 2, ?, 587, 1, ?, 523, 1, ?, 440, 2, ?, 392, 2, ?
          DW 440, 2, ?, 523, 4, ?, 523, 2, ?, 440, 2, ?, 523, 2, ?, 440, 1, ?
          DW 392, 1, ?
          DW 349, 2, ?, 440, 2, ?, 392, 4, ?, 440, 2, ?, 440, 2, ?, 392, 4, ?
          DW 294, 2, ?, 349, 2, ?, 392, 4, ?, 587, 4, ?, 587, 2, ?, 523, 2, ?
          DW 440, 2, ?, 587, 2, ?, 523, 4, ?
          DW 523, 2, ?, 440, 2, ?, 392, 2, ?, 440, 2, ?, 523, 8, ?, 523, 2, ?
          DW 440, 2, ?, 392, 2, ?, 440, 2, ?, 523, 2, ?, 440, 2, ?, 392, 2, ?
          DW 440, 2, ?, 294, 2, ?, 349, 2, ?, 392, 2, ?, 440, 2, ?, 349, 8, ?
          DW -1

FACTOR   DB 3           ; 速度调节因子

DATA     ENDS

CODE     SEGMENT
          ASSUME CS:CODE, DS:DATA, SS:STACK

MYMUSIC  PROC    FAR
          PUSH    DS
          XOR     AX, AX
          PUSH    AX
          MOV     AX, DATA
          MOV     DS, AX
          LEA     SI, MYTAB      ; 频率时间表首地址
MYM1:    MOV     BX, [SI]        ; 取频率值

```

```
    CMP     BX, -1
    JE      MYM2          ; 演奏结束转 MYM2
    MOV     AX, [SI+2]    ; 取延时单位数
    MOV     AH, FACTOR    ; 取速度因子
    MUL     AH            ; 求得延时倍数
    MOV     DX, AX
    CALL    MYSOUND
    ADD     SI, 6
    JMP     MYM1
MYM2:  RETF
MYMUSIC ENDP
CODE   ENDS
      END    MYMUSIC
```

## 习题 11

1. 用软件延时的方法改编例 1 的 MYSOUND 通用发声子程序。
2. 编制演奏“松花江上”的乐曲程序。

部分的频率时间表可参考下面的数据段定义：

```
DATA    SEGMENT
MYTAB   DW 262, 3, ?, 330, 1, ?, 392, 8, ?, 523, 4, ?, 523, 3, ?, 392, 1, ?
        DW 440, 3, ?, 392, 1, ?, 440, 4, ?, 392, 8, ?, 262, 2, ?, 294, 2, ?
        DW 330, 8, ?, 440, 4, ?, 523, 4, ?, 392, 4, ?, 330, 8, ?
        DW 294, 2, ?, 262, 2, ?, 294, 8, ?, 440, 4, ?, 523, 4, ?, 392, 4, ?
        DW 349, 3, ?, 330, 1, ?, 294, 3, ?, 262, 1, ?, 330, 2, ?, 294, 2, ?
        DW 262, 8, ?, 262, 3, ?, 330, 1, ?, 392, 8, ?, 523, 4, ?
        DW 523, 3, ?, 392, 1, ?, 440, 3, ?, 392, 1, ?, 440, 4, ?, 392, 8, ?
        DW 262, 2, ?, 294, 2, ?, 330, 8, ?, 440, 6, ?, 523, 2, ?, 392, 4, ?
        DW 330, 8, ?, 294, 2, ?, 262, 2, ?, 294, 8, ?
        DW 523, 4, ?, 494, 3, ?, 440, 1, ?, 392, 4, ?, 349, 2, ?, 330, 2, ?
        DW 294, 6, ?, 330, 2, ?, 262, 8, ?, 523, 2, ?, 494, 2, ?, 440, 8, ?
        DW 587, 2, ?, 523, 2, ?, 392, 8, ?, 440, 4, ?, 440, 2, ?, 330, 2, ?
        DW 294, 4, ?, -1
FACTOR  DB 3
DATA    ENDS
```

3. 用计算机键盘代替钢琴键盘，编写一个八度音程的钢琴程序。