8080/8085 软件设计

C·A泰特斯 等著 张梅岗 译

·人民邮电出版社

[上册]

封面设计：邵　新

科技新书目：98-123·统一书号：15045·总3057-无6338·定价：2.20元

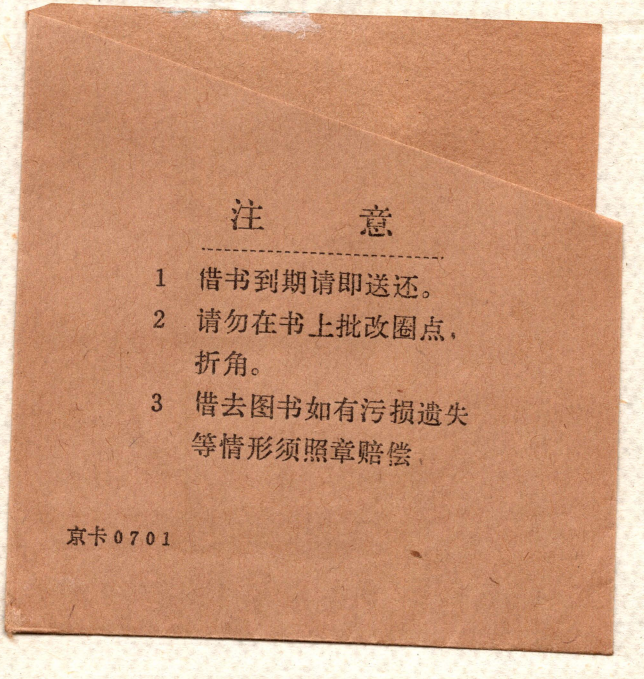
8080/8085软件设计

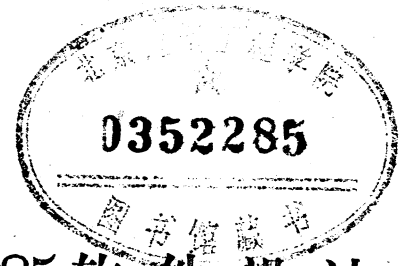
上　册

C. A. 泰特斯　等著

张　梅　岗　　译

符　明　何　诚　审校





8080/8085软件设计

上　册

C. A. 泰特斯　等著

张　梅　岗　　译

符　明　何　诚　审校

人民邮电出版社

8080/8085 Software Design
by christopher A. Titus, Peter R. Rony, David G. Larsen, and Jonathan A. Titus
Howard W. Sams & CO., Inc., 1979

内容提要

本书通俗地讲述软件设计，分上、下两册。上册主要讨论8080/8085的基本指令和高级指令、应用程序、系统程序。书中给出了很多程序设计实例。这些程序具有很强的通用性和实用性。通过这些实例，总结了8080/8085软件设计的经验和技巧，告诉读者如何根据具体情况，设计出可靠的、有效的软件。

本书在讲述上采用循序渐进的方式，适于不具备很多计算机知识的人员阅读。同时，可供从事计算机应用的工程技术人员、大专院校有关专业师生参考。

8080/8085软件设计

上 册

C.A.泰特斯 等著

张 梅 岗 译

符 明 何 诚 审校

*

人民邮电出版社出版

北京东长安街27号

河北省邮电印刷厂印刷

新华书店北京发行所发行

各地新华书店经售

*

开本：787×1092 1/32　　1985年7月 第一版

印张：15　页数：240　　1985年7月河北第一次印刷

字数：335 千字　　印数：1—20,000 册

统一书号：15045·总3057—无6338

定价：2.20元

译者的话

微型电子计算机体积小、重量轻、成本低、可靠性高，作为一种轻便而通用的电子控制装置，得到越来越广泛的应用。微型计算机开创了计算技术真正得以普及应用的新阶段，它不仅给电子工业和其它部门带来了深刻的影响，而且，它将对人类的物质文明起着重要的作用，将出人意料地改变人们的生活方式。

近几年来我国微型计算机的研制和生产，发展很快，主要产品有 DJS-030、DJS-040 DJS-050 和 DJS-060 几个系列。DJS-050 相当于美国 Intel 公司的 8080A 微处理机系列。这种处理机及其系列部件，我国已经能够批量生产，今后的问题主要在于推广应用。随着微型计算机的广泛应用，软件显得越来越重要。

«8080/8085 软件设计» 一书，通俗地讲述软件设计，内容丰富。全书分上、下两册出版。书中不仅讨论了 8080/8085 的基本指令和高级指令、应用程序、系统程序和文件库，而且还介绍了组装 8080/8085 微型计算机系统所需要的硬件及其组装方法，所以它可以作为微型计算机应用的参考书。我们还要特别指出，书中给出了三百多个程序设计实例，具有广泛的通用性和实用性，拿到机上就可以运行，解决实际问题。因此，本书

可作为程序设计手册使用。«8080/8085 软件设计» 总结了这种机器的软件设计的经验和技巧，告诉读者如何根据应用环境和现有资料，在充分理解和分析问题的基础上设计出可靠的、有效的、完善的软件。本书提供了许多改进软件设计的技巧，可供程序设计者借鉴。本书可供大专院校计算机专业的师生、广大工程技术人员和计算机爱好者作参考。

最后，需要对本书程序中的指令格式作一点说明。本书仍采用了原著者所使用的指令格式。严格地讲，这种书写格式是不很恰当的。如原书中 MOVAE 的正确格式应为 MOVA, E。读者在书写程序或使用本书所给的程序时，应采用图 3-6 给出的指令格式。

湖南大学谢卓杰教授认真地审阅了全书，在此表示衷心感谢。

我还要特别感谢顾徐珍同志，她协助我工作，付出了辛勤的劳动。

译者衷心希望广大读者对本书的错误和不妥之处给予批评指正。

一九八三年十月

序　言

英特尔（Intel）公司制出了四位字长的微处理器“芯片”4004，从而向人类世界引入了微处理器和微型计算机的概念。这在当时看来还不象是一场革命。随着对早期集成电路的改进，微型计算机革命得到了有力的推动，越来越多的厂家开始生产微处理器集成电路和微型计算机。目前，这些由一种化学元素——“纯化的砂子”作成的器件，实际上已经开始改变着人们的生活方式。

在超级市场，电子现金出纳机不仅能完成机械现金出纳机的各种任务，而且还能进行编目控制。有些缝纫机再也不需要用复杂的齿轮和凸轮去完成特殊的针法了。当需要使用特殊的缝针法时，可以用微型计算机来控制针的位置。在不久的将来，你甚至可以“教”缝纫机做最新式的服装。

计算器可以取代计算尺以完成复杂的计算任务，目前计算器的功能要比一、二十年前生产的先进的计算机更强。钟表不再需要上发条了，术语“顺时针”和“逆时针”不久也许将成为历史名词。今天，你甚至可以使用一种电子记录现有余款的支票薄。再也不必每月都进行结账了。在不久的将来，微波炉“知道”的烹调技术也许比你知道的还要多。

当然，使用微型计算机解决某个问题，需要硬件和软件。

硬件包括微处理器集成电路、某些存储器集成电路、发光二极管的接口电子器件，以便微型计算机能对外围设备（指示灯、电动机、阀门、线圈和仪表）进行控制。软件由微型计算机执行的指令序列组成，有了它，微型计算机才能处理数据或控制外部设备。遗憾的是，软件的设计和实现并不象硬件的设计和实现那样确定。

现在，有些普通微型计算机的用户用 BASIC 程序设计语言给计算机编程序。要这样做，就必须把 BASIC 的解释程序（一种汇编语言程序）存入到 5000 到 8000 个存储单元中。有了这种汇编语言程序，你就能把 BASIC 程序装入微型计算机，并且实际地加以执行。象世界上的其他任何事物一样，用 BASIC 语言给微型计算机编程序，有优点，也有缺点。由于用 BASIC 语言编程序存在着缺点，因此，在微型计算机上进行程序设计时，常常使用汇编程序设计语言。这种语言正是我们要在本书讨论的程序设计语言。

毫无疑问，汇编程序设计语言在教和学两方面都比较困难。你不可能用汇编程序设计语言在 10 分或 15 分钟内编写一个程序，让计算机算出 1 到 1000 之间的各个数的立方根（有效数字为 6 位）。如果使用 BASIC 程序设计语言，也许能做到这一点。但是，使用汇编程序设计语言是有优点的。事实上，用汇编语言可以完成的许多任务，用 BASIC 语言则不能完成。特别是在过程控制、外围设备控制、高速计算和实时数据采集等方面更是如此。实际上，装有微型计算机的大多数消费电子产品都用汇编语言编制程序；例如：电视游戏机，微波炉，缝纫机，电子现金出纳机（出售点终端），煤气泵，血压监控器等等。所以，汇编程序设计语言是一种功能很强而有用的语言，值得学习。

在本书的第一章，我们并不讨论汇编程序设计语言的程序设计，而只是介绍 8080 和 8085 微处理器集成电路的特性；包括这两种微处理器芯片内的寄存器。这些寄存器是很重要的，因为你在用汇编程序设计语言编程序时要反复用到它们。紧接着的三章，讨论 8080 和 8085 能实际执行的汇编语言程序指令。由于你对理解长而复杂的程序用的各条重要指令还不够熟悉，所以在头几章，读者见不到长而复杂的程序。掌握了这些重要指令之后，读者才能开始用汇编程序设计语言给微型计算机编写程序，完成有用作业。本书其余各章讨论 8080 汇编程序设计语言指令的应用，诸如算术运算、数制转换、输入/输出设备（外围设备）控制等。

本书具有一些与众不同的特点，我们相信读者对这些特点会深为赞赏。我们并没有用一章来专门讨论二进制、八进制和十六进制各种数制。我们只在某些章节中告诉读者如何用纸和笔来做这些数制的转换运算。但是，我们自己更感兴趣的是微型计算机为完成上述运算所必须执行的指令序列。如果读者需要了解更多的有关基本算术运算的知识，可以参考有关计算机程序设计的其他著作。读者还会发现，凡是以 8080 微处理器为基础而组装的微型计算机，大都可以使用本书给出的程序。我们在本书给出的硬件或软件例子并不只是代表某个厂家的硬件特征。本书所给出的程序例子，下列公司的 8080 微计算机都可以运用：*MITS*公司，处理机技术公司(Processor Technology)，英特尔公司（Intel Corporation）国家半导体公司(National Semiconductor corporation)，数字团体公司(Digital Group)，控制逻辑公司(Control Logic)，IMS公司(IMS Associates)和E&L仪器公司(E&L Instruments)。当然，各个微型计算机系统的外围设备可以各有不同，正如各个系统

的外围设备的地址可以各有不同一样。

在输入/输出这一章，即第七章，我们给出了用软件控制的外围设备的电路方框图。如果读者不知道外围设备怎样具体与微型计算机连接，那么控制该外部设备的程序软件也就没有什么用处了。此外，如果读者希望在自己的微型计算机上应用某个具体程序，则也许需要了解该程序控制的外围设备的电路方框图，以便照着进行硬件的连接。

本书的前三章讨论八进制和十六进制数形式的指令操作码。我们比较喜欢用八进制数表示操作码。因为我们深深地感到，用八进制数表示 8080 的汇编语言程序指令的操作码比较容易记住，而且，当读者检查八进制数操作码时，很容易确定这条指令与哪些寄存器或存储器的地址单元有关联。如果用十六进制数表示操作码，作出这种判断是非常困难的。不少程序员却比较喜欢用十六进制数表示操作码，因为他们只需要记住两位数字的操作码，而不需要记住三位数字的操作码。因此，我们只能采取在程序例子中两种数制都用的办法。我们建议：程序员初次给 808 0或 8085 微型计算机编写程序，最好使用八进制数，因为八进制数比较容易记住。日后，使用十六进制数也许更方便。如果读者确定不了使用哪种数制，则最好两种都试一下，然后选用自己感到最方便的一种。在本书经常可以看到十六进制数，它们是用括号括起来的；例如：(5F)。

我们的另一个目的，就是要对本书的程序例子如何操作给出详细的说明解释。我们并不说："这儿的程序，你捉摸捉摸它是如何操作的。"如果采用这种方法，读者可能所得甚少。相反地，我们从能完成某个具体任务的最简单的指令序列开始，逐步开发出一个个程序。这些简单程序常常有某些局限性；碰到这种情况，我们就给它增添几条指令，使微型计算机能更好

地完成更通用的任务或某一特定的任务。因此，通过学习以前的例子，我们就能对某个问题设计出最佳的解题程序。

读者通览本书后可以发现，书中程序例子的计算机打印输出格式与其他书的作者所用的格式有所不同。我们选择的这种格式，有助于初学者掌握汇编语言程序设计的概念。其中最重要的概念之一是，必须把多字节的指令存储在连续的存储单元内。如果读者看一下其他书的程序例子的计算机打印输出，则很难掌握这一概念。正是这个原因，我们编写了常驻编辑程序/汇编程序(TEA)，用来产生程序表；我们认为，对于一本关于汇编语言的程序设计的书，这种做法是最合适的。值得强调的是，读者只要熟悉汇编语言程序设计的适当的方法，无论用什么编辑程序/汇编程序软件包来开发汇编语言程序都是可以的。请记住，编辑程序和汇编程序只不过是达到目的的手段。它们是用来帮助我们开发适当操作的汇编语言程序的工具。

软件的优秀特征之一是容易复制和分布使用。正是由于这一特点，不少专业杂志和业余杂志刊登了许多软件中的程序表，提供源程序和目标程序纸带或音频录音磁带。另外，还有许多软件库，其中包括：英特尔用户库 (Intel User's Library, 3065 Bowers Ave, Santa Clara, CA 95051)，微型计算机软件库(The Microcomputer Software Depository, 2361 E • Foothill Blvd, Pasadena, CA 91107)。

«8080/8085 软件设计» 所阐述的有关程序设计的许多概念已经编入了大学研究生班的教材，由布莱克斯堡的 Tychon 公司发行。该教材分三个教程：微型计算机设计 (626)，微处理器接口技术(628)，和 8080/8085 微处理器的软件设计(690)。如果读者希望得到这些教程，可以写信向 Tychon 公司的教程部经理索取 (P. O. Box 242, Blacksburg, VA 24060)。也可

以通过连续教育中心和弗吉尼亚工学院州立大学 (Blacksburg, VA 24061)，希望他们提供。为了了解更多的情况，请打电话给Linda Leffel博士(703)(961—5241)。

C.A.泰特斯
P. R. 罗尼
D. G. 拉森
J.A.泰特斯

目　录

程序例目录

第　一　章

第　二　章

第　三　章

第 四 章

第 五 章

第 六 章

第 七 章

程序例目录

第一章

第二章

第 三 章

第 四 章

第 五 章

第 六 章

第 七 章

第一章　8080/8085微处理器介绍

本章的目的是向不熟悉 8080/8085 微型计算机工作原理的读者介绍 8080 和 8085 微处理器以及 8080/8085 微型计算机。这种介绍非常详细，以便读者能够懂得 8080 和 8085 微处理器集成电路的操作，掌握微型计算机最小系统的各项功能。但是，并不能指望只读了这一章就能成为应用这类集成电路的专家。

8080 微处理器集成电路是于 1973 年由英特尔(Intel)公司最先制造出来的。它是由该公司以前出产的 8008 八位微处理器发展而来的。现在，因为 8080 的应用非常普遍，所以许多“第二手”的厂，如先进微器件公司 (Advaned Micro Device Inc)，NEC 微型计算机公司(NEC Microcomputer Inc)，德克萨斯仪器公司(Texas Instruments Inc)，国家半导体公司(National Semiconductor Corporation)等等也都生产 8080 微处理器。8080 问世后刚刚五年，单块 8080 微处理器集成电路的价格已从 360 美元降低到 10 美元，这是令人兴奋的。

8080 微处理器

8080 微处理器集成电路可以用作微型计算机系统的“心脏”，它能够执行 72 种不同类型的指令，总数为 244 条。8080 微处理器的大部分集成电路是控制逻辑和判定逻辑，因此，如果读者希望组装一个微型计算机小系统，只要增添几块集成电

路就行了。这些附加控制逻辑电路的功能之一就是给8080微处理器提供双相非重叠的时钟信号(该时钟信号的电压为0～+12伏)。8080微处理器也需要三种电源(+12伏，+5伏，和−5伏)。8080的其他输入电平(两个时钟信号输入电平除外)和全部输出电平都与晶体管—晶体管—逻辑电路(TTL)的标准电压电平兼容。

综上所述，8080微型计算机小系统由8080微处理器和几块用来提供两相时钟和附加控制信号的支持集成电路组成。这些控制信号使8080微处理器能对存储器集成电路和输入/输出(I/O)设备进行存取存储器保存指令码和数据。8080使用输入/输出设备与"外界"通信。8080借助16条地址总线对存储器寻址，寻址的能力为2^{16}，即65,536个存储单元(64K)。每个存储单元能够存储一个8位字节。

数据通过8位双向数据总线在8080微处理器与存储器或输入/输出设备之间进行传送。8080微处理器和存储器之间的数据流是由8080给出的两个控制信号控制的。这两个控制信号通常叫做$\overline{\text{MEMORY READ}}$ ($\overline{\text{MR}}$或$\overline{\text{MEMR}}$，存储器读)信号和$\overline{\text{MEMORY WRITE}}$ ($\overline{\text{MW}}$或$\overline{\text{MEMW}}$,存储器写)信号。当8080微处理器从存储器读出指令或数据时，它必须在地址总线上给出一个16位的地址，并且在"存储器读"信号线加一个脉冲。然后，存储器把一个8位的数据值经数据总线传送给8080微处理器。当8080微处理器向存储器写入数据时，它要再提供一个16位的地址，但是，这一次它把一个8位的数据字节置于双向数据总线，然后在"存储器写"信号线上加脉冲。

大多数微型计算机系统所拥有的存储器集成电路有两种基本类型。它们是读/写存储器(R/W)和只读存储器(ROM)。如

果必须把信息存储在存储器，而在稍后的时间由程序改变信息，那么，应该把这种信息存储在读/写存储器。一般而言，用ROM存储程序步(指令)和不需要改变的数据值。

8080 微处理器还包括一些内部寄存器，这些内部寄存器好象读/写存储器一样。8080 微处理器芯片内共有七个通用寄存器。这就是说，程序设计员可以利用这七个寄存器编写程序，把数据值计算的某一操作结果，甚至地址信息暂时保存在这些寄存器中。每个寄存器能存储 8 位的信息。为了便于对8080进行程序设计，我们给每个寄存器分别用一个字母命名：A、B、C、D、E、H、L。A寄存器又叫做累加器，因为能用它来累加 8080 进行的算术或逻辑运算的结果。如果一个 8 位的数据值装在A寄存器里，那么，它能向左或向右循环移位；如果该数据值装在其他任何通用寄存器中，就不能做到这一点。A寄存器一般用来从输入设备(如键盘、检测器、开关、或模-数转换器)接收数据。在程序的控制下，也可以把A寄存器的内容输出给诸如数—模转换器、指示灯、控制器和打印机等外部设备。

与A寄存器相关的标识位有五个。当一条 8080 逻辑指令或算术指令被执行之后，这些标识位分别被置位或清零。三个标识位分别用来表示奇偶校验，运算结果的符号和运算结果是否为零。另外有两个进位标识位，分别用来表示进位和辅助进位。通常程序设计员把这个 8 位标识字(实际上该标志字只用5位)和A寄存器一起称为处理器状态字，即 PSW。也许这五位标识位的最重要的特征是，可以给 8080 编程序来测试一位或几位标识位的状态。根据检测结果，8080就能决定是否执行某一指令序列。这些标识位将保持置位或清零状态，直到另一个逻辑运算或算术运算使它们的状态改变时为止。

8080 微处理器集成电路还有另外三个重要寄存器，值得我们注意。这三个重要寄存器是堆栈指示器(SP)，指令寄存器(IR)和程序计数器(PC)。这些寄存器不是通用寄存器，因为它们只能用来执行特定的任务，不能用作为数据暂存器。

堆栈指示器(SP)是一个 16 位的寄存器，它用来存储通常与读/写存储器相关联的一个 16 位的地址。8080 可以执行指令把通用寄存器的内容存入“堆栈”。这时堆栈指示器给 8080 提供一个地址，以便能够把这些寄存器的内容存储在读/写存储器里。因此，堆栈(即栈区)只不过是读/写存储器的一部分，被划分出来作为暂存器。也可以把地址信息保留在堆栈中。为了使用堆栈，必须把和读/写存储器相关联的一个地址装入堆栈指示器。当程序中有调用子程序的情况时，堆栈对于存储返回(即连接)的地址格外有用。

16 位程序计数器(PC)和 8 位指令寄存器(IR)实际上控制程序的执行序列，所以它们是密切相关的。8080微处理器执行指令时，指令寄存器具体确定执行什么操作，以及数据从哪里来，送到哪里去，即具体确定 8080 内部以及它和存储器，及外围设备的数据交换。程序计数器用来提供要执行的下一条指令的存储地址单元。我们已经讨论的几个寄存器以及尚未介绍的许多寄存器如图 1-1 所示。

8080 微处理器的复位是由它的 RESET(复位)引线接地来实现的。复位后，程序计数器 PC 被清零。因为 PC 始终指示到存储下一条要被执行的指令的存储单元，所以复位后执行的第一条指令一定存储在存储器的第 0 号地址单元。因此，复位后程序计数器的 16 位内容置于 16 位地址总线，把存储在该存储单元的这条指令从存储器取出，写入 8 位指令寄存器 IR。然后与指令寄存器 IR 有关的逻辑电路把这条指令译码，确定要

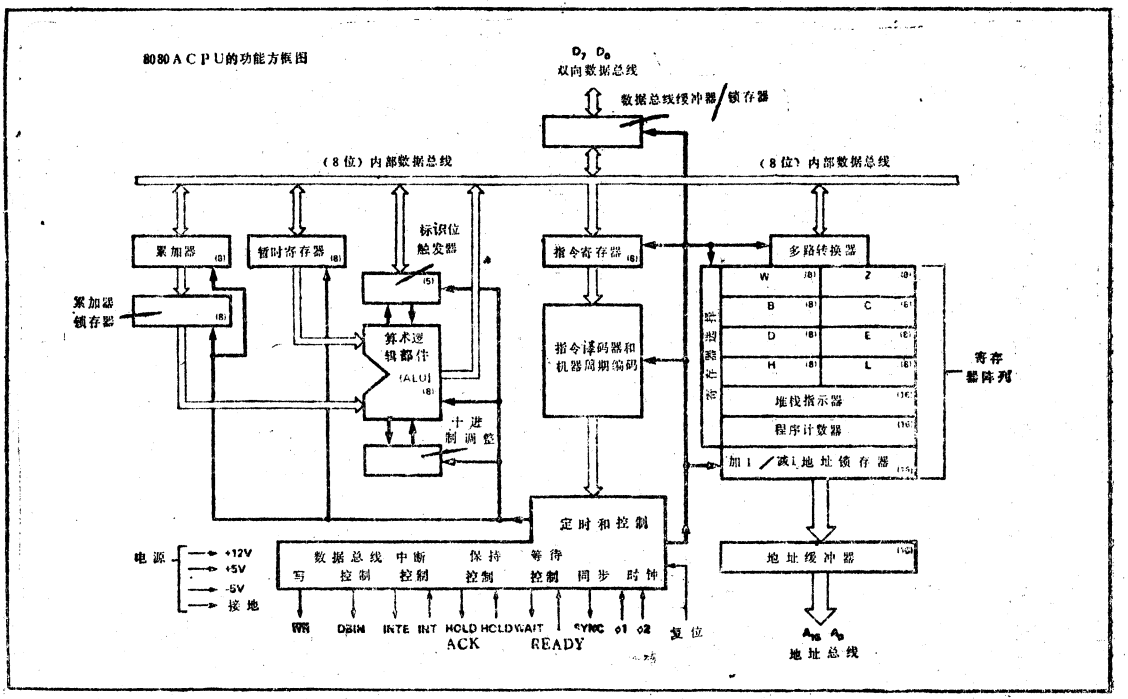


图 1-1 8080 微处理器集成电路的方框图

被执行的这条指令需要做什么操作。这条指令被译码后，8080微处理器可以把数据从一个寄存器送到另一个寄存器，可以把数据向外围设备输出，或者，可以检测标识字中某个标识位的状态。

当指令被取出并执行后，程序计数器 PC 的内容加 1，从而指示到存储下一条要被取出并执行的指令的存储单元。程序计数器的内容不仅可以复位为零，或加 1，而且还可以不按顺序装入任何一个 16 位的地址。这种操作一般在 8080 微处理器分支(转移)到存储在存储器的另一部分的指令时发生。

关于产生上面提到的控制信号所需要的电路器件，以及用 8080 微处理器组装一个微型计算机小系统所需要的更详细资料，请参考 Bugbook 8080 A®：«微型计算机接口技术与程序设计»(Microcomputer Interfacing and Programming[1])。

8080 微处理器可以用怎样的速度处理数据或与外围设备通信呢？这个问题决定于被执行的指令。所有的指令执行需要的时间是已知的，但是并不是所有指令的执行时间都相同。指令执行的速度是由两相时钟频率来决定的。8080 微处理器的时钟频率是 500 KHz～2 MHz。频率为 2 MHz 时，执行一条最简单的指令所需要的时间是 2 微秒(μs)，执行一条最复杂的指令所需要的时间是 9 微秒。一般而言，大多数指令的执行时间大约是 4～5 微秒，每一条指令所需要的执行时间是已知的，这些时间在 8080 生产厂的大多数有关 8080 的用户文献资料上都有记载。

Bugbook 是 E&L 仪器公司(Derby，CT 06418)的注册商标

机器语言和汇编语言

为了使8080微处理器能够完成一定的任务，必须给它编制程序。这就是说，必须把指令序列存储在8080微处理器的存储器里，从第0号地址单元开始存储。这些指令可以存储在只读存储器ROM中，也可以存储在读/写存储器中。使用哪种存储器，对于8080微处理器来说都无关紧要。例如，存储在存储器的8080用程序如下所示：

例 1-1 8080微处理器的典型程序

二进制数	八进制数	十六进制数
00111100	074	3C
11111110	376	FE
10000000	200	80
11000000	302	C2
00101101	055	2D
00111000	070	38

在这个例子中，用二进制数、八进制数和十六进制数列出了六个连续存储单元的内容。这就是说，00111100_2，074_8和$3C_{16}$这三个数是相等的。有时，在数的右下角用“H”表示十六进制数，例如，$3C_H$。

从上一段程序的存在形式，很难看出该程序要做什么。8080微处理器实际上可以执行244条不同的指令，每一条指令都是用不同的操作码来代表的。244条指令的操作码都可以用二进制数、八进制数或十六进制数来表示。使用哪种数制都

可以。但是，必须把适当的指令序列存储到存储器，才能使程序正确地执行。

可是，用上面这种方式来给 8080 微处理器编程序非常困难。因为必须把 244 条指令的操作码都记住。程序设计员给 8080 编制程序，常常不是用数字，而是用助记符。助记符只是 8080 微处理器执行指令时所做操作的缩写符号。例如，助记符 INRA 是 "increment the eight-bit content of the A register by one"(A 寄存器的 8 位内容加 1)的缩写。表 1-1 列出了几个助记符和 8080 所执行的相应操作。

表 1-1　　一些典型助记符及其操作

助记符	操作
DCRA	A寄存器的 8 位内容减 1。
JMP	转移到另一个存储单元，从该存储单元开始继续执行程序。
PUSHB	把寄存器对 B 的内容保存(压)到堆栈。
XCHG	交换寄存器对D和H的内容。
OUT	把A寄存器的内容输出到指定的外围设备。

助记符的作用是什么呢？程序设计员只要见到助记符，就很容易想到微处理器要做什么操作。例 1-2 同时列出了 8080 的一些助记符及其相应的二进制数。八进制数和十六进制数操作码。

即使读者不知道这些助记符是什么意思，现在也许可以意识到，理解一个程序表的助记符比了解一个程序表的二进制数、八进制数或十六进制数容易。

读者给 8080 微处理器编程序，直接把助记符存入存储器行吗？不行。只能把与助记符相应的二进制数操作码存入存储器。

例 1-2　8080 的几条指令助记符及其操作码

二进制数	八进制数	十六进制数	助记符
00111100	074	3C	INRA
11111110	376	FE	CPI
10000000	200	80	〈数据〉
11000010	302	C2	JNZ
00101101	055	2D	〈LO〉
00111000	070	38	〈HI〉

就连八进制数和十六进制数表示法也是二进制数代码的简写形式。助记符实际上是程序设计员的“拐杖”。用助记符编写程序，比较容易理解指令执行的序列以及它们的总的作用。但是，只能把指令的二进制数代码存入存储器。所以，程序设计员必须把助记符翻译成适当的操作码。为完成这种翻译工作，有几种方法可以使用。

虽然读者可能还没有实际经验，但现在已经了解到机器语言和汇编程序设计语言之间的差别。程序员用机器语言来编写程序，就只好与二进制数、八进制数、或十六进制数形式的指令操作码打交道。程序设计员可以把包含操作码序列的程序直接存入 8080 微处理器的存储器，然后加以执行。程序员用汇编程序设计语言编程序，就是写出要执行指令的助记符序列。确定出助记符的适当的二进制数操作码，再把二进制码存入存储器，该程序才能够被执行。因为理解和记住助记符比二进制数、八进制数或十六进制数操作码容易，所以，本书后面各章节的所有程序都是助记符的形式。这就是说，所有程序都是用汇编语言编写的。

8085 微处理器

8085 微处理器集成电路是 8080 的改进型产品(见图1-2)。大部分改进之处都影响到以 8085 微处理器为基础的微型计算机的硬件设计。8085 只需要一种+5 伏的电源，而 8080 需要三种电源(+12 伏，+5 伏和−5伏)。8085 的这一特性很好。但是，不少存储器件和接口器件除了需要+5 伏电源以外，还需要附加其他电源。

8085 的时钟输入条件也简化了，不再需要 0～+12 伏的非重叠的两相时钟信号。我们可以直接把晶体或阻容结构与 8085 连接。这种功能与 MOS Technology(MOS技术)公司所生产的 6502 相似。

8080 微型计算机系统所需要的大部分控制逻辑在 8085 微型计算机系统设计上可以不要。但是，为了产生控制信号。它仍然需要一两块集成电路来控制 8085 产生的某些信号。

8085 只有 8 条地址输出线，而 8080 有 16 条地址输出线。但是，这两种微处理器都能够直接寻址 65,536 (64 K)存储单元。只有 8 条地址线的 8085 能够寻址 64 K存储单元，是因为它的 16 位地址是多路转换的。8085 微处理器的 8 位地址线专门用来传送地址的高 8 位(A_8～A_{15})，地址的低 8 位(A_0～A_7)是与在多路转换的地址/数据总线上的 8 位数据位(AD_0～AD_7)多路转换的。很容易把地址/数据总线上的低 8 位地址位锁存起来，因为有一个正向地址锁存器启动(ALE)脉冲，这个脉冲就是 8085 为这一用途而产生的。

把一组 8 位总线上地址和数据信号多路复用，就省出了微

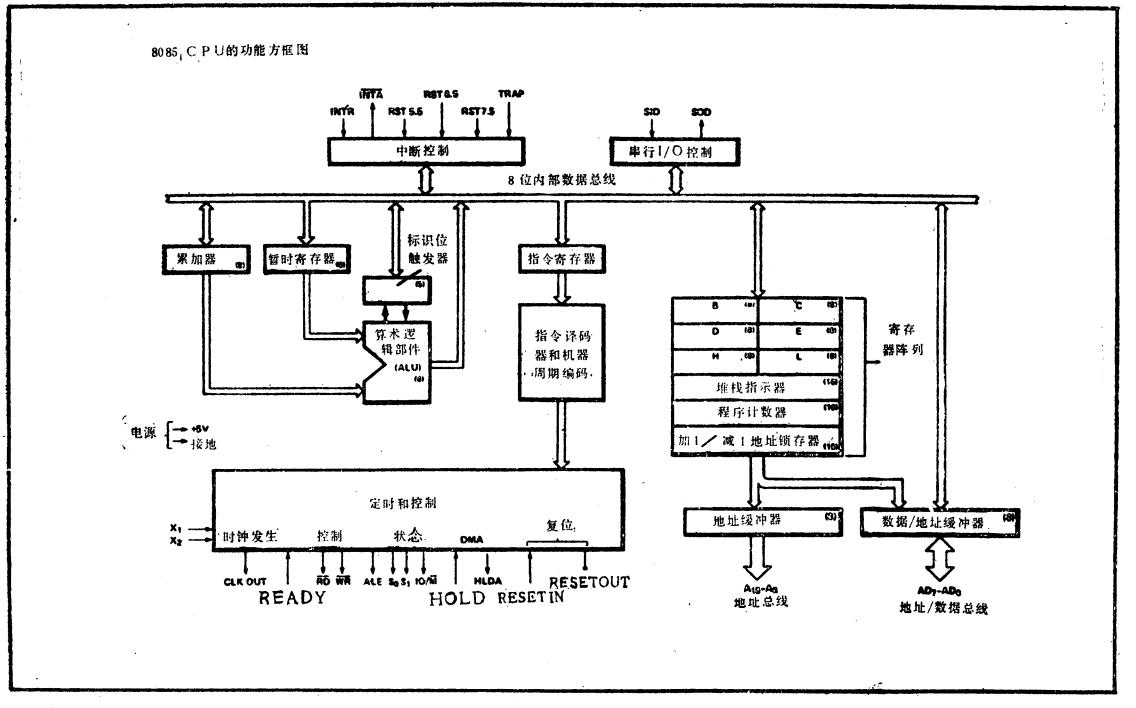


图 1-2　8085 微处理器集成电路的方框图

处理器集成电路芯片的几条引线，可供扩展 8085 的中断能力用。关于这个问题，我们将在后面有关中断的一章详细讨论。

8085 微处理器的最后一个重要改进是专门用来传送 8085 和外部输入/输出(I/O)设备之间的信息用的两条引线。在程序软件的控制下，8085 能够把一条引线 SOD 置逻辑 1，或逻辑 0 电平(与 TTL 电平兼容)。8085 还能读出另一条引线 SID 的状态。在串行通信方面，这两条引线是特别有用的。

本书用的数制

在全书中，我们将用八进制数(基数是 8)、十进制数（基数是 10）和十六进制数(基数是 16)。因为用八进制数和十六进制数给 8080 和 8085 微型计算机编程序是最普遍的，所以在程序表和解释栏，我们都用了这两种数制。例如，某一章中有一个典型语句可能是："这条输出指令执行之后，如果A寄存器的内容是 360 (F0)，那么，发光二极管(LED)熄灭，然后……"这里要指出的是：A寄存器存储的是八进制数 360 或与它相等的十六进制数F0。在每个解释栏中，不写"八进制数XXX或十六进制数YY"的字样，而只列出八进制数，则在该数的后面的括号中就是相等的十六进制数。在某些情况下，解释栏使用十六进制数。解释栏偶尔也出现二进制数或十进制数，并在后面的括号内注上相等的八进制数和十六进制数。例如，"…当B寄存器所存储的数是 11001010 (八进制数 312，十六进制数CA)时，转移……"对于这种特殊例子，重要的是应该注意观察这个 8 位字的二进制位的排列。为了读者方便起见，在我们的程序例子中，只用八进制数和十六进制数。

程 序 格 式

8080 和 8085 微处理器能执行的指令的字长，也有 16 位的或 24 位的。由于这两种微型计算机用的存储器的存储单元的宽度只有 8 位，所以，必须分别把字长为 16 位或 24 位的指令存储在两个或三个连续的存储单元。因此，我们在程序表中，一个字节的指令占用一行，两个字节的指令占用两行，三个字节的指令占用三行。例 1-3 包含有一个字节的，两个字节的和三个字节的指令。

例 1-3 程序表的格式

```
ADDB

MVIA
103

JMP
314
023
```

我们先不管这些助记符代表什么，或者该程序起什么作用，只要知道 ADDB 是一条单字节指令，MVIA 是一条双字节指令，JMP 是一条三字节指令就行了。这就是说，MVIA 指令的第二字节就是 103 (43)；314(CC)和 023(13)分别是 JMP 指令的第二和第三个字节。请注意上一个语句中的八进制数及其括号内等效的十六进制数。在有关汇编语言程序设计的其他书中，上面这段程序可以写成如例 1-4 所示的样式。

例 1-4 另一种可能的程序表格式

```
ADDB
MVIA, 103
JMP, 314  023
```

关于怎样把这种程序表中的单字节指令、双字节指令和三字节指令存入存储器的问题，对有些读者来说，可能是个难题。但是，指出这一点是很重要的：无论在纸上如何写出程序，都必须把适当的二进制数存储在连续的存储单元，程序才能够被微型计算机执行。

8080 和 8085 的相同之处

读者在本章的前一部分已经看到，8085 与 8080 很相似。其实，本书介绍的大多数程序和电路既可以用于 8080，又可以用于 8085。我们考虑过用符号“8080/8085”表示这样一种含义：我们所介绍的硬件或软件都可以用于这两种微处理器。可是，这种表现形式不便于阅读。**因此，如果没有说明，本书的程序和接口电路器件都可以用于 8080 和 8085。如果某个程序或接口只能用于 8080 或8085，那么，我们会说明它只能用于哪一种微处理器。**

在对 8080 和 8085 这两种微处理器的简单介绍之后，我们再来研究这两种微处理器能够执行的指令。这样，我们就能够把 8080 或 8085 完成有用任务的指令序列编集到一起。

第二章　80 80/80 85的基本指令

8080 微处理器能执行 244 条不同的指令。我们并不去零零碎碎地讨论每一条指令，而是把功能基本相同的指令归为一类进行讨论，可分成如下五类：

1．数据传送指令

2．输入/输出(I/O)指令

3．逻辑指令与算术运算指令

4．分支指令与判定指令

5．中断指令

读者已经知道，8080 微处理器集成电路芯片内有七个通用寄存器，我们可以利用程序指令对每个寄存器进行存取。通过讨论 8080 微处理器可以执行的指令，我们将能够确定这些寄存器的内容发生什么变化，每条指令如何影响标识位，我们怎样可以用指令测试这些标识位的状态，所以我们可以给 8080 微处理器编程序，以 8080 正在操作的数据为基础做出判定。而且，通过讨论输入/输出(I/O) 和中断指令，我们还能够学会怎样给 8080 微处理器编程序，使 8080 能够与外界通信。

正如读者所知，8080 是真正的 8 位**微处理器**，用来组装成 8 位**微型计算机**。这就是说，数据的传递，即用算术和逻辑指令处理数据，在一般情况下一次处理 8 位。我们在第一章已经提到，8080 微处理器有一条双向数据总线；这就是说，数据可以朝两个方向流动，（一次朝一个方向流动）。8080 微处理器利用这条数据总线与存储器和输入/输出设备进行通信。

数据传送指令

最容易理解的最简单的指令也许是 8080 的数据传送指令。这些指令用来在内部通用寄存器（A、B、C、D、E、H 和 L）之间传送数据，也可以用来在这些寄存器和存储器之间传送数据。假设我们已经把一些数据存储在某些寄存器中，我们可以容易地给 8080 微处理器编程序，用一条传送指令，把这些数据传送到其他任何一个通用寄存器。这条传送指令的助记符是 MOV。这是一条单字节指令，存储这条指令只需要一个存储单元。我们怎样知道哪个寄存器要接收数据，哪个寄存器要传送数据呢？这条 MOV 指令必须指定**源寄存器**，源寄存器存储了要传送的数据；这条 MOV 指令还必须指定**目的寄存器**，数据要被传送到目的地。例如，为了把存储在 E 寄存器的 8 位数据字移到 B 寄存器，我们应该使用 MOV BE 这条指令。这条指令**把存储在 E 寄存器的 8 位数据字送到 B 寄存器**。为了执行这条指令，必须把 MOV BE 的二进制数的适当操作码（即 OP）存储在 8080 微型计算机的存贮器里。在这个 8 位操作码中，必须有一个方法来区别源寄存器和目的寄存器，还必须有一个方法来区别各个通用寄存器。因此，英特尔（Intel）公司（8080 的研制公司）给每个寄存器已经分配了一个二进制代码，如表 2-1 所示。

读者可以看出，表 2-1 并没有列出二进制操作码 110（八进制数 6）。这个操作码只有特殊用途，我们将要简单地予以讨论。既然我们能把寄存器区别开来，那么，我们还一定有一种方法，规定传送指令的类型。为此，读者需要记住一条很简

表 2-1　　分配给每个 8 位寄存器的二进制操作码

寄 存 器	分配的二进制操作码	等效的八进制操作码
A	111	7
B	000	0
C	001	1
D	010	2
E	011	3
H	100	4
L	101	5

单的规则：**寄存器一寄存器传送指令都是从八进制数1开始的。**这就是说，我们正在讨论的传送指令的八进制操作码都在 100 和 177 之间，（十六进制数 40 到 7 F）。为了指定目的寄存器和源寄存器，读者需要记住 MOVDS＝1 DS，D 代表指定目的寄存器用的八进制代码，S 代表指定源寄存器用的八进制代码。因此，这条 MOVBE 指令的八进制数操作码是 103（十六进制数 43）。当该操作码被存储在 8080 微型计算机的存储器，然后被执行时，E 寄存器的内容被送到 B 寄存器里。

数据可以从任何一个寄存器送到另一个寄存器。表 2-2 列出了几条 MOV 指令和这些指令的相应的八进制的及十六进制的操作码。当 8080 执行任何一条传送指令之后，把这些数据从源寄存器复写到目的寄存器。**当一条传送指令被执行之后，源寄存器的内容仍然保持不变。**

假设读者在一个程序中已经使用了一条 MOVLD 指令。这条指令起什么作用呢？它的作用是把 D 寄存器的内容移到 L 寄存器。这条指令的操作码被存储在存储器中，它实际上是二进制数 01101010，但是可以用八进制数表示为 152，或者用十六进制数表示为 6 A。可是，我们假设，为了使这个程序适当地操

表 2-2 几条 MOV 指令及其八进制和十六进制操作码

助 记 符	八进制数	十六进制数	功 能
MOVCB	110	48	把B的内容送到C
MOVLD	152	6 A	把D的内容送到L
MOVDL	125	55	把L的内容送到D
MOVEE	133	5 B	把E的内容送到E
MOVAH	174	7 C	把H的内容送到A

作，而需要用一条 MOVDL 指令取代 MOVLD 指令。为了改变这个程序，我们只需要用八进制数 125 或十六进制数 55 代替存储在存储器的前一条指令(MOVLD)。注意，这两条指令的唯一差别是目的寄存器的操作码和源寄存器的操作码被对换了。读者可以明白，当用八进制数 152 和 125，而不用十六进制数 6 A和 55 来表示这两条指令时，更换寄存器的操作码是特别容易的。因此，许多程序设计员给 8080 微型计算机编程序时比较喜欢使用八进制。

读者还应该注意表 2-2 中的这条 MOVEE 指令。这条指令被执行后，其结果将是，E 寄存器的内容保留不变。这是 8080 微处理器的一条有用指令，但是它的作用根本看不见，因此，可以把它叫做“不做事”指令。其他六个通用寄存器也有这样的指令：MOVAA、MOVBB、MOVCC、MOVDD、MOVHH、MOVLL。

8080 微处理器可以在这些通用寄存器里存储多少位信息呢?8080微处理器有七个通用寄存器，因此,在这些寄存器里可以存储 56 位信息，或七个 8 位字。如果我们需要存储更多的信息位，则会发生什么问题呢？为了解决这个问题，也可以把

数值数据存入存储器，或从存储器中取出。

用读/写存储器存储数据

在8080微处理器的任何一个通用寄存器和存储器之间传送数据，比在8080的通用寄存器之间传送数据要困难些。读者可以猜猜，这是什么原因呢？在8080微处理器的内部寄存器之间传送一个数据字时，必须指定该数据字的源寄存器和目的寄存器。在存储器和 8080 微处理器的内部寄存器之间传送一个数据字时，也必须指定该数据字的源和目的寄存器。但是，从存储器传送数据字，就有 65536 个可能的存储单元供选择。这就是说，读者必须指定 8080 微处理器具体与哪个存储器单元传送数 据。

出人意料的是，单字节指令可以用于 8080 的内部寄存器和存储器的任何一个存储单元之间传送一个数据字。这是怎么实现的呢？ 8080 在执行这条单字节指令之前， 必须把一个 16 位存储器地址存储在H寄存器和L 寄存器。H寄存器用来保存高 8 位字节，即该地址的 8 位最高有效位，L 寄存器用来保存该地址的低 8 位字节，即该地址的 8 位最低有效位。一旦这两个地址字节被装入H和L 寄存器之后，就可以用一条单字节指令来将一个 8 位数据字节在H及 L 寄存器所指定的存储器单元和 8080 微处理器的 A、B、C、D 或E 寄存器之间传送。8080 微处理器有若干条传送数据的单字节指令。这些单字节指令的格式都与寄存器的数据传送指令的格式相同；即：MOVDS＝1 DS。其中，目的寄存器和源寄存器使用八进制操作码。当一个外部存储器单元由H寄存器和L 寄存器的内容寻址时，把八进制代码 6 分配给它。当我们希望指定由H和L 寄存器寻址的一个存储器

单元或者作为数据的目的存储器单元或者作为数据的源存储器单元时，必须使用 6 这个八进制操作码。

例如，为了把存储器单元 030 123(1853)的内容传送到D寄存器，可以使用一条 MOVDM 指令。但是，在这条指令被执行之前，必须把一个 16 位地址装入H和L寄存器；必须把这个 16 位地址的高 8 位，即八进制数 030(18)，装入H寄存器；把地址的低 8 位，即八进制数 123(53)，装入L寄存器，然后才能执行这条 MOVDM 指令。这些传送指令使用H寄存器和L寄存器寻址的一个存储器单元作为数据的源寄存器或数据的目标寄存器，所有这些传送指令，读者能够想得到吗？我们在表 2-3 归纳了这些指令。

表 2-3　　存储器访问指令 MOV

存储器作为源寄存器		存储器作为目的寄存器
MOVAM		MOVMA
MOVBM		MOVMB
MOVCM		MOVMC
MOVDM		MOVMD
MOVEM		MOVME
MOVHM*		MOVMH $
MOVLM*		MOVML $
	MOVMM	

这条 MOVMM 指令（八进制数 166，十六进制数 76）不象其他一些传送指令，例如 MOVEE，和 MOVBB；它不是一条“不做事”的指令。相反，它是一条供 8080 微处理器暂停的指令，即 HLT。如果 8080 执行这条指令时，它就停止执行程序。我们已经在 MOVHM 和 MOVLM 指令旁边加上了星号。请读者想想，为什么要给这两条指令加上星号标志呢？因为，

H寄存器或L寄存器的8位地址字节的改变取决于执行这些指令的哪一条。如果8080执行MOVHM指令，则必须把H和L寄存器寻址的存储器单元的内容装入H寄存器，H寄存器原来存储的内容是这个存储器单元地址的高8位。如果8080执行MOVLM指令，则必须把H和L寄存器寻址的存储器单元的内容装入L寄存器。

如果读者使用MOVMH和MOVML指令时不够小心，还会带来某些问题。这两条指令把H寄存器或L寄存器的地址存储在H和L寄存器所寻址的存储器单元。如果H寄存器存储的内容是001(01)，L寄存器存储的内容是100(40)，那么，这条MOVMH指令则把一个八进制数001（01）存入第001100(0140)号存储器单元，如果，H和L寄存器所存储的存储器地址是001 100(0140)，这条MOVML指令被执行时，将把100(40)存入第001 100(0140)号存储器单元。

8080微处理器可以执行若干条指令，来把数据存入或从存储器取出。我们刚才讨论了**存储器访问指令**，它们的格式是MOVDS。8080微处理器执行这些指令，就把8位数据字在某个寄存器和一个唯一被寻址的存储器单元之间传送。其他存储器访问指令将在另一章讨论。

我们还没有讨论过的许多问题之一，包括把数据值装入寄存器，或把地址值装入H和L寄存器。怎样解决这个问题呢？数据存储在寄存器时，我们已经知道怎样传送，但是，我们开始怎样把数据值装入寄存器呢？这个问题将在下一节讨论。

立即数传送指令

首先，人们怎样把数据值装入 8080 微处理器的任一个通用寄存器呢？把数据值装入某个寄存器，其方法之一是执行一条只有一个数据字节的指令。仅仅一条指令既包含指令操作码，又包含一个数据字节，这怎么可能呢？对于 8080 微处理器来说，这个问题非常容易，因为在 8080 微处理器的指令系统中有两个字节的指令。存储一条双字节指令需要两个连续的存储器单元，第一个存储器单元用来存储这条指令的八位操作码，第二个存储器单元用来存储**立即数字节**。立即数传送指令实际上都是双字节指令。这些指令用来把指令的第二个字节，即数据字节装入指定的寄存器。8080 微处理器执行一条立即数传送指令，就能把一个 8 位数据值装入它的任何一个通用寄存器，立即数传送指令的格式如下：MVID〈B_2〉。MVID 指令中的D代表这条指令的数据字节的目的寄存器。读者可以用我们已经讨论过的A，B，C，D，E，H或L寄存器来代替D。〈B_2〉这个符号表示，下一个紧接着的存储器单元存储一个 8 位数据字节(字)，将用来装入立即数传送指令所指定的寄存器。

假设读者需要把 246（A6)这个数装入B，C，D和E寄存器，则可用例 2-1 中这段程序来完成。

例 2-1 把同一个数值装入B，C，D和E寄存器

八进制	助记符	十六进制	说明
006	MVIB	06	把立即数据字节
246	〈B_2〉	A 6	送到B寄存器
110	MOVCB	48	把它从B寄存器

			送到C寄存器
120	MOVDB	50	把它从B送到D
130	MOVEB	58	从B移到E

在例 2-1 中，为什么没有使用MVIC、MVID和MVIE这三条指令呢？这是因为：如果用了这三条指令，存储这段程序需要八个存储单元，而例 2-1 所列出的程序不用这三条指令，只需要五个存储单元。如果读者可以编写两种程序，都能完成同样的任务，一种所需要的存储单元较少，那么，你应该使用哪种程序呢？你或许会使用所需要的存储单元最少的程序。关于这个问题，例 2-2 做了说明。

例 2-2 在四个寄存器复制一个数据字节的两种方法

方法 1	方法 2
MVIB	MVIB
$\langle B_2 \rangle$	$\langle B_2 \rangle$
MOVCB	MVIC
MOVDB	$\langle B_2 \rangle$
MOVEB	MVID
	$\langle B_2 \rangle$
	MVIE
	$\langle B_2 \rangle$

对于要执行的每条立即数传送指令而言，必须相应地把哪些操作码存入存储器呢？某些立即数传送指令相应操作码的一些例子如表 2-4 所示。

表 2-4 没有示出立即数传送指令的数据字节。根据这个表，你能确定MVIH，MVIL 和MVIM这三条指令的操作码吗？根据立即数传送指令的操作码通用公式0D6，可以得到

表 2-4　　某些立即数传送指令的操作码

八　进　制	助　记　符	十　六　进　制
006	MVIB	06
016	MVIC	0E
026	MVID	16
036	MVIE	1E
076	MVIA	3E

任何立即数传送指令的操作码。在这个通用操作码表达式中，D表示要装入立即数据字节的寄存器的八进制数代码。只要知道这一点，并参考表 2-1 列出的寄存器的八进制代码，你就能够确定 MVIH，MVIL 和 MVIM 这三条指令的操作码。这些指令的八进制代码和十六进制代码如表 2-5 所示。象其他立即数传送指令一样，这些立即数传送指令的字长都是两个字节。这就是说。无论哪里使用这些指令，必须把指令的操作码存入存储器，跟在操作码后面的必须是将要被装入指定的寄存器或

表 2-5　各条立即数传送指令的八进制和十六进制操作码

八　进　制	助　记　符	十　六　进　制
006	MVIB	06
016	MVIC	0E
026	MVID	16
036	MVIE	1E
046	MVIH	26
056	MVIL	2E
066	MVIM	36
076	MVIA	3E

存储器单元的 8 位数据字节。

这条 MVIM 指令特别令人感兴趣。当执行这条 指 令时，8080 微处理器做些什么呢？这个问题你知道吗？可以用 MVIM 指令来**把一个立即数字节装入 H 和 L 寄存器所寻址的存 储 器单元**。象别的存储器访问指令一样，它用来 在 存储 器和一个 8080 通用寄存器之间传送 8 位字节。在执行这条指令之前，必须把一个地址存入H和L寄存器。

现在，我们可以编写一段程序，把一个地址装入H和L寄存器，然后在存贮器和一个通用寄存器之间传送一个 8 位字节。例如，我们可以把存储器的第 030 123(1853)单元的 内容送到D寄存器。为了完成传送，我们必须先把地 址 030 123(1853)装入H和L寄存器，才能执行 MOVDM 这条 指令。例 2-3 的指令顺序就是用来实际地完成这种数据传送的。

例 2-3 把存储器单元 030 123(1853) 的内容传到D寄存器

八进制	助记符	十六进制
056	MVIL	2E
123	$\langle B_2 \rangle$	53
046	MVIH	26
030	$\langle B_2 \rangle$	18
126	MOVDM	56

在例 2-3 里，8080 执行 MVIL 指令，把一个数据字节123(53)装入 L 寄存器，然后把 030(18)数据字节装入 H 寄存器。这种装入操作一完成，H和L 寄存器寻址的存储器单元的内容就被复制到D寄存器。先装入 L 寄存器，再装入H寄存器，有什么理由可讲吗？没有。首先可以把一个立即数据字节装入H寄存器。需要记住的要点是：我们已经讨论过的任何存储器访

问指令被执行之前，寄存器H和L必须存储了一个完整的16位地址。注意，这并不是说H和L寄存器只能存储存储器的地址。H和L寄存器也是通用寄存器，所以它们也能存储8位数据值。读者务必注意的是：在执行我们已经讨论过的存储器访问指令之前，必须保证H和L寄存器存储一个地址。这两个寄存器存储的是数据值还是地址，8080微处理器是不知道的。但是，正如我们在前面所提到的那样，当H和L寄存器用来指定一个16位的地址时，H寄存器必须存储地址的高位字节，L寄存器必须存储地址的低位字节。

如果要把042(22)这个数据字节写入存储器的第102 341(42E1)号存储器单元，其指令程序是怎样的呢？ 我们可以使用例2-4示出的这段程序。

例 2-4 把一个立即数据字节存入存储器

八进制	助记符	十六进制
046	MVIH	26
102	$\langle B_2 \rangle$	42
056	MVIL	2E
341	$\langle B_2 \rangle$	E1
066	MVIM	36
042	$\langle B_2 \rangle$	22

在这段程序中，在L寄存器被装入之前，要把一个立即数据字节装入H寄存器。请记住，即使这两个寄存器装入的是两个立即数字节，这两个数据字节也将是8080微处理器执行存贮器访问指令时所用的地址。H和L寄存器被装入地址之后，8080微处理器在执行MVIM指令时实际上才把042(22)这个数据字节传送给存储器。

简单的寄存器对指令

读者已经知道，在执行诸如MOVBM一类存储器访问指令之前，必须把一个16位地址存储在H和L寄存器。因此，人们常常把H和L这两个寄存器叫做寄存器对H。利用或改变寄存器对H的内容的指令，总是把H和L寄存器的内容作为一个16位的数据值来对待。

在前面的例子中，用MVIL和MVIH这两条双字节指令把8位数据值分别装入L和H寄存器。还有一条三字节指令，可以用来将16位的数据(或地址)装入寄存器对H(H和L寄存器)。这条指令的形式是：LXIH〈B_2〉〈B_3〉；它的操作码是041(21)。

在例2-5中，8080执行LXIH指令，将005 341(05 E 1)装入寄存器对H。读者可以看到这条指令所完成的任务与执行MVIL和MVIH，或MVIH和MVIL所完成的任务相同。使用LXIH指令的优点是：存储这条指令只需要三个存储器单元，而等效的两条MVI指令，即MVIH和MVIL需要四个储贮器单元。

例 2-5 LXIH指令及其等效的MVI指令

MVIL	LXIH	MVIH
341	341	005
MVIH	005	MVIL
005		341

8080微处理器执行LXIH指令时，把哪个数据字节装入

H寄存器，哪个数据字节装入L寄存器呢？**如果一条指令由三个字节组成，把第二个字节(如例 2-5 的 341 或 E 1)装入 L 寄存器，把第三个字节(例 2-5 的 005 或 05)装入 H 寄存器。**

H和L寄存器是通用寄存器，所以，LXIH 指令可以用来把 16 位数据值或 16 位地址值装入寄存器对H。因为一个完整的 16 位的数可以存入两个 8 位寄存器（寄存器对），所以我们常常使用 MSBY(高 8 位有效字节)和 LSBY(低 8 位有效字节)这两条术语来描述寄存器对H所存储的 8 位字节。

8080 微处理器的通用寄存器可以分组，构成寄存器对的不只是H和L这两个寄存器，B寄存器和C寄存器，D寄存器和E寄存器也都分别可以作为寄存器对使用。这两对寄存器对和 MSBY 与 LSBY 的分配情况，列在表 2-6 中。

表 2-6　　寄存器对及其 MSBY 和 LSBY 的分配

寄存器对B，B寄存器存储 MSBY，C寄存器存储 LSBY。
寄存器对D，D寄存器存储 MSBY，E寄存器存储 LSBY。
寄存器对H，H寄存器存储 MSBY，L寄存器存储 LSBY。

还有一类 LXI 指令，可以用来把 16 位数值装入寄存器对B和D。表 2-7 列出了 8080 的这些附加指令，并给出了它们相应的八进制和十六进制操作码。

这些 LXI 指令都是三个字节长。把这些指令的操作码存入存储器后，必须把这两个数据字节存储到它们后面的两个连续存储器单元，首先存储的是 LSBY，然后是 MSBY。

当执行例 2-6 的程序时，八位寄存器各自分别存储什么内容呢？

表 2-7　寄存器对 B,D 和 H用 LXI 指令的八进制和十六进制操作码

八进制数	助记符	十六进制数
001	LXIB	01
021	LXID	11
041	LXIH	21

例 2-6　使用 LXIB，LXID 和 LXIH 指令

八进制数	助记符	十六进制数
001	LXIB	01
100	$\langle B_2 \rangle$	40
002	$\langle B_3 \rangle$	02
041	LXIH	21
005	$\langle B_2 \rangle$	05
341	$\langle B_3 \rangle$	E1
021	LXID	11
275	$\langle B_2 \rangle$	BD
300	$\langle B_3 \rangle$	CO

8080 执行例 2-6 的这三条三字节的指令以后，8080 微处理器的通用寄存器所存储的数值如表 2-8 所示。

请读者记住，这些三字节指令的第二字节是 LSBY，它到底装入C、E寄存器还是L寄存器应该视执行 LXI 指令的哪一条而定。这条指令的第三个字节，也就是最后一个字节是 MSBY，它被装入 B、D 寄存器还是装入H 寄存器，取决于 LXI 指令的哪一条被执行。要记住的最后一点：这些指令可以用来把一个 16 位地址或数据值装入寄存器对。存储在寄存器

表 2-8　　执行例 2-6 的 LXI 指令后各寄存器的内容

八 进 制 数	寄　存　器	十六进制数
002	B	02
100	C	40
300	D	CO
275	E	BD
341	H	EI
005	L	05

对的数据信息究竟是地址还是数值，8080 微处理器无法区别。只有在执行 MOVMS-和 MOVDM-类指令时，8080 微处理器才假设寄存器对H存储了一个地址。

因为读者不能任意把 8080 微处理器的寄存器配成寄存器对，B寄存器只能与C寄存器配对，D寄存器只能与E寄存器配对，H寄存器只能与L寄存器配对，所以，读者必须仔细地设计你的软件，才能有效地利用寄存器对的指令。例如，你希望把某些操作用的数值装入C寄存器和D寄存器，你就不能用一条 LXI 类指令来把它们装入，因为，没有指令能把C和D寄存器作为寄存器对来装入。相反，必须使用两条两个字节的 MVI 类指令。如果读者已选定使用B、C寄存器或者D和E寄存器，那么可以用一条 LXI 类指令把数据值装入这些寄存器。

当使用两个字节和三个字节的指令时，请读者记住，在编软件(程序)时，应分别留出一个或两个存储单元，以便供添加数据或地址字节用。一条两个字节或三个字节的指令必须分别存储在两个或三个连续的存储单元中。

输入指令和输出指令

为了使数据能在外部设备（通常叫做输入/输出设备）和8080微计算机之间传送，我们可以使用输入（IN）和输出（OUT）指令。如果与8080微计算机连接的有电传打字机、显示终端（CRT）、软磁盘、录音机、数-模或模-数转换器，那么，8080就可以使用输入和输出指令来与这些外部设备通信。这两条两字节指令的八进制和十六进制操作码如表2-9所示。

表 2-9　IN 和 OUT 指令的八进制和十六进制操作码

八进制数	助　记　符	十六进制数
333	IN	DB
027	$\langle B_2 \rangle$	17
323	OUT	D3
171	$\langle B_2 \rangle$	79

当然，8080微型计算机一定有办法，能够从许多台外部设备中指定一台，让数据通过数据总线送给8080微处理器；也有办法从许多台输出设备中指定一台，接收8080通过数据总线输出的数据。因此，输入和输出指令是双字节指令，其中第二个字节是传送或接收数据的外部设备的8位地址。该地址与8080给出的存储器地址在功能上是类似的。但是，这个输入/输出设备地址只能在000～377（十六进制的00～FF）的范围内。这就是说，8080微型计算机连接输入设备和输出设备最多可各达256个，每一台设备都可以分别由8080寻址。IN和OUT

(输入和输出)指令使8位的数据字节在被寻址的输入/输出外部设备和8080的A寄存器之间传送。

为了把数据输出给外部设备(也叫做**输出端口**)，必须首先把该数据存储在8080微处理器的A寄存器。有若干条指令可以用来把一个8位的数据字节装入A寄存器，其中一些如下：

MVIA　　MOVAM　　MOVAE

〈B_2〉

当A寄存器已存有这个数据时，则可以把该数据输出给被选中的输出端口。这段程序如例2-7所示。

例 2-7　把8位数据值传送给第015(0D)号外部设备

八进制	助记符	十六进制
076	MVIA	3E
101	〈B_2〉	41
323	OUT	D3
015	〈ADDR〉	0D

这段程序把数据值101(41)装入A寄存器，然后把A寄存器的数据字传送给第015(0D)号输出端口。执行完这条输出(OUT)指令之后，A寄存器仍旧存储数101(41)。

数据输入象数据输出一样容易，该程序如例2-8所示。

例 2-8　从第103(43)号外部设备接收一个8位数

八进制数	助记符	十六进制数
333	IN	DB
103	〈ADDR〉	43
107	MOVBA	47

在例2-8中，当执行IN(输出)指令时，地址为103(43)

的输入端口的 8 位数据字被装入 A 寄存器。然后 8080 执行 MOVBA 指令，把这个数据字送到 B 寄存器。8080 在执行这条 MOVBA 指令后，A 寄存器和 B 寄存器所存储的输入数据值是相同的。**当使用 IN(输入)指令和 OUT(输出)指令时，数据只能在输入/输出设备和 8080 微处理器的 A 寄存器之间进行输入或输出传送**。可以用来把数据直接输入到其他任何寄存器或存储器的指令是没有的。与输入/输出设备通信的功能更强的技术，即**存储器映象输入/输出**，将在本书的其他章节讨论。

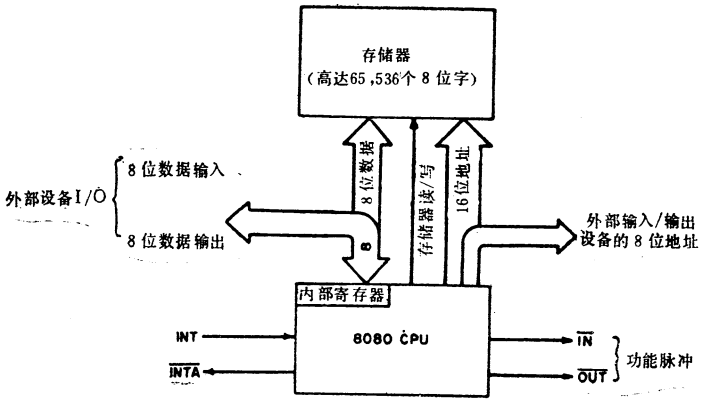
输入/输出设备实际上是怎样接收输入和输出指令中的第二个字节所含有的输入/输出的地址的呢？输入/输出设备怎样知道是送数据给 8080 微型计算机，还是从 8080 微型计算机接收数据呢？当 8080 执行 IN 指令或 OUT 指令时，它把该指令的第二个字节，即输入/输出设备的地址送到 8080 的 16 位地址总线。该设备的地址可以在地址线 A_{15}～A_8 上找到，或者在地址线 A_7～A_0 上找到。这些信号来自 8080 微处理器集成电路的 16 条地址引线。如果读者在 8080 执行 IN 指令或者执行 OUT 指令时，观察存储器地址总线，则会出现下述现象：

A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
0	1	1	0	1	0	0	1	0	1	1	0	1	0	0	1

只通过检查这条地址总线，是根本没有办法确定正在被执行的指令是 IN(输入)指令还是 OUT(输出)指令的。但是，这两条指令的任何一条的第二个字节是用作为外部设备的地址的，这一点是可以确定的。前面出现在总线上的外部设备的地址是什么呢？这个外部设备的地址是 151(69)。请注意，地址 151(69) 既可以出现在地址线 A_{15}～A_8 上，也可以出现在地址线 A_7～A_0 上。这些地址线又叫做高位地址总线（A_{15}～A_8）和低位地址总线(A_7～A_0)。

这两组地址线都可以作为外部设备的地址译码用。外部设备的地址译码是每个外部设备都必须执行的一种操作。外部设备对地址线 A_{15}～A_8 或 A_7～A_0 上的输入/输出设备的地址进行译码；由此确定，它是否与传送给 8080 或从 8080 传送来的数据有关。

当 8080 微型计算机正在执行 IN 指令或 OUT 指令时，输入/输出设备怎样知道它是传送还是接收数据呢？ 8080 微型计算机通过一些外接控制门产生两个“控制”脉冲信号，用来告诉输入/输出设备在 8 位双向数据总线上传送的数据的方向。这两个脉冲，一个叫做 $\overline{\text{I/OR}}$（读输入/输出设备），供输入用，一个叫做 $\overline{\text{I/OW}}$（写输入/输出设备），供输出用。有时，又把这两个脉冲分别简称为 $\overline{\text{IN}}$ 和 $\overline{\text{OUT}}$，I/OR（或 IN）脉冲可以用来将某个选中的输入设备的数据选通到数据总线上，以便在 IN 指令执行时，8080 能把这个数据装入 A 寄存器。$\overline{\text{I/OW}}$（或



微型计算机

图 2-1　典型的 8080 微型计算机系统的方框图

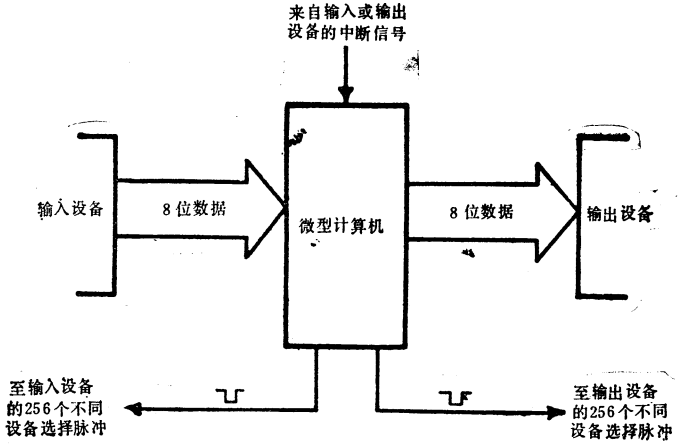


图 2-2　8080微型计算机系统的重要的输入和输出信号

$\overline{\text{OUT}}$)脉冲可以由选中的输出设备用来锁存A寄存器的内容；OUT 指令被执行时，该内容由 8080 送到数据总线上。这些概念可表示成图 2-1 和 2-2。如果需要进一步了解外部设备寻址和外部设备的地址译码情况，请参考本章末列出的参考文献。

逻辑指令和算术运算指令（8 位）

引　言

8080 微处理机能够执行的逻辑指令和算术指令大多数是单字节指令。这就是说，只能指定被操作的一个数据字的源，即一个变量。因为识别每个寄存器和存储器需要三位信息，

实际上，单字节指令的其余5位用来指定要执行的逻辑指令或算术指令的类型。例如，如果要把两个8位数相加，那么，这条加法指令只能告诉8080某个变量存储的地方。那么，8080从哪里获得另一个变量呢？相加的结果又存贮何处呢？

对于把8位数据字进行算术或逻辑运算指令而言，8080微处理器总是用A寄存器的内容作为一个变量。执行逻辑或算术指令的结果总是放回或存储在A寄存器。这就是说，8080可以"把B寄存器的内容加到A寄存器的内容之上，并把相加的结果保留在A寄存器"，或者"从A寄存器的内容减去D寄存器的内容，然后把相减的结果保留在A寄存器。"

我们在前面讨论传送指令时，读者已经看到数据源基本上包括三种：某个通用寄存器的内容，寄存器对H寻址的存储器单元的内容和立即数据字节。这三个数据源也可以用来作为产生8位结果的逻辑或算术指令的一个变量。例如，寄存器对H寻址的存储器单元的内容，或者一个立即数据字节，都可以加到A寄存器的内容之上。相加的结果自动地保留在8080的A寄存器里。因为算术和逻辑指令的一个变量是从A寄存器中获得的，并且这些操作的结果存放在A寄存器，所以，8080的A寄存器又叫做**累加器**。有一点很重要读者必须记住，就是，存储在A寄存器的变量或者数据会被这些操作"丢失"或者破坏。算术或逻辑运算操作完毕之后，A寄存器里仅能找到结果。但是，还有一组不遵循这一规则的逻辑指令。

那么，8080微处理器实际上能执行哪些逻辑指令和算术指令呢？8080能执行的逻辑指令有AND（与），OR（或），exOR（异或）和比较两个8位数值的指令。8080还能执行四种算术功能的指令组求得8位的结果。这四种运算包括加，带进位加，减，和带借位减。8080微处理器没有乘法指令和除

法指令。严格地说，即使 8080 没有逻辑指令或算术指令，8080 也能对任何一个通用寄存器的内容增量或减量，给寄存器对H寻址的存储单元的内容增量或减量，或给寄存器对的内容（一个 16 位的数）增量和减量。8080 微处理器还能把A 寄存器的内容向左或向右移动一位。

8080 的标识位

8080 执行一条逻辑或算术指令之后，可以测试已经完成了的操作的结果。8080 完成这一测试任务是通过测试四个内部状态标识位的状态（逻辑 1 或逻辑 0）来完成的。表 2-10

表 2-10　　四个状态标识位及其含义

1. 进位标识位	该标识位用来表示逻辑运算或算术运算是否出现了进位或借位。
2. 符号标识位	符号标识位用来表示执行了算术或逻辑指令后，其结果的符号是正的还是负的。
3. 奇偶标识位	奇偶标识位用来表示算术和逻辑指令操作的结果的奇偶是偶还是奇。
4. 零标识位	零标识位用来表示逻辑或算术指令执行后的结果是否是零。

表 2-11　　状态标识位的状态和意义

进位标识＝0	没出现进位或借位。
＝1	出现了进位或借位。
零标识位＝0	结果不是零。
＝1	结果是零。
符号标识位＝0	结果的符号为正。
＝1	结果的符号为负。
奇偶标识位＝0	结果的奇偶位数是奇数。
＝1	结果的奇偶位数是偶数。

列出了这些标识位，并对它们的含义作了说明。

在本书给出的大多数程序例中，需要测试进位标识位和零标识位的状态。符合标识位和奇偶标识位并不如其它两个标识位那样常用。符号标识位只有在牵涉到算术运算的结果的符号（如 5 和－ 7 相加）时才被检测；通信时，使用奇偶标识位。

在 8080 微处理器集成电路中，读者可以 把 这些标识位视为单个的触发器。它们都有两种可能的状态，如表 2-11 所示。

逻 辑 指 令

逻辑操作能够使软件指令启动硬件逻辑电路(器件)。在软件的控制下，可以把A寄存器的内容进行“与”，“或”,异“或”，比较和循环移位操作。这类指令的大多数处理两个数据字节。只有循环移位指令只处理一个字节。必须把逻辑指令所包含的一个数据字节存储在A寄存器。应用这些指令的某些例子是有用的，读者通过这些例子，能够看清如何应用这些指令。

当希望在一个 8 位的字节内，具体给某些位清除，过滤，或屏蔽时，“与” 操作是特别有用的。例如，美国信息标 准 交换码 （ASCII） 中，0～9 这十个字符分别被指定了 一 定 的数值，如表 2-12 所示。假设最高有效位 （MSB） 或校 验 位总是逻辑 1 。

如果 ASCII 值的四位高有效位被屏蔽掉，使它 们 的 值为零，那么，每个字符的二-十进制 （BCD） 值仍旧是四位低有效位。

5_{10}＝ASCII 5＝10110101_2，00000101_2＝5(BCD)

字符“5”用 ASCII 值表示为 265 （B 5）。怎样把四 位 高有

表 2-12　　0～9十个字符的 ASCII 代码值

字　符	ASCII		二 进 制 数	二-十进制数
	八 进 制 数	十六进制数		
0	260	B0	10110000	0000
1	261	B1	10110001	0001
2	262	B2	10110010	0010
3	263	B3	10110011	0011
4	264	B4	10110100	0100
5	265	B5	10110101	0101
6	266	B6	10110110	0110
7	267	B7	10110111	0111
8	270	B8	10111000	1000
9	271	B9	10111001	1001

效位置为零（即屏蔽了）呢？两位二进制位的逻辑“与”操作的真值表如表 2-13 所示。

读者可以知道，我们可以把这两位二进制位假设成两个不同的状态；因此，逻辑 1 与逻辑 0 相“与”，可能有四种不同的组合。请注意只有当A是逻辑 1，B也是逻辑 1，两者相“与”后其结果才是逻辑 1。也可以给A栏加上“Mask”（屏蔽）标志，B栏加上“Data”(数据)标志。

表 2-13　　两位二进制位A和B的逻辑与操作

A	B	结　　果
0	0	0
0	1	0
1	0	0
1	1	1

表 2-14　　屏蔽位和数据位的逻辑“与”操作

屏　蔽	数　据	结　果
0	0	0
0	1	0
1	0	0
1	1	1

到现在为止，我们只讨论过了 8080 微处理器操作 8 位数据字（字节）的指令。“与”指令也可以归纳为这一类。因此，表 2-14 的“屏蔽”只代表一个正在与A寄存器的那个 8 位字的相应位进行“与操作的 8 位字内的 1 位。这就是说，如果一个字节与A寄存器的这个字节进行“与”操作，则 A 寄存器的 D_0 位与另一个字节的 D_0位进行“与”操作。然后，把这一“与”逻辑操作的结果保留在 A 寄存器的 D_0。每个字节的 D_1 位用同样的方式进行“与”操作，然后把操作结果保留在 A 寄存器的 D_1。这种逻辑“与”操作如表 2-15 所示。

表 2-15　　两个8位字的相应位的逻辑“与”操作

数据字节		A 的内容	A 的结果
D_0	与	D_0	D_0
D_1	与	D_1	D_1
D_2	与	D_2	D_2
D_3	与	D_3	D_3
D_4	与	D_4	D_4
D_5	与	D_5	D_5
D_6	与	D_6	D_6
D_7	与	D_7	D_7

把两个 8 位字节的相应位同时进行“与”操作，然后把八位字节的操作结果保留在 A 寄存器。因此，人们把 8080 微处理器叫做并行微处理器，它与串行微处理器相对应而存在，串行微处理器一次只能处理 1 位数据，依次进行处理。

程序例 2-9 和 2-10 所示的“与”操作被 8080 执行之后，A 寄存器的低四位就是 ASCII 键盘上的被按下的那只键的二-十进制数。我们已经假设，只按下键盘上的数字键 0～9 十个键。A 寄存器的其余四位可以用来存储另外四位二-十进制数。但是，为了把第二个二-十进制数字存放在 A 寄存器的其余四位上，则需要增加程序指令。实际上，微型计算机的存储器容量的 50％用来存储二-十进制数，如果其余四位空间不使用的话，存储器的 50％的空间则被浪费了。把一个字分成两个或两个以上的信息块，这种技术通常叫做**压缩**。

至于讨论中提到的信息分块技术，并不需要我们知道，ASCII 键盘怎样与 8080 微型计算机的接口连接，即 8080 微型计算机具体怎样从键盘输入数据。为了对某个软件问题的逻辑指令的应用加以说明，我们只不过把 ASCII 键盘作为一个例子来使用罢了。

8080 微型计算机可以用来执行压缩操作的逻辑指令另外还有两条，这样 8080 微型计算机的存储器（存贮空间）可以得到有效的利用。为了把高四位有效位（不需要的信息位）去掉，用屏蔽数据值 01 7（0F）把 ASCII 字符屏蔽之后，必须把低四位包含的二-十进制数移到该 8 位字的高四位上。8080 为了把这些位移入高四位，它要使用循环移位指令。8080微处理器可以执行的循环移位指令有四条，这四条指令对于A 寄存器的内容所起的作用如图 2-3 所示。

请读者注意，有一位**进位位**与A寄存器（累加器）联系。

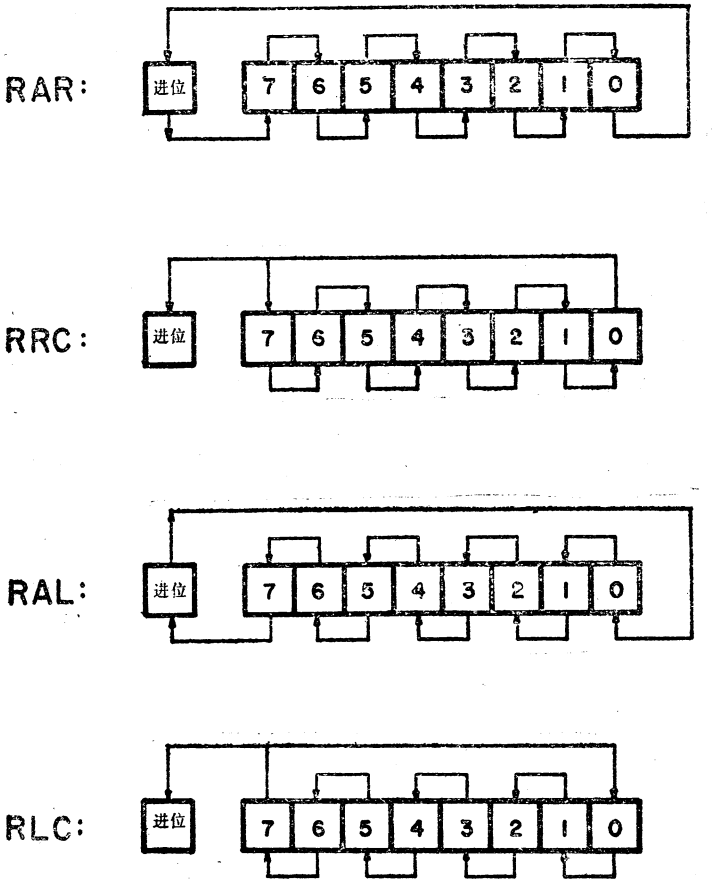


图 2-3 循环移位指令对A寄存器的内容和进位的影响

进位标识位表示该进位位的状态。A寄存器是唯一的能使用这些指令将寄存器的内容循环移位的寄存器。其它寄存器的内容只能送到A寄存器，然后使A寄存器的内容循环移位，然后送回到原来的寄存器，从而完成了循环移位操作。这种循环移位

指令在我们讨论 8080 微处理器能够执行的判定操作时还会提到。

请注意区别 RAR(A通过进位循环右移)和 RAL (A通过进位循环左移)这两条指令，区别 RRC(循环右移)和 RLC (循环左移) 这两条指令。这条 RAR 指令使A寄存器的最低有效位循环右移，进入进位，然后把进位循环右移，进入最高有效位，其他相应各位依此类推。RAL 指令以 RAR 指令同样的方式操作，不同的只是方向相反而已；例如，D_7 循环左移进入进位位，进位位移入 D_0，其余各位依此类推。当执行其他两条循环移位指令，即 RRC 和 RLC 时，将 D_7 位或 D_0 位的内容复制到进位位。(循环移位成为进位位)。RRC 指令使 D_0 位移位，进入 D_7 位和进位位。这条 RRC 指令执行之后，D_7 位和进位位会是相同的。RLC 指令使 D_7 位的内容循环移位，进入 D_0 位和进位位。RLC 指令被执行之后，D_0 位和进位位将是相同的。

正如我们前面所讨论的那样，我们可以把两个二-十进制数循环移位，并且结合起来，把它们压缩成为一个字节。假设已经给 8080 输入了一个 ASCII 字符，并且把这个字符与 017 (0F) 进行了“与”操作；8080 通过连续执行四条 RLC 指令，就把该“与”操作的结果送到A寄存器的高四位有效位。然后，把这个循环移位了的二-十进制数存储在B寄存器里。假设再输入第二个 ASCII 字符，并且将它和 017(0F)进行“与”操作。为了把A寄存器的低四位的数和B寄存器的高 4 位的另一个数结合起来，可以执行逻辑“或”操作。把两位二进制数相“或”的真值表如表 2-16 所示。

为了使我们只把一个 ASCII 字符的低四位有效位存储在A寄存器里，我们可以把屏蔽数 00001111_2(八进制数 017，十

表 2-16　　逻辑“或”操作

A	B	结　果
0	0	0
0	1	1
1	0	1
1	1	1

六进制数 OF)与每个 ASCII 字符进行“与”操作。如果这样做了，则在“与”操作的结果中，就只包含低四位有效位，高四位有效位全部为零。

例 2-9　屏蔽一个 ASCII 字符的高四位有效位

助记符	A寄存器的内容
IN	
〈ADDR〉	10110101
ANI	
$\langle B_2 \rangle$	00000101

在程序例 2-9 中，我们使用了一条双字节“与”立即数指令 ANI。象所有其他立即数指令一样，ANI 这条指令的第二个字节是立即数据字节。读者还能否想得到可以完成同样的功能，而指令序列有所不同的指令吗？我们的回答是：办法众多，其一见程序例 2-10。

例 2-10　用一条 ANDC 指令屏蔽四位高有效位

助记符
MVIC
$\langle B_2 \rangle$
IN

〈ADDR〉
ANDC

在程序例 2-10 中，至于为什么用C寄存器来储存屏蔽位，这并没有什么特别的理由。无论使用哪个寄存器，立即数传送指令的立即数据字节的值必须是什么呢？立即数据字节的值必须是屏蔽数的值，即 017 (0F)。

8080 执行这条 IN 指令之后，把A寄存器的内容与C寄存器的内容进行“与”操作，然后把这一操作的结果保存在A寄存器里。读者已经知道，当使用的屏蔽数为 00001111_2 时，ANDC 和 ANI 这两条指令可以用来屏蔽四位高有效位。我们也可以使用 ANDM 指令。但是，在正确地执行这条指令之前，必须做些什么呢？在执行 ANDM 指令之前，必须把一个 16 位存储器地址装入寄存器对 H(H 和 L 两个寄存器)。这一操作一完毕，就可以执行 ANDM 指令。读者必须记住，实际上还要把这个屏蔽标志存储在参考存贮器单元。

读者已经看到，这些逻辑指令需要两个变量。一个变量存储在A寄存器里，另一个变量也可以是一个立即数据字节，即另一个寄存器的内容，或者是寄存器对H的内容所寻址的存储器单元的内容。

8080 执行了逻辑“或”指令后，可以把包含四位高有效位的数据字与包含四位低有效位的数据字结合在一起，任何一组的数据位都不会影响另一组的数据位。它们的结果应该是一个包含两个四位二-十进制数的压缩了的字节，如表 2-17 所示。

输入两个 ASCII 字符，把不需要的位屏蔽，并且把这两个数据字结合在一起所需要的软件指令如例 2-11 所示。

在例 2-11 里，并没有列出各条指令的操作码。为了表示该

表 2-17　用一条“或”指令把两个二-十进制数合并

00000111 01010000	A寄存器的内容与B寄存器的内容进行“或”操作。
01010111	把结果存入A寄存器。

程序的各条指令执行后的结果，我们列出了A寄存器的内容。这些指令被执行之后，将两个8位ASCII字符的二-十进制数合并，压缩成一个数据字节。该结果被保存在A寄存器里，以供后用；也可以把结果存入某个存储器单元，或者另一个寄存器。

例 2-11　输入，屏蔽循环移位以及合并两个ASCII字符

```
A的内容   助记符        说明
          IN
10110101  015          /从第 015 (0D) 号外部设备输入一
                       /个 ASCII 字符。
          ANI
00000101  017          /用 017 (0F) 进行屏蔽，使四
                       /位高有效位置零。
00001010  RLC          /把四位
00010100  RLC          /低有效位
00101000  RLC          /向循环左移四次。
01010000  RLC
01010000  MOVBA        /把这个二一十进制数存入 B
                       /寄存器里。
          IN           /从第 015 (0D) 号外部设备输入
10110111  015          /第二个 ASCII 字符。
          ANI          /用 017(0F)进行屏蔽，
```

```
00000111  017      /把四位高有效位置零。
01010111  ORAB     /把A的内容与B的内容进行
                   /“或”操作。
                   /现在，这两个二～十进制数
                   /已被压缩。
```

在程序例 2-11 里，有一个解释栏。本书后面给出的程序例都以这种方式列出了解释栏，用符号“ 1 ”把解释栏与指令助记符分开。

A寄存器的内容还可以与其他寄存器的内容进行比较，与寄存器对H寻址的存贮器单元的内容进行比较，或者与一个立即数据字节进行比较。如果读者需要 8080 微型计算机测试或等待外部设备输入一个数据，那么，把A寄存器的内容与一个立即数据字进行比较的比较指令是特别有用的。也可以用这条比较指令，使 8080 微型计算机根据一个数据是否小于等于，或者大于另一数据来作出判断。

如果我们希望知道某个数是在上限还是下限，常常也使用比较指令。8080 微处理器使用特定的内部寄存器(这些寄存器，不能直接用来存取)，执行比较指令，从A寄存器的内容减出指定的寄存器的内容，或者减去指定的存储器单元的内容，或者减去一个立即数据字节。减操作的结果只在四个状态标识位的状态上反映出来。正在被比较的下列内容：

通用寄存器的内容，寄存器对H所寻址的存储器单元的内容，或者一个立即数字节和A寄存器的内容都不受比较指令的影响。所进行的减法操作，只影响四位状态标识位。

我们用一条两个字节的立即数比较指令 CPI，把A寄存器的内容与一个立即数据字节进行比较，下面给出了一些这方面

的例子。

假设A寄存器的内容是 127(57)。执行一条

CPI	或	CPI
126_8		56_{16}

之后，**进位标识位**是逻辑 0，**零标识位**是逻辑 0。这表示A寄存器的内容比这个数据字节大。符号标识位和奇偶标识位的状态无关紧要。

8080 执行一条：

CPI	或	CPI
127_8		57_{16}

之后，进位标识位将是逻辑 0，零标识位是逻辑 1。这表示A寄存器的内容等于这个数据字节。符号位和校验标识位的状态是无关紧要的。

8080 执行一条：

CPI	或	CPI
130_8		58_{16}

之后，进位标识位将是逻辑 1，零标识位将是逻辑 0。这表示A寄存器的内容比这个数据字节小。符号标识位和校验标识位是无关紧要的。

这些例子说明，进位标识位和零标识位的状态反映了双字节立即数比较指令 CPI 的执行结果。**CPI 指令被执行之后，A寄存器的内容仍旧是 127(57)**。请注意，在执行算术指令或逻辑指令之前，不必要把状态标识位清零；这正如把数据装入寄存器之前不必要把寄存器清零一样。状态标识位置位或清零，是为了表示刚执行完的算术指令或逻辑指令的结果。标识位置位或清零的状态保持不变，直到执行另一条指令，才能改变它们的状态。

读者已经看到，逻辑指令的字长有一个字节的，也有两个字节的。表 2-18 归纳了 8080 微处理器的逻辑指令。

表 2-18　　8080 的逻辑指令一览表

源寄存器	与	或	异　或	比　较
A	ANAA	ORAA	XRAA	CMPA
B	ANAB	ORAB	XRAB	CMPB
C	ANAC	ORAC	XRAC	CMPC
D	ANAD	ORAD	XRAD	CMPD
E	ANAE	ORAE	XRAE	CMPE
H	ANAH	ORAH	XRAH	CMPH
L	ANAL	ORAL	XRAL	CMPL
存储器	ANAM	ORAM	XRAM	CMPM
立即数指令	ANI	ORI	XRI	CPI
	$\langle B_2 \rangle$	$\langle B_2 \rangle$	$\langle B_2 \rangle$	$\langle B_2 \rangle$
循环移位指令——RLC, RRC, RAL, RAR				

请读者记住，如在诸如 ANAM 指令中，存储器是指寄存器对H所寻址的存储器的地址单元；还应该记住，立即数指令总是双字节指令。虽然我们并没有给出如何应用“异或”指令(如 XRAA,XRAE,即 XRI)的例子，但是，在本书的其它章节，我们将要讨论它们。

算术运算指令

8080 微处理器只能执行加法指令和减法指令。较复杂功能的操作，如乘法和除法运算，8080 也可以做，但是，这些运算必须借助于多条加法指令或多条减法指令才能完成。

这一章将要讨论的算术指令都是对 8 位不带符号的二进制数进行运算的指令。但是，可以进行 2 的补码算术运算。8080 微处理器也能够对二-十进制 数进行运算，也能做 16 位二进制数的加法运算。执行这些操作的指令要在另一章讨论。

对 8 位二进制数进行运算的算术指令，同逻辑指令一样，总是把 A 寄存器的内容作为被加或被减的一个量。状态标识位都会受到这些指令的影响。在下面的程序例中，应用了这些指令，它们是相当简单的。

加法操作

8080 微处理器能够执行两种加法指令：加(ADD)和带进位加(ADC)。用加指令把 A 寄存 器的内容加到 任何一个通常寄存器的内容上，或者加到寄存器对 H 所寻址的存储器单元的内容上，或者加到一个立即数字节上。无论使用哪一条加法指令(ADD)，相加的结果总是保存在 A 寄存器。此外，进位标识位要置位或清零，以表示进位标志 位 的状态。如果出现进位，则进位标识位是“真”，即为逻辑 1。

有 8 条不同的单字节加法指令(ADD)，一条加立 即数指令(ADI)。这条 ADI 指令和其他立 即指令一样，是双字节指令。下面是一些加法指令：

ADDB，ADDM，ADDE，ADDA，ADDH，ADI $\langle B_2 \rangle$

注意，任何一个通用寄存器，包括 A 寄存器都可以用作加数寄存器。和所有的立即指令一样：在立即数加 指令 ADI 的操作码后面，一定有一个数据 字节，它被存储在 ADI 指令的操作码之后的一个连续存储单元。

为了把 B 寄存器的内容加到 A 寄存器的内容上，在程序中

必须有一条单字节指令 ADDB（八进制 200，十六进制 80）。例 2-12 示出了 ADDB 指令的执行结果。

例 2-12 用 ADDB 指令把 B 的内容加到 A 的内容上

寄存器A的内容	10110101
＋寄存器B的内容	＋00101111
寄存器A的内容	11100100

8080 执行这条 ADDB 指令之后，A 寄存器的内容是 $1110\ 0100_2$。由于没有产生进位，所以，ADDB 指令被执行之后，进位标志位是逻辑 0。例 2-13 的程序把两个 8 位数分别装入 A 寄存器和 B 寄存器，然后执行 ADDB 指令，把这两个 8 位数相加。

例 2-13 把 A 寄存器的内容和 B 寄存器的内容相加的程序

```
        /这是一段把B寄存器的内容加到A寄存器的
        /内容之上的程序。相加的结果留在A寄存器。
ADD, MVIA/把一个数值装入A寄存器。
    265   /(八进制数265，十六进制数B5)。
    MVIB /把一个数值装入B寄存器。
    057   /(八进制数057，十六进制数2F)。
    ADDB/把B的内容加到A的内容上，
    •     /结果保留在A。
    •     /程序的其余部分被存储在此处。
```

两个数相加，其结果大于 $255_{10}(11111111_2)$时，出现进位的现象，如例 2-14 所示。

例 2-14 A、B 两个寄存器的内容相加，产生一位进位

寄存器A	11110111_2	247_{10}
+寄存器B	$+00101111_2$	$+\ 47_{10}$
寄存器A	00100110_2	38_{10}
进位=	1	

注意，在例 2-14 中，当把 B 寄存器的内容加到 A 寄存器的内容上时，进位为逻辑 1。为了表示进位是逻辑 1，进位标识位也置成逻辑 1。

读者或许知道，47_{10} 加 247_{10} 并不等于 38_{10}。为了求出正确的结果，你可以把 256_{10} 这个十进制数规定为一位进位。在使用了 ADDB 这条指令的第一个例子（例 2-12)中，并没有进位，因为加的结果没有超过 255_{10}。但是，当把 47_{10} 与 247_{10} 相加时，就产生了一位进位。因此，正确的结果是 256_{10}(进位位)$+38_{10}=294_{10}$。这个数才是你把 47_{10} 与 247_{10} 相加，所希望得到的结果。

例 2-15 所列出的程序是把 B 寄存器的内容加到 A 寄存器的内容上用的程序。相加的结果保留在 A 寄存器，并且，进位是逻辑 1，作为该加法运算的结果。进位标识位将也是逻辑 1，以表示进位的状态。

如果在下一个加法运算过程中，使用了一条带进位加(ADC)指令，那么，进位与存储器单元所存储的一个 8 位数或者通用寄存器的一个 8 位数，或者一个立即数据字节一起都可以加到 A 寄存器的内容上。这一特征使 8080 微型计算机处理的数据字的字长可以大于 8 位。如果要把两个 16 位数相加，先用一条 ADD 指令，把两个 LSBY（最低有效位字节）相加。把相加的结果保留后，然后用一条 ADC 指令把两个 MSBY(最高有效位字节)相加。这条 ADC 指令把两个 LSBY 相加

所得到的进位加到两个 MSBY（最高有效字节）相加的结果上去。

例 2-15 两个数相加，进位置逻辑 1 的程序

```
/该程序是把两个数相加，产生一个 8 位
/结果和 1 位进位的程序。
ADD1，MVIA/把一个数值装入 A 寄存器。
        367     /(八进制数 367，十六进制数 F7)。
        MVIB    /把一个数值装入 B 寄存器。
        057     /(八进制数 057，十六进制数 2F)。
        ADDB    /把 B 的内容加到 A 的内容上，结果留在 A。
        •       /程序的其余部分存储
        •       /在此处。
```

八位二进制数可以表示 0～255 的任何一个十进制数：

128	64	32	16	8	4	2	1	十进制数的权
0	0	0	0	0	0	0	0	=0 最小的字
1	1	1	1	1	1	1	1	=255最大的字

显然，解决许多问题所需要的数大于 255_{10}。因此，常常使用两个、三个和四个字节的数据字（16 位、24 位和 32 位）。

32,768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1	十进制数的权
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	=0 最小的字
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	=65,532 最大的字

字长为 16 位的字所表示的最小的数据字仍然还是 0，字长为 16 位的字所表示的最大的数据字是 65,536。使用的字节越多，能处理的数则越大。如果前例（例 2-15）相加的两个数

仅仅是两个 16 位数的两个低位有效字节，那么，这两 MSBY (高位有效字节)也能用带进位 加指令 (例 2-16) 实现 加法操作。

读者知道，两个 LSBY 相加所产生的进位一定要加到两个 MSBY 相加的结果上。带进位加(ADC)指令使 8080 微处理器能完成这一任务。例 2-16 的程序用来把两个 LSBY 相加 的进位加到 MSBY 上。现在，进位的状态反映了 MSBY 与 前面

例 2-16 两个 16 位数相加

```
最高有效位字节(MSBY)      最低有效位字节(LSBY)
  10101010(D寄存器的)       11110111(E 寄存器的)
+01000100(B寄存器的)      +  00101111(C寄存器的)
  11101110                1  00100110(保留在L寄存器)
+         1←←←←←进位
  11101111(保留在H寄存器)
```

的进位相加的结果。我们可以用带进位加 (ADC) 指令来把两个二进制数加在一起，而这两个二进制数的字节数几乎可以是任意的。把寄存器对 B 的 16 位数加到寄存器对 D 的 16 位数上的程序如程序例 2-17 所示。作为相加结果的 16 位数保留在寄存器对 H 里。请读者注意，这条 ADCB (带进位加到寄存器 B) 指令的应用。

我们编写这些算术程序的目的在于 说明 如何 使 用算术指令。在我们的程序中，没有包含保留两个 MSBY (最高有效字节) 相加所产生的最后进位的指令。如果使用 ADD 16 这个程序，来把 377 377(FFFF)加到 377 376(FFFE)上，那么，存储在寄存器对 H 的结果应该是 377 376 (FFFE)。这个进位将置成逻辑 1。但是，在 ADD 16 这个程序中，并没有存储最后

的进位位的指令。因此，16 位的结果 377 376(FFFE) 是不正确的。如果使用这些例子的程序来做加法运算，你必须保证任何两个数相加的和不大于 377 377(FFFF)。

例 2-17 把 ADC 类指令用于 16 位加法运算

```
/该程序把寄存器对 B 的 16 位数加
/到寄存器对 D 的 16 位数上。
/其结果保留在寄存器对 H 里。

ADD16,LXIB    /把一个 16 位数
        057   /装入寄存器对 B
        104   /(八进制数 104 057＝十六进制数 442F)。
        LXID  /把另一个 16 位数装入寄存器
        367   /对 D。(八进制数 252 367＝
        252   /十六进制数 AAF7)。
        MOVAE /把寄存器对 D 的 LSBY 移到 A。
        ADDC  /把寄存器对 B 的 LSBY 加到 A。
        MOVLA /把相加的结果保存在 L。
        MOVAD /把寄存器对 D 的 MSBY 加到 A。
        ADCB  /把寄存器对 B 的 MSBY 实行带进位加。
        MOVHA /把 MSBY 相加的结果保留在 H。
        .     /现在执行
        .     /程序 ADD 16 的其余指令
```

如果相加的两个数大于 8 位，那么，必须总是最先把 LSBY 相加，然后再加有效位较高的 LSBY，按此顺序进行，最后把 MSBY 相加；记住这一原则是十分重要的。

减法操作

减法操作指令有两类。它们是减(SUB) 和带借位减(SBB)。

象加法操作指令一样，可以从A寄存器的内容减去寄存器对H寻址的存储器单元的内容，或者减去一个立即数据字节，或者减去其通用寄存器的内容。然后把减法操作的结果保留在A寄存器里。

例 2-18 从A寄存器的内容减去E寄存器的内容

寄存器A	00110111
−寄存器 E	−00010100
寄存器A	00100011

从B寄存器的内容减去E寄存器的内容的程序如例2-19所示。把B寄存器的内容送到A寄存器之后，才能执行减法操作。

如果执行从较小的数减去较大的数的减法操作，则进位会置逻辑1，这就表示出现了借位。从较小的数（存储在A寄存器）减去较大的数（存储在B寄存器)的减法操作如例2-20所示。

例 2-19 B寄存器的内容减去E寄存器的内容

```
/这个程序是用来从B寄存器的内容减去E寄
/存器的内容，然后把结果
/保留在D寄存器的程序。
SUB1,   MVIB    /把被减数(八进制数 067
        067     /=十六进制数 37)装入B寄存器。
        MVIE    /把减数(八进制数 024
        024     /=十六进制数 14)装入E寄存器。
        MOVAB   /把被减数送到A寄存器。
        SUBE    /减去减数，
        MOVDA   /然后，把结果保留在D寄存器。
        .       /存储在此处的是
```

```
        •          /该程序的其余指令。
```

在例 2-21 所列出的程序中，有一条用来把立即数据字节装入A和B寄存器的指令助记符。然后，从A寄存器的内容减去B寄存器的内容。

例 2-20 做减法操作时，产生一位借位

```
  寄存器A          01010110
－寄存器B        －10001011
  寄存器A          11001011
    进位＝              1
```

例 2-21 从较小的数减去较大的数

```
/该程序是用来从A寄存器的内容减去B寄存
/器的内容，然后把结果保留在L寄存器的程
/序。B寄存器的数大于A寄存器的数。

SUB 2,  MVIA    /把被减数装入A寄存器
        126     /(八进制数 126＝十六进制数 56)。
        MVIB    /把减数装入B寄存器里。
        213     /(八进制数 213＝十六进制数 8B)。
        SUBB    /A寄存器减去B寄存器的内容。
        MOVLA   /结果保留在L寄存器。
        •       /然后执行该程序
        •       /的其余的指令。
```

进位位是逻辑 1，这就使我们能够实现减去两个很大的数的减运算——它只要我们把这两个数分成字节。在两个 16 位数进行减法运算时，首先将 LSBY 相减。在两个 16 位数进行

加运算时，首先将两个 LSBY(最低有效字节)相加。

在完成减法操作后，如果进位位置逻辑 1，则确实出现了借位。在这种情况下，必须从两个 MSBY(最高有效字节)的减法操作的结果减去进位 1。

例 2-22 两个 16 位数相减

01110101(D寄存器的)	11010000(E寄存器的)
−00101011(B寄存器的)	10111001(C寄存器的)
01001010(留在H寄存器)	00010111(留在L寄存器)

在例 2-22，因为C寄存器的 10111001 比E寄存器的 11010000 小，所以没有出现借位。执行 16 位数的减法操作的程序指令如例 2-23 所示。

在下一个 16 位数减法操作的例子(例 2-24)中，两个 LSBY 相减时，出现了借位。这是因为C寄存器的 11000101 比E寄存器的 01011100 大的缘故。

例 2-23 寄存器对D的内容减去寄存器对B的内容

```
/该程序是用来从寄存器对D的16位数
/减去寄存器对B的16位数。
/结果保留在寄存器对H的程序。
SUB16,  LXID    /把被减数
        320     /(八进制数165 320=十六进制数75DO)
        165     /装入寄存器对D,
        LXIB    /把减数(八进制数053 271
        271     /=十六进制数2 BB 9)
        053     /装入寄存器对B。
        MOVAE   /把被减数的LSBY送到A寄存器。
        SUBC    /从A减去减数的LSBY。
        MOVLA   /把LSBY结果保留在L寄存器。
```

```
MOVAD   /把被减数的 MSBY 送到A寄存器。
SBBB    /带借位减去减数的 MSBY。
MOVHA   /把结果的 MSBY 保留在H寄存器。
•       /然后执行该程序的
•       /其余的指令。
```

例 2-24 寄存器对D的内容减去寄存器对B的内容，产生了一位借位

```
    MSBY                LSBY
  01101111(D寄存器)   01011100(E寄存器)
-00010011(B寄存器)    11000101(C寄存器)
  01011100            1 10010111(保留在E寄存器)
-          1<-<-<-<-借位
01011011(保留在D寄存器)
```

在例 2-24 中，因为从 01011100 减去 11000101，被减数小于减数，所以产生了借位，进位位置成逻辑 1。执行 16 位数的减法操作的程序如例 2-25 所示。

例 2-25 的程序与例 2-23 的程序是相同的，都是用来完成减法操作的。读者应该注意，该程序有效地运行，可能产生借位或者不产生借位。我们已经使该程序具有通用性，因为借位可以出现，也可以不出现。

立即数算术指令也有两个字节的，例如加立即数(ADI)指令，带进位的加立即数(ACI)指令，减立即数(SUI)指令，和带借位减立即数(SBI)指令。表 2-19 简要地列出了这些指令以及其它加操作和减操作的指令。

例 2-25 寄存器对D的内容减去寄存器对B的内容

```
/该程序是用来从寄存器对D的 16 位数减去
```

表 2-19　8080 的 8 位数加操作和减操作指令一览表

源寄存器	加	带进位加	减	带借位减
A	ADDA	ADCA	SUBA	SBBA
B	ADDB	ADCB	SUBB	SBBB
C	ADDC	ADCC	SUBC	SBBC
D	ADDD	ADCD	SUBD	SBBD
E	ADDE	ADCE	SUBE	SBBE
H	ADDH	ADCH	SUBH	SBBH
L	ADDL	ADCL	SUBL	SBBL
存储器单元	ADDM	ADCM	SUBM	SBBM
立即数指令	ADI	ACI	SUI	SBI
	$\langle B_2 \rangle$	$\langle B_2 \rangle$	$\langle B_2 \rangle$	$\langle B_2 \rangle$

```
/寄存器对B的16位数，并
/把其结果保留在寄存器对D的程序。
SUB 16，LXIB      /把减数装入寄存器对B
      305         /(八进制数 023 305＝十六
      023         /进制数 13C5)。
      LXID        /把被减数装入寄存器对D。
      134         /(八进制数 157 134＝十六进
      157         /制数 6F5C)。
      MOVAE       /把被减数的 LSBY 移到A寄存器。
      SUBC        /减去减数的 LSBY。
      MOVEA       /把结果的 LSBY 保留在E寄存器。
      MOVAD       /把被减数的 MSBY 送到A寄存器。
      SBBB        /带借位减去减数的 MSBY，
      MOVDA       /把结果 MSBY 保留在D寄存器。
        .         /然后执行该程序的
        .         /其余指令。
```

加 1 指令和减 1 指令

有两条特别的算术指令：加 1(INR)指令和减 1(DCR) 指令。这两条指令的一条能使 8080 微处理器把 1 加到某个 通用寄存器或寄存器对H寻址的存储器单元的内容上，或从某个通用寄存器的内容，或寄存器对H寻址的存储器单元的内容减去 1。表 2-20 概括了这两种指令。

表 2-20　　加 1 指令和减 1 指令

INRS＝OS 4	寄存器 S 加 1
DCRS＝OS 5	寄存器 S 减 1

读者可以使用任何一个八进制寄存器代码，来供源寄存器 S 用，这些八进制代码，我们在描述 MOV 指令的章节中已经用到过。加 1 指令(INR) 和减 1 指令 (DCR) 并不影响进位标识位，但是，当寄存器加 1 或减 1，使它的内容等于零时，零标识位则置成逻辑 1。

当你需要程序来给事件计数时，这些增量和减量指令是特别有用的。这些事件可能是 8080 微型计算机系统的外部事件，例如许多汽车通过交叉路口，这些事件也可能是 8080 微型 计算机系统内部发生的事件，例如具体执行某个算术运算程序的次数。

加 1 指令(INR)和减 1 指令(DCR)都是用来处理一个数据字节的。为了弄清一条加 1 指令是怎样影响寄存器的内容和零标识位的，我们在例 2-26 假设把 374(FC) 已经装入到 B 寄存器。

例 2-26 用 INRB 指令使 B 寄存器的内容加 1

B寄存器的内容			指令执行后	
二进制数	八进制数	十六进制数	助记符	零标识位的状态
11111100	374	FC	MVIB $\langle B_2 \rangle$	不明
11111101	375	FD	INRB	逻辑 0
11111110	376	FE	INRB	逻辑 0
11111111	377	FF	INRB	逻辑 0
00000000	000	00	INRB	逻辑 1
00000001	001	01	INRB	逻辑 0
00000010	002	02	INRB	逻辑 0

请读者注意， 只有 B 寄存器的内容从 377 (FF) 增加到 000(00)时， 零标志位才置于逻辑 1 状态。 下一条加 1 指令把这种状态清零，这是因为 B 寄存器加 1 后的内容不再是零。

减操作的方法如例 2-27 所示。

注意，只有 E 寄存器的内容已经被减到 000 (00)时，零标识位方是逻辑 1 。其后，下一条减 1 指令(DCRE) 把 E 寄存器的内容减到 377(FF)，并且把零标志位清除为零。

例 2-27 用 DCRE 指令使 E 寄存器的内容减 1

E寄存器的内容			指令执行后	
二进制数	八进制数	十六进制数	助记符	零标识位状态
00000100	004	04	MVIE $\langle B_2 \rangle$	不明
00000011	003	03	DCRE	逻辑 0
00000010	002	02	DCRE	逻辑 0
00000001	001	01	DCRE	逻辑 0
00000000	000	00	DCRE	逻辑 1

11111111	377	FF	DCRE	逻辑 0
11111110	376	FE	DCRE	逻辑 0

寄存器对加 1 指令和减 1 指令

此外，还有一类特殊的加 1 和减 1 指令(INX 和 DCX)，我们用这两种指令来处理寄存器对 B，D和H的内容。8080 执行这些加 1 和减 1 指令，使这些寄存器对的 16 位数据内容加 1 或减 1 。例如，为了将寄存器对H的 16 位内容加 1 ,应该使用一条 INXH 指令。在例 2-28 里，我们假设寄存器对H已经存放了 00001000 11111110 (八进制数 010376 ，十六进制数 08 FE)这个数。

例 2-28 INXH 指令的应用

指令执行后寄存器对H的内容			助记符
二进制数		八进制数	十六进制数
H寄存器	L寄存器		
00001000	11111110	010 376 08FE	INXH
00001000	11111111	010 377 08FE	INXH
00001001	00000000	011 000 0900	INXH
00001001	00000001	011 001 0901	INXH

当 8080 微型计算机的另一个操作需要使用某个存储器单元的内容时，可以用寄存器对H来指明这个存储单元，这个问题我们已经在例 2-3 作了说明。回想那个例子后，假设你需要把第 030 123 (1853)号存储器地址单元的内容送到D寄存器，把第 030 124 (1854)号存贮器单元的内容送到 E 寄存器。首先你似乎可以考虑采用例 2-29 的程序。

当然，这一段程序一定会很有效的；但是，使用我们上面

刚刚介绍了几条指令的一条，可以简化该程序。

在例 2-29 和例 2-30 中，我们采用了一条 LXIH 指令来把存储器的初始地址装入寄存器对 H。在例 2-29 中，把寄存器对 H 寻址的存储器地址单元的内容装入 D 寄存器。然后，把下一个存储器地址装入寄存器对 H，

例 2-29 把存储器的内容装入 D 寄存器和 E 寄存器

```
LXIH    /把一个 16 位的存储器地址
123     /装入寄存器对 H,(八进制数
030     /030 123＝十六进制数 1853 )。
MOVDM   /把这个存储器单元的内容移到 D 寄存器。
LXIH    /把下一个连续存储器单元的 16 位
124     /地址装入寄存器对 H 里
030     /(八进制 030 124 ＝十六进制数 1854 )。
MOVEM   /把这个存储器单元的内容送到 E 寄存器
```

再把这个地址单元的内容装入 E 寄存器，这段程序如例 2-29 所示。

例 2-30 把存储器的内容装入 D 寄存器和 E 寄存器的改进程序

```
LXIH    /把存储器单元 16 位地址装入
123     /寄存器对 H，(八进制数
030     /030 123＝十六进制数 1853 )。
MOVDM   /把该存储器单元的内容送到 D 寄存器。
INXH    /将地址(八进制 030124，十六进制的 1854)加 1 。
MOVEM   /把该存储单元的内容送到 E 寄存器。
```

如第二个例子(例 2-30)所示，取出存储在连续的存储单元的每个数值，我们不必执行第二条 LXIH 指令。把第一个数

装入D寄存器之后，执行这条INXH指令时，寄存器对H的内容加1。现在，寄存器对H的内容指示到下一个连续的存储器单元，这个存储器单元所存储的要送到E寄存器的数据值。8080执行这条INXH指令以后，再把寄存器对H所寻址的存储器单元的内容装入E寄存器里。

8080微处理器执行完了这条MOVEM指令以后，寄存器对H存储的内容是什么呢？寄存器对H存储的内容是030 124(1854)。如果数据值存储在连续的存储器单元中，那么诸如例2-30那样的程序对于向连续的存储器区存储或取数是很有用的。

这条DCXH指令也是很有用的，有了这条指令我们便可以检查或转送存储器单元的内容(从最高地址开始，传送到最低地址为止)。例2-31示出了这条DCXH指令对寄存器对H的内容的影响。

读者在例2-31中可以看到，DCXH指令是用来对寄存器对H的内容进行16位减操作的。

例 2-31 DCXH指令的具体说明

指令执行后，寄存器对H的内容					助记符
二进制数		八进制数		十六进制数	
H寄存器	L寄存器				
00101011	00000011	053	003	2B03	DCXH
00101011	00000010	053	002	2B02	DCXH
00101011	00000001	053	001	2B01	DCXH
00101011	00000000	053	000	2B00	DCXH
00101010	11111111	052	377	2AFF	DCXH
00101010	11111110	052	376	2AFE	DCXH
00101010	11111101	052	375	2AFD	DCXH

我们在前面提到过，可以将寄存器对B，D和H的内容加1和减1。实现这些操作的指令及其八进制和十六进制操作码如表2-21所示。

表 2-21　　寄存器的加1和减1指令

八　进　制	助　记　符	十　六　进　制
003	INXB	03
013	DCXB	0B
023	INXD	13
033	DCXD	1B
043	INXH	23
053	DCXH	2B

表2-21所列出的指令与用来使寄存器的8位内容加"1"或减1的INR和DCR指令不同，它们不影响任何标识位的状态。因此我们不能直接测试"等于零"条件的加1或减1的16位数据值。

到现在为止，寄存器对的基本指令，我们已经向读者介绍完了。还有一些寄存器对指令，将在后面予以讨论。上面我们讨论过的寄存器对是寄存器对B、D和H。影响这些寄存器对的指令包括LXI，INX，和DCH这三条。

暂停指令和空操作指令

在还没有讨论的许多指令中，有两条简单的单字节指令HLT和NOP。这两条指令的操作码如表2-22所示。

HLT是一条停机指令。当执行这条指令时，8080微型计算机停止一切操作。如果程序很短，只需执行一次，则可以把HLT指令放在该程序的末尾。其典型的程序如例2-32所示。

表 2-22　　HLT 和 NOP 指令的操作码

八　进　制	助　记　符	十 六 进 制
166	HLT	76
000	NOP	00

例 2-32　HLT 指令的应用

```
MVIA   /把 001 装入A寄存器里
001    /(八进制数 001 =十六进制数 01 )。
OUT    /把该值送给第 053 号外部
053    /设备(八进制数 053 =十六进制数 2B)。
INRA   /A寄存器的内容加 1 。
OUT    /把该值输出给第 054 号
054    /外部设备(八进制数 054 =十六进制数 2C)。
HLT    /然后停机。
```

这个程序在两个输出端口(外部设备)产生操作;然后,8080 微型计算停机。一条暂停指令(HLT)执行完毕后，并没有简易办法使 8080 微型计算机恢复程序正常操作。为了使 8080 微型计算机重新运行，读者可以采用硬件中断的办法（关于硬件中断问题，我们将在另外的章目中讨论)，读者也可以使 8080 微型计算机复位，然后从程序存储器的第 000 000 (0000)号单元开始执行程序。

当读者在用简单的调试程序或监控程序给 8080 微 型 计算机编程序时，一般都使用空操作指令(NOP)。如果需要这条 NOP 指令的话，通常把它插入到程序段，以便留出空存储器单元，供需要增添指令时用。如果你在调试程序时发现，错误地省略了一些指令，则可以把补充的指令存储到这条 NOP 所

存储的单元。例 2-33 示出了如何用其他指令填充 NOP 指令的存贮器单元。

例 2-33　使用 NOP 指令在程序中留出空单元

```
MVIA MVIA  /把 075 装入A寄存器里
075  075   /(八进制数 075＝十六进制数 3 D)。
ADDB MVIB
NOP  051
NOP  INRC
NOP  NOP
OUT  OUT   /把A寄存器的内容输出给
120  120   /第 120 号外部设备(八进制数 120＝十六进数 50)
INRA INRA
OUT  OUT   /把A寄存器的内容输出给第 121
121  121   /外部设备(八进制数 121＝十六进制数 51)。
HLT  HLT   /然后停机。
```

实际上，这个程序做什么，确实无关紧要。重要的是在程序中留一些 NOP（空操作)指令，从而可以给这个程序添加上 MVIB 051 和 INRC 两条指令，而这条不正确的指令(ADDB)可以被删去。

转移、传送控制和判定指令

在这以前，所给出的程序例都是直线程序，常把它叫做流水线程序。因为执行一条指令或者一串指令以后，并没有根据此执行结果把控制转到程序的另一部分。下面我们要讨论的这类指令就是告诉读者如何把程序执行序列从一个程序的某一部

分转移到另一部分，并且说明为什么它们对你很有用处。

转移指令

转移指令用来把程序执行的控制转到一段新的程序，即现在被执行的程序的另一部分，或许送到某个具体测试程序。转移指令可以看作程序中的一条用来中断程序正常执行序列，并使8080微型计算机在其他某个点上继续执行程序指令。如果你在编制一组循序渐进的指令来装配汽车，其中一条指令可以这样叙述："返回到第5步"。这条返回指令就是转移指令，因为它已经使你把你的注意力(程序执行序列)回到第五条指令。注意，这条指令正好指明你希望转移到的地方。读者已经懂得，为了指定唯一的一个存储器单元，需要一个16位地址。这也符合转移指令的概念，因为8080微型计算机必须确切知道它要转移到什么地方。

8080微型计算机的转移指令都是三字节指令。如例2-34所示，第一个字节是转移指令的操作码，第二个字节是低8位地址，第三个字节是高8位地址。

例 2-34 转移指令的格式

八进制	助记符	十六进制
303	JMP	C3
034	$\langle B_2 \rangle$	1C
005	$\langle B_2 \rangle$	05

这条指令和指定存储器地址单元的其他各条指令的第二个字节都是低位地址或数的8位低有效位，即最低有效字节(LSBY)，第三个字节是16位地址的高8位，或数的8位高有效位，即最高有效字节(MSBY)。如例2-34所示，8080微型

计算机执行这条转移指令(JMP)时，它应该转移到第 005 034 (051C)号存储器地址单元。在这个地址中，应该存储一些另外的程序指令。

你能猜到这条指令是怎样执行的吗？8080 微型计算机从存储器取出转移指令 JMP 的操作码时，它就知道这是一条转移指令。所以，这条转移指令取出之后，把存储在存储器的第二和第三字节直接装入程序计数器 PC。因为程序计数器 PC 是用来指示存储下一条要执行的指令的存储器单元的，所以下一条要执行的指令将会存放在这条转移指令的第二和第三字节所指定的存储器地址单元。

我们明确地规定了 8080 微型计算机要转移到的存储器地址单元。在一般程序的末尾，你或许要使 8080 返回到该程序的起点，重新执行这段程序。如果这个程序的启始地址是 000 000(0000)，那么，你可以使用例 2-35 的转移指令。

例 2-35 返回到程序的起点

八进制数	助记符	十六进制数
303	JMP	C3
000	$<B_2>$	00
000	$<B_3>$	00

8080 微型计算机执行这条转移指令以后，它在存储器的第 000 000(0000)号地址单元找到要执行的下一条指令。

当 8080 微型计算机的程序控制到达程序中的这条转移指令 JMP 时，它总是要执行这条指令的。因此，转移指令 JMP 叫做无条件指令，任何软件或标识位的状态都不能阻止执行这条 JMP 指令。

但是，有许多程序，只有当某些条件得到满足时，8080 才

执行一条 JMP 指令。如例 2-11 所示，ASCII 数字字符被压缩成一个字节，并且已经假设被输入的字符都是数值字符。实际上，有时你可能偶尔输入了一个非数值字符。那么，你必须写成这样的程序，以便忽略这个非数值字符。我们假定：当疑问号(?)(ASCII 字符值为八进制数 277 或十六进制数 BF)被输入时，必须中止二-十进制压缩子程序，然后控制程序转移到该程序的另一部分。这应该是条件转移指令的很好的应用。

条件转移指令

8080 微处理器的条件转移指令允许我们给转移指令附加某些条件，从而只有规定的条件得到满足时，才执行转移指令。以这样方式所要进行的测试的状态是 8080 微处理器的一个内部标志位的状态。这些内部标识位是进位，符号，零，和奇偶这四个状态标志位。我们将要举例重点阐述进位标识位和零标识位的运用。

正如读者所猜测一样，条件转移指令都是三字节指令，其中第二和第三字节规定为 16 位地址。请读者记住，根据执行逻辑指令和算术指令的结果，这些标识位或者被置位或者被清零，不管条件指令是否被执行，标识位的状态并不改变。8080 微处理器的条件转移指令如表 2-23 所示。

请读者注意，四个状态标识位都可以分别用条件转移指令进行测试。而且，条件转移指令可以测试每个标识位的“真”或者“假”状态，这就是说，8080 微处理器能执行的条件转移指令有 8 条。

在前面讨论的 ASCII 码输入压缩程序(例 2-11)中，没有办法从执行的程序中退出来，去继续执行该程序的另一部分。因此，必须添加一些指令，以便当键盘上的某个特定键被按

表 2-23　　8080 的条件转移指令

操　　作	助记符	八进制数	十六进制数
如果指令*执行的结果不等于零，则转移	JNZ	302	C2
如果指令执行的结果等于零，则转移	JZ	312	CA
如果指令的执行结果没有进位(或借位)，则转移	JNC	322	D2
如果指令执行的结果有进位(或借位)则转移	JC	332	DA
如果指令执行的结果有奇校验位，则转移	JPO	342	EZ
如果指令执行的结果有偶校验位，则转移	JPE	352	EA
如果指令执行的结果为正则转移	JP	362	F2
如果指令执行的结果为负，则转移	JM	372	FA

* 算术的或逻辑的

下时，8080 即开始执行程序的另一段指令。我们可以任意选择疑问号(?)作为被用来中止程序输入段的字符。数的符号(#)，惊叹号(!)，或者百分号(%)与疑问号(?)一样，可以容易地使用。假设最高有效位(MSB)，即奇偶校验位是逻辑 1，那么代表疑问号的 ASCII 字符值是 277(BF)。

在 ASCII 字符输入程序中，我们必须把所有的输入字符与数值 277(BF)进行比较。如果通过比较操作，零标识位置逻辑 1，那么，肯定是按下了键盘上的疑问号键，A寄存器的八位字符一定已经产生了。然后，8080 微处理器可以按条件转移到程序的另一部分。例 2-36 包含了这些增添的程序指令。

例 2-36　用疑问号中止输入程序

```
IN    /从第 015(十六进制 OD)输入设备
015   /输入一个 ASCII 字符。
CPI   /把这个字符与 277(ASCII)键盘上
277   /的疑问号这个数据字节比较。
JZ    /如果这个字符是个疑问号，
```

〈LO〉/指定地址的 JZ 指令
〈HI〉/将被执行。
ANI /这个字符不是疑问号，
017 /所以，把该字符与 017(十六进制数 OF)进行比较。
· /程序的其余指令
· /存储在这里。

在例 2-11 的程序的基础上，我们增添了两条新指令，存储这两条指令需要五个存储器地址单元。增添的这两条指令是立即数比较指令(CPI)和零转移指令(JZ)。首先，A寄存器存储了从第 015(0 D)号输入设备输入的数据。然后，比较指令把A寄存器的内容与立即数据字节 277 (BF) 进行比较。于是比较(减)的结果相应地使标志位置位或者清零。

比较指令并不改变A寄存器的内容。所以，如果A寄存器没有存储 277(BF)这个数，则不执行这条 JZ 指令，因为并没有满足这条指令所规定的零转移条件。如果只有零标志位置位，即"真"这就是说，这条 CPI 指令被执行后，A寄存器所装的内容确实是 277(BF)，才能执行这条 JZ 指令。如果不执行这条转移指令(JZ)，那么，8080 会执行存储在下一个连续存储器单元的指令，即 ANI 指令。如果不执行转移指令，则忽视这条转移指令的地址字节。这是因为 8080 微处理器集成电路确切地知道每条指令需要几个(一个，两个还是三个)字节。所以，8080 知道 JZ 指令后的下一条指令的十六位地址。因此，只要你的程序编写得正确，8080 不会试图把两个字节或者三个字节的指令的地址或者数据字节作为操作码来执行。

读者可以用其他方式运用判定指令吗？读者可以测试输入的全部 ASCII 字符，因此，只有 0～9 十个数字字符被"压缩"。除了疑问号以外，其他字符都被忽略。ASCII 字符 0～9 的数

值表示是 260—271(B0—B9)。那么，应该怎么忽略比 260(B0)小或者比 271 大的数呢，有许多方法，可以用来屏蔽其他的 ASCII 字符。然而，这个程序利用数条 CPI 指令，使得它非常简单。

例 2-37 屏蔽其他 ASCII 字符（数字 0～9 和疑问号（?）除外)的程序

```
START,  IN      /从第 015(十六进制 0 D)号外。
        015     /部设备输入一个 ASCII 字符。
        CPI     /这个字符是个疑问号码?
        277
        JZ      /是的，转移到分配给符号地址
        FERMIT/"TERMIT"的地址，
        0       /从而中止输入。
        CPI     /输入的字符不是疑问号(?)，
        260     /则把这个数据和 ASCII 0 比较。
        JC      /如果进位标识位置位(逻辑 1)，则
        START   /260 比键入的 ASCII 字符值大。
        0       /因此，转移到"START"。
        CPI     /现在，判别输入的字符值是否比 271 大。
        272     /把这个字符与 ASCII 字符 9 比较，大 1。
        JNC     /这个字符是 272 或更大吗?
        START   /如果是这样，则返回到程序。
        0       /的起点("START")。
        ANI     /输入的字符是 0～9 这 10 个数字
        017     /的某一个，则把这个字符的
        •       /高四位屏蔽掉。
        •
        •
        ETC
```

在这个程序中，我们用了两个**符号地**址。你知道这两个符号地址是什么吗？一个是 START，另一个是 TERMIT。在例 2-37 中，符号地址 START 用了三次，符号地址 TERMIT 只用了一次，使用符号地址是一种方法，它表示存储器单元所存储的是程序必须引用的指令或数值数据。在这个程序中，我们知道：如果疑问号键被按下，那么 8080 微处理器不应该执行该程序的压缩 ASCII 码的程序段的任何指令。如果没有按下问号键，那么，8080 必须断定某个数字键（0～9）是否已被按下。如果此时按下了一个非数字键，8080 微处理机则必须返回到 IN 指令，从而忽略该键。

如果把这个程序装入存储器，我们必须记住存储这条 IN 指令的存储器单元的地址。然后，把这个地址用作为 JC 和 JNC 指令的几位地址。如果疑问号键被按下，还必须确定 8080 要返回的地址，并把它存储在 JZ 指令之后。如果输入的字符的数值小于 260（B0）或大于 271(B9)，我们通过定义符号地址 START，或者把该地址与这条 IN 指令联系起来，就能容易地记住 8080 应该转移到什么地方。符号地址被定义以后，它就可以与 JC 和 JNC 指令一起使用。实际上，在其它程序例子中，你会看到符号地址可以与所有的三个字节的指令一道使用。

当然，把这个程序装入存储器以后，我们还必须确定 JZ，JC 和 JNC 这三条指令所包含的 16 位地址字节。但是，在转移指令后，使用符号地址比写〈B_2〉和〈B_3〉更为方便。如果我们使用〈B_2〉和〈B_3〉的形式来表示地址，那么，准确地记住我们要 8080 微处理器转移到什么地方是比较困难的。你可以看到，我们在 JZ，JC 和 JNC 这三条指令所使用的符号地址之后，还保存了一个零(0)。这就会使得我们记住，要为〈B_3〉即这条指令的最高地址字节留下一个地方。请读者记住，转移指令被装入

存储器时，不管我们使用什么表示法或符号地址，存储这条指令需要三个存储单元。

我们给例 2-37 增添了四条指令，因此，用来存储这四条指令需要增加十个存储单元。第一条比较指令用来判定是否已经输入了疑问号(?)。如果已经输入了疑问号，这条 JZ 指令被执行后，8080 微处理器会把程序控制权转到其它程序段。如果输入的字符不是疑问号，则把这个字符与 260(B0) 进行比较。如果输入的数值小于 260(B0)，则进位标识位会置逻辑 1。如果该输入数值等于或大于 260(B0)，则将进位标志位清零。

请读者注意，从键盘输入的数据小于 260 (B 0) 时，只有进位标识位是逻辑 1，这时，程序指令把小于 ASCII 0 (260, B 0)的字符都屏蔽起来。只有进位标识位为逻辑 1 时，8080 才执行从 JC 到 START 的指令。

第三条 CPI 指令用来确定输入的 ASCII 字符的值是否比 272 (BA) 小。即它们是不是 ASCII 字符值 0～9 (260～272, B 0～B 9)；它们是不是要被压缩的值。在执行第三条 CPI 指令以后，如果按下了一个数字键，则进位标识位是逻辑 1，因为 ASCII 字符 0～9 (260～271, B 0～B 9) 的各个值小于 CPI指令的立即数字节。因此，不会执行这条 JNC 指令。如果按下了一个非数字键，比较指令会使进位标志位清零，并执行 JNC-START 指令。

为什么我们首先把疑问号作为输入字符测试，而不首先测试 ASCII 数字字符 (0～9) 呢？我们测试疑问号，可以在测试 ASCII 数字字符之前进行，也可以在测试 ASCII 数字字符之后进行，这种情况如例 2-38 所示。

例 2-38 先测试 ASCII 数字字符

```
START,  IN      /从第 015 (十六进制数 0 D)号
```

```
        015     /设备输入一个 ASCII 字符。
        CPI     /这个字符小于 260，
        260     /(十六进制数 BD，ASCII 0)吗？
        JC      /是的。进位标识位置位，然后
        START   /把这个字符屏蔽，
        0       /并返回到“START”。
        CPI     /这个字符小于 272
        272     /(十六进制数 BA，ASCII。) 吗？
        JC      /是的。进位标识位置位，所以
        CHROK   /这是一个有效字符(0～9)，
        0       /所以，转移到“CHROK”(字符 OK)。
        CPI     /它不是字符 0～9 的某一个。
        277     /它是疑问号吗？
        JNZ     /不是，它不是疑问号。
        START   /所以，这回到这个程序
        0       /的“START”。
          •     /如果该字符是疑问号，则
          •     /8080 执行存储
          •     /在这里的指令
          •
CHROK,  AN 1    /如果输入的字符是 0～9，那么，
        017     /8080 执行这条指令。
          •
          •
          •
        ETC
```

面向位操作

除了测试一个完整的 8 位字外，还可以测试一个数据字的

表 2-24　　三种不同的比较情况

A 寄存器的内容	比较字节	进位标志
257（AF）或小于	260（B0）	逻辑 1
260（B0）	260（B0）	逻辑 0
261（B1）或大于	260（B0）	逻辑 0

表 2-25　　把 ASCII 字符与 272 (B9)比较

A 寄存器的内容	比较字节	进位标志
271（B9）或小于	272（BA）	逻辑 1
272（BA）	272（BA）	逻辑 0
273（BB）或大于	272（BA）	逻辑 0

各个位，以确定它们是逻辑 1 还是逻辑 0 。正如大多数其他软件程序一样，有许多方法可以用来分别测试一个 8 位字中的各位。我们可以假设，当从第 125（55）号外部设备输入一个 8 位的数据字时，你希望检测A寄存器的 D_3 位。

例 2-39　使用循环移位指令和进位标识位来测试所选择的一位。

```
IN      /从第 125(十六进制数 55)号
125     /输入设备输入这个 8 位数据字。
RRC     /把 A 寄存器的内容循环右移四次。
RRC
RRC
RRC
JC      /如果标识位（移入
BITOK   /进位）是逻辑 1，则转移到
```

```
0       /"BITOK"（这1位是OK）。
•       /否则，将执行存储在这里的
•       /指令。
```

在例 2-39 中，把一个数据字从第 125（55）号外部设备输入到A寄存器，A寄存器的这些数据位循环右移四次，每次右移一位，直到把 D_3 移入进位为止。然后用条件转移指令对进位标识位进行测试。进位标志位的状态表示进位的状态。如果进位标识位是逻辑 1（把逻辑 1 移入进位所产生的），则只执行 JC（进位转移）指令。在执行循环移位指令后，如果 D_3 是逻辑 0，则不执行这条指令后的 JC 指令。

如果你希望 D_3 是逻辑 1 时，程序控制不转移，而只有 D_3 是逻辑 0 时，8080 才转移，那么，这个程序应该做出什么改变呢？这个问题看起来比实际作起来好象要困难得多。其实，只要用一条 JNC（无进位转移）指令代替 JC（进位转移）指令，零状态就很容易测试出来。

如例 2-40 所示，使用"与"指令也可以完成位测试操作。在例 2-40 中，使用了一个屏蔽字节 010（08）。在这条 ANI指令被执行之后，只有 D_3 是逻辑 1 时，A寄存器的内容才不是 0。如果 D_3 是逻辑 1，则执行 JNZ（非零转移）指令。如果出现这种情况，存贮在由 JNZ 指令的第二和第三个字节所指定的存储器地址单元的指令是将要执行的下一条指令。如果D_3是逻辑 0，则不执行 JNZ 指令。然而，要执行的下一条指令是存储在这条三字节指令 JNZ 之后的指令。假设你要 8080 在 D_3 是逻辑 0 时才转移，那么这个操作应该怎样实现呢？

例 2-40 用 ANI 指令和零标识位测试 A 寄存器的某一位。

```
IN      /从第 125(十六进制数 55)号
125     /输入设备输入这个 8 位的数据字。
ANI     /用 010 (十六进位制 08)
010     /屏蔽 D3 除外的各位。
JNZ     /如果 A 寄存器的内容不是 0 (而是 00001000),
D 3 IS 1 /则执行 JNZ-"D 3 IS 1 "指令
0       /(D3 是 1)。
•       /只有 D3 是逻辑 1 时, 8080 才执行
•       /存储在这里的指令。
```

解决这个问题的最简单的办法应该是把 JNZ 指令改为 JZ 指令。把 JNZ 指令改变为 JZ 指令，只要把 JZ 指令的操作码存储在 JNZ 指令的操作码原来所存储的存储器单元就行了。这样做之后，只有当A寄存器的 D_3 是逻辑 0 时，才执行这条 JZ 指令。如果 D_3 是逻辑 1，则不执行 JZ 指令。如果不把 JNZ 指令改变成 JZ 指令，也可改变输入端口的电路。只要简单把 D_3 位反相就行了。但是，你已经看到，只要改变判定转移发生的条件，改变这条指令是很容易的。

这是硬件和软件折衷的许许多多的例子之一。硬件和软件折衷就是指改变软件或许比改变硬件更容易，或者改变硬件比改变软件更容易，这当然应该根据要被处理的具体问题而定。在这个例子中，我们假定改变软件比改变硬件容易。

使用循环移位指令后，即使信息位的位置改变了，但是，整个 8 位字仍旧被存储在A寄存器里。使用 ANI 指令时，则有些数据被损坏。如果使用 AND(与)指令来测试其它位置上的位是逻辑 1 或逻辑 0，那么应该暂时把这个数据字存储起来。如果使用循环移位指令，那么被测试的字则不需要暂时存储起来。

多位的测试操作

你已经知道怎样使用旋转指令和 AND(与) 指令，来分别测试一个 8 位字的某一位。那么测试同一个 8 位字的若干位应该怎样进行呢？为了测试同一个 8 位字的若干位，可以反复使用移位指令，一次移动一位，进行测试，如例 2-41 所示。

在例 2-41 中，我们无意把优先权已经分配给了进行测试的 8 位字的每一位。我们建立的优先权是：D_3 的状态比 D_4 的状态更为重要；D_4 位的状态比 D_6 位的状态更重要。这种情况出现，是因为要首先测试数据位 D_3 位，看看是逻辑 1 还是逻辑 0 。如果 D_3 位是逻辑 1 ，则执行 JC 指令。如果执行 JC 指令，那么，将不测试数据位 D_4 和 D_6，因为 8080 微型计算机会自动转移到该程序的其他部分去执行。如果数据位 D_4 是逻辑 0 。则执行 JNC 指令，并且不测试数据位 D_6。只有数据位 D_3 是逻辑 0 。数据位 D_4 是逻辑 1 ，才测试数据位 D_6。假设我们必须改变此优先权（即测试数据位的次序），那么，应该怎样改变呢？

例 2-41 用循环移位指令测试一个 8 位字的若干位

```
IN
125      /输入这个状态字。
RRC      /对 D3 循环右移
RRC      /进入进位
RRC
RRC
JC       /如果 D3 等于逻辑 1，
D 3 IS 1 /则执行这条转移指令。
0
RRC      /否则，程序执行来到这儿。
```

```
JNC     /现在对 D4 位循环移位并测试 D4 位。
D 4 IS 0 /如果 D4 等于逻辑 0，
0       /则执行这条转移指令。
RRC     /现在测试 D6。
RRC
JC      /如果 D6 等于逻辑 1，
D 4 IS 0 /则执行这条转移指令。
0
•
•
•
```

假设使用这样的优先权：$D_6 > D_3 > D_4$。使用循环移位指令，以这种先后次序来测试这些数据位的最简单的程序如例 2-42 所示。

通过分析例 2-41 和例 2-42 这两个程序例，你就可以明白：如果必须改变 8 位数据字的这些位的优先权的话，那么改变它的程序指令可能是棘手的和艰难的，由于分别跟踪每一位和记录这些位移位的方向是困难的，因此必须把它们循环移位，进入进位。单独测试每位的更为容易、更有效地利用存贮器的存贮空间的方法是使用 ANI 指令；如例 2-43 所示。

在例 2-43 中，把将要被测试的三位暂时存储在 B 寄存器里，然后把它们移入到 A 寄存器，供每一个与操作或者屏蔽操作用。

例 2-42 按数据位 D_6，D_3 和 D_4 的顺序进行测试

```
IN
125     /输入这个状态字。
RLC     /将 D6 循环移入进位。
RLC
```

```
JC      /如果 D6=1，则执行这条转移指令。
D 6 IS 1
0
RLC     /现在把 D3 循环移入进位。
RLC     /
RLC
JC      /如果 D3=1，则执行这条转移指令
D 3 IS 1
0
RRC     /对 D4 循环移位，进入进位
RRC
JNC     /如果 D4=0，则执行这条转移指令。
D 4 IS 0
0
•
•
•
```

例 2-43 使用 ANI 指令测试一个字的三位数据位

程序 1

```
IN
125
MOVBA
ANI
OO4     /测试 D2
JNI     /如果 D2=1，
D 2 IS 1 /则执这条转移指令。
0
MOVAB
ANI
```

程序 2

```
IN
125     /输入这个状态字。
MOVBA/把该字存入“B”。
ANI
100     /测试 D6
JZ      /如果 D6=0
D 6 IS 0 /则执行这条转移指令。
0
MOVAB
ANI
```

```
010      /测试D3,                004      /测试 D2。
JZ       /如果 D3=0,             JNI      /如果 D2=1,
D 4 IS 0 /则执行这条转移指令。   D 2 IS 1 /则执行这条转移指令。
0                                0
MOVAB                            MOVAB
ANI                              ANI
100      /测试 D6。              010      /测试 D3。
JZ       /如果 D6=0,             JZ       /如果 D3=0,
D 6 IS 0 /则执行这条转移指令,    D 3 IS 0 /则执行这条转移指令。
0                                0
•                                •
•                                •
•                                •
```

你可以看到，使用 ANI 指令有许多优点。你会发现，甚至在被测试的几位的先后顺序改变时，该程序的长度仍旧保持不变。此外，你只要看看 ANI 指令的屏蔽位(立即数据字节)，就可以较容易地知道正在被测试的数据位的先后次序。

位测试的循环(递归位测试)

在位测试的程序例子中，例 2-39～例 2-43，这些程序一次只测试一位。例如，如果 D_5 是逻辑 1，则可以执行 JNZ 指令，以便把程序的执行转到该程序的另一部分。但是，如果数据位 D_3 是逻辑 0，则执行其他一些操作。

假设 8080 微型计算机在执行该程序的其余部分之前，读者需要 8080 等待从一台输入设备输入的一个数据字节 的某一 位是逻辑 1 或逻辑 0，那么，这个程序应该怎样编写呢？

例 2-44 等待一位数据变成为逻辑 0

```
START,  IN      /从第 125(十六进制数 55)号输
        125     /入设备输入这个标识字。
```

```
ANI    /用 010(十六进制数 08)
010    /屏蔽各位(D3 位除外)
JNZ    /如果 D3=1，则 8080 执行这条
START/JNZ 指令，返回到"START"。
0
•      /只有 D3 等于逻辑 0 时,8080 才
•      /执行存储在这儿的指令。
```

8080 微型计算机执行例 2-44 的程序时，它处在这三条指令的循环之中，直到从第 125(55)号输入设备输入的数据字节的 D_3 置为逻辑 0 时为止。这个数据位等于逻辑 0 时，8080 不会再执行 JNZ 指令，因为非零的条件再也得不到满足。8080微型计算机从存储在这条三字节转移指令后的指令起，开始执行该程序的其余部分。如果在这一循环之前，数据位 D_3 是逻辑 0，那么这些循环指令只执行一次。这条JNZ 指令将被执行吗？当该程序的这一部分被执行后，如果数据位 D_3 等于逻辑 0，将不执行这条 JNZ 指令。

置位/复位操作

读者也可以给 8080 编程序，用 AND(与)和 OR(或)这两条指令使数据位分别置位或清零，这正如你给 8080 编程序分别检查数据位是逻辑 0 还是逻辑 1 一样。我们通常使用 AND(与)指令把某个数据位，或一组数据位清零。我们可以用 OR(或)指令来使某个数据位或一组数据位置逻辑 1。为了把A寄存器的数据位 D_2 置逻辑 1，读者可以用例 2-45 所示的一组程序指令给 8080 微型计算机编程序。

例 2-45 D_2 位的置位操作

A寄存器的内容*			助记符
二进制	八进制	十六进制	
10000010	202	82	MOVAM
			ORI
10000110	206	86	004
10000110	206	86	MOVMA

* 这条指令执行后。

在例 2-45 里，从存储器送来的数据字只有 1 位置逻辑 1。如果我们希望 D_5，D_4 和 D_3 位都置为逻辑 1，那么，我们可以使用 070(38)这个立即数据字节。我们假设，原来让 1 位置逻辑 1 的这个数据字已经被存储在存贮器。我们还假设寄存器对 H已经存放了这个数据字节的适当地址。

从输入设备得到这个数据字，并且把其中的 1 位或者多位置逻辑 1，然后把这个新的数据字输出给输出设备，这是很容易的。请读者注意，A寄存器总是存放将要使这些位置位的那个数据字。

单个数据位或者若干组数据位怎样才能被清除为逻辑零呢？读者可以使用 ANI 指令。为了把数据位 D_5、D_4 和 D_2 清除为逻辑零，可以执行例 2-46 所列出的程序。

例 2-46 位清零或者位复位指令

A寄存器的内容*			助记符
二进制	八进制	十六进制	
10111111	277	BF	MOVAM
			ANI
10001011	213	8 B	213
10001011	213	8 B	MOVMA

* 这条指令被执行后。

在例 2-46 里，A 寄存器的 D_5，D_4 和 D_2 位被清除为逻辑零状态。读者在我们屏蔽一个 ASCII 字符的位时的应用中已经看到了这个操作。

小　结

总之，读者已经弄清楚了 8080 微型计算机的大多数基本指令。有许多指令，我们还没有涉及到。但是，我们坚信，你可以在继续学习 8080 的高级指令之前，必须懂得和能够运用这些基本指令。应用这些基本指令去完成编制有用的软件的任务，这是下一章要讨论的问题。

即使我们已经讨论了的指令比 8080 的尚未讨论的指令少得多，但是，大多数程序(60%～80%)都是由这些基本指令构成的。这对于有些读者来说或许会感到奇怪。更高级和功能更强的指令犹如在“蛋糕上抹了奶油”，应用它们，就程序执行的时间而言，使程序执行更快，就存储器的使用而言，所需要的存贮空间则更少，就工程之间的相互适应性而言，则更灵活。

第三章　子程序与基本指令的应用

到现在为止，读者已经知道了怎样把数据传送指令、逻辑指令、算术指令和转移指令结合起来编制简短的程序。我们马上就会明白，怎样编写长而复杂的程序。

假设读者为医院的微型计算机控制的病人监视系统编写了一个程序。在程序的许多部分，必须把与病人有关的数据输入给微型计算机；例如，病人的名字、年龄，社会保险帐号(SSAN) 和地址。对于这种具体程序来说，可以把电传打字机，或者键盘显示器 (CRT) 用作为数据输入设备。这个程序的开始部分可以完成图 3-1 所示的任务。

当读者需要从 CRT 输入数据时，如果 CRT 显示器终端的输入子程序需要 10～15 个存储器单元，不一定要在不同的程序段都编写上相同的输入子程序。你希望把子程序只写一次，然后当数据必须由 CRT 输入给微型计算机时，使用这个子程序，这才是合理的设计思想。这种程序设计思想如图 3-2 所示。

程序可以这样编写：当每一次从 CRT 输入一个字符时，8080 微型计算机转移到 CRT 子程序，如图 3-2 的右边部分所示。8080 转移到 CRT 子程序时，这个输入子程序指令只许输入一个字符。然后，把程序的控制转移到主程序。在图 3-2 中，在许多不同的程序段上，要从 CRT 输入数据。如果使用转移指令控制转到 CRT 子程序，那么，从 CRT 输入了一个字符之后，8080 应该怎样返回到主程序呢？主程序的存储单

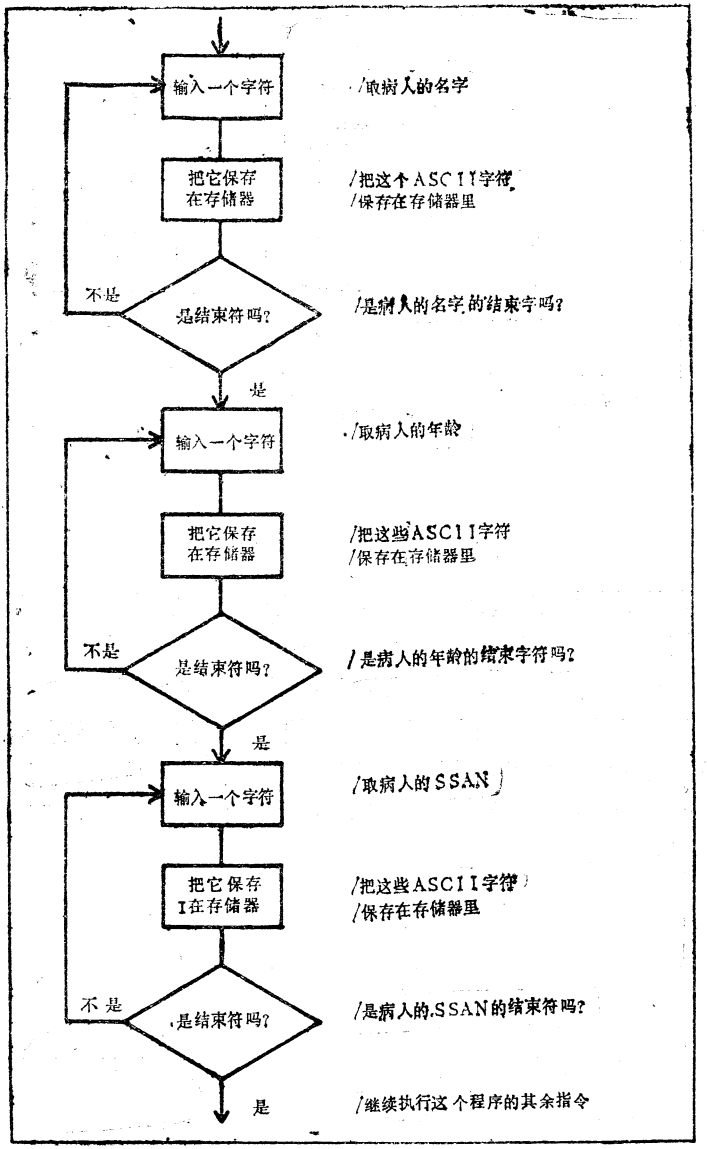


图 3-1 病人监控程序的流程图

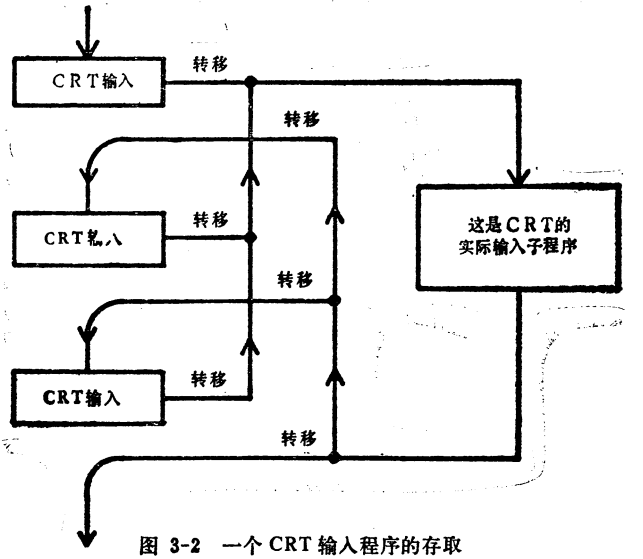


图 3-2 一个 CRT 输入程序的存取

元存储了包括三条转移到 CRT 输入子程序的转移 指令，可惜它们存储单元的地址是不相同的。因此，在 CRT 输入子程序的末尾的一条转移指令，不能用来控制程序返回到主程序的三个不同的地址。实际上，如果你想使用转移指令，记住甚至计算重返主程序的正确地址，对于 8080 来说，是不容易的。因此，下面要讨论一条新的指令，即 CALL（调用）。这条 CALL 指令的操作码是 315（CD）。

调用子程序

调用（CALL）指令与条件转移指令和非条件转移指令一样，也是一条三字节的指令。这条指令的第二个和第三个字节包

含被调用的子程序的起始地址的低 8 位和高 8 位。当执行 CALL 指令时，程序执行控制转移到被存储在 CALL 指令的低位字节和高位字节所寻址的存储单元的指令。就这一点而言，CALL 指令所执行的操作与转移指令所执行的操作完全相同。

在第一章，我们简单地介绍了所谓程序计数器（PC），它是一个 16 位的内部寄存器。程序计数器存储要执行的下一条指令的 16 位地址。当 CALL 指令被执行时，程序计数器(PC)的 16 位地址正指示到紧跟在这条三字节的 CALL 指令之后的那条指令。程序计数器的 16 位地址存储在堆栈。

什么叫堆栈呢？堆栈就是读/写存储器的一部分，我们把这部分指定作为堆栈区。读者怎样指定堆栈区呢？读者已经知道如何使用 LXIB，LXID 和 LXIH 这三条指令。还有一条 LXISP 指令，包含三个字节。这条指令用来把它的第二和第三个字节所包含的地址装入 8080 微处理器的内部堆栈指示器(SP)，该指示器是一个 16 位的寄存器。这个堆栈指示器寄存器用来提供读/写存储器地址，该单元地址将供任何堆栈操作用，**因此，当执行 CALL 指令时，16 位堆栈指示器寄存器的内容用来提供有效读/写存储单元的存储器地址，程序计数器(PC)的内容可以被保存在这里。**

因为堆栈指示器存储一个 16 位地址，因此，你可以把这个堆栈指示器地址选定在 8080 微型计算机系统的读/写存储器的任何部分。请读者记住，LXISP 指令与其它 LXI 指令是类似的；它也是一条三字节的指令，第二个字节包含堆栈地址的低 8 位字节，第三个字节包含堆栈地址的高 8 位字节。

当执行 CALL 指令时，把程序计数器的内容保存在读/写存储单元，存储器单元的地址由堆栈指示器（SP）提供。然后把紧跟在 CALL 指令的操作码后的两个地址字节装入程序

计数器（PC），8080 微处理器接着从这个地址开始执行子程序。

8080 微处理器执行这条三字节的 CALL 指令和被调入的子程序的全部指令后，需要 8080 从紧跟在这条三字节的 CALL 指令后的指令起，继续执行程序。这条指令就是 MOVAB，它的地址单元是 x＋3，如图 3-3 所示。

8080 执行图 3-3 中的 CALL 指令时，把 MOVAB 指令的地址存储在堆栈，作为返回主程序的返回地址。如果X 代表 CALL 指令的地址，那么，应该把地址 x＋3 存储在堆栈。请读者记住，8080 所用的所有地址的宽度都是 16 位，即使读者的 8080 微型计算机系统的存储器的容量只有一两千个字，也是如此。如果存储器单元的宽度只有 8 位，8080 怎样才能把 16 位返回地址保存在读/写存储器呢？我们就用读/写存储器的两个连续的存储器单元来存储这个 16 位的返回地址，一个单元存储这个 16 位地址的低 8 位，一个单元存储 16 位地址的高 8 位。当执行 CALL 指令时，8080 自动地把这个返回地址保存在堆栈。这种操作方式就叫做把信息压入堆栈。

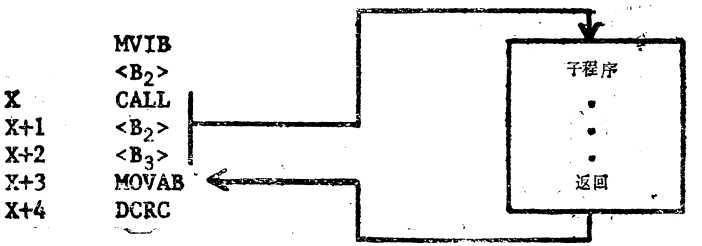


图 3-3　子程序被执行后，要被执行的指令

8080 一旦把子程序的指令执行完毕之后，它怎样知道什么时候把返回地址弹出堆栈，并把它装入程序计数器呢？8080

为了完成这项任务，**不管怎样，任何子程序结尾处都必须用一条返回指令**。这些返回指令之一就是 RET。这条 RET 指令的操作码是 311（C 9）。子程序末尾的 RET 指令被执行时，8080 微型计算机自动地从堆栈弹出这个返回地址。然后把这个 16 位的返回地址装入程序计数器 PC，于是 PC 就指示到紧紧跟在 CALL 这条三字节指令后面的那条指令的地址。如图 3-3 所示，程序控制应该返回到 MOVAB 指令。

这时，读者不必考虑返回地址实际上是怎样存储在堆栈的，或者怎样从堆栈取出的。但是，读者必须确信，在试图调入任何子程序之前，在堆栈（CP）已经被装入了读/写存储器的一个地址。我们在前面说过，堆栈的位置可以安排在读/写存储器的任何地方；并且通过执行三字节指令 LXISP，就能给堆栈预置初值。例如，只要执行例 3-1 所包括的指令，就可以把堆栈预置初值。

例 3-1 用 LXISP 指令对堆栈指示器进行装入操作

八进制数	助记符	十六进制数
061	LXISP	31
300	(B_2)	C 0
003	(B_3)	03

在例 3-1 中，LXISP 指令被执行之后，就把地址 003,300（03 C 0）装入 8080 微处理器的堆栈指示器里。对于实用的堆栈来说，它必须是一个读/写存储器地址。现在，读者知道，当 8080 执行 CALL 指令时，它把其返回地址压入堆栈。这种操作出现时，为了提供将用来存储返回地址的第二个字节的存储单元的地址，存储在这个堆栈的指示器的地址是加 1 还是减 1 呢？

当信息被压入堆栈时，堆栈指示器减“1”；当信息从堆栈弹出时，堆栈指示器增“1”。如果你的 8080 微型计算机系统的读/写存储器只有 1024 个字的存储空间，那么，它的地址可以从000000 到 003377 (0000 到 03 FF)。如果属于这种情况，读者可以把堆栈指示器安排在读/写存储器的顶部，这正如例 3-1 所示，把堆栈指示器置于 003300 (03 C 0) 号地址单元。实际上，003300 (03 C 0) 号地址只不过是堆栈指示器的静止位置而已。

如果读者把堆栈指示器置于 003300 (03 C 0) 号地址来调入子程序，你应在地址为 003277 和 003276 (03 BF 和 03 BE) 这两个存储器单元找那个返回地址。请读者记住，在堆栈中保存一个 16 位的地址；因此，需要两个读/写存储器单元才能存储这个返回地址。8080 微型计算机的大多数用户并不考虑数据存储在堆栈的具体顺序。但是读者为了取出信息方便，通常先把高位地址字节或高位数据字节保存在堆栈。这样，返回地址的高位字节应该存储在第 003277 (03 BF) 号存储器地址单元；返回地址的低位字节应该存储在第 003276 (03 BE) 号存储单元。注意，第 003300 (03 C 0) 号存储单元没有存储任何信息，003300 号地址单元与 LXISP 指令一起使用，用来给堆栈指示器预置初值。读者从这里可以得出结论；**在数据或地址信息被压入堆栈之前，堆栈指示器一直是向着存储器较低的地址减 1。在数据或地址信息被弹出堆栈之后，堆栈指示器向存储器的较高地址加 1。**请读者注意，我们说过，数据和地址信息都可以被保存在堆栈，但是，我们只讨论了 8080 执行 CALL 指令时，怎样把地址信息存储在堆栈的问题。至于怎样把数据值保存在堆栈的问题，将在下一章讨论。

在子程序已经被调用之后，如果读者要检查读/写存储器

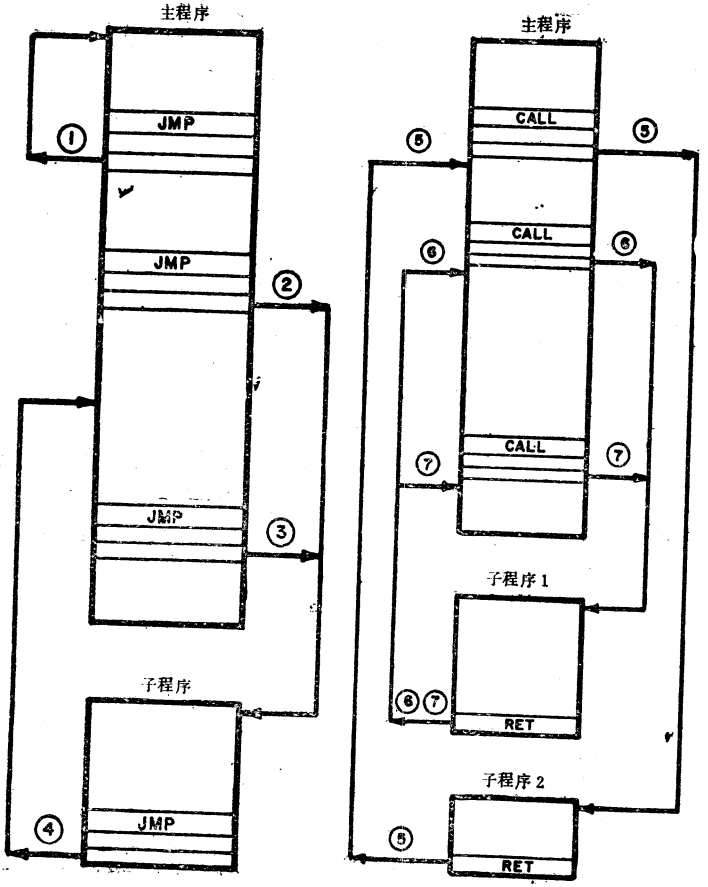


图 3-4 CALL 和 JMP 两条指令的区别

中的堆栈区，那么你还是应该找到堆栈指示器所用的存储在读/写存储器单元的返回地址。这是怎么成为可能的呢？堆栈区是读/写存储器的一部分，因此，8080 执行返回指令时，它

把堆栈指示器指示到的两个存储器地址单元的内容从存储器读出，送入程序计数器。为了清除返回地址，什么也不写入堆栈。从堆栈弹出一个 16 位的数据值时，只发生读存储器操作。这种操作与执行数据传送指令所得到的结果相似。如果 8080 执行 MOVAB 指令，则 B 寄存器的内容仍旧没变，因为 B 寄存器的内容被复制在 A 寄存器里。从堆栈弹出一个返回地址时，出现同样的复制操作过程。8080 从堆栈读出返回地址时，不改变读/写存储器单元的内容。但是，8080 执行下一条 CALL 指令操作时，将把现行返回地址再次写入这两个地址单元。堆栈区是被反复使用的，不需在堆栈上为每一条 CALL 指令或每个子程序设置两个存储单元。

在图 3-4 里，有三次调用和返回循环，分别标上了数字 5、6、7。你可以看到，每当调入一个子程序时，好像子程序被插入在主程序之中似的。子程序被执行完毕后，程序控制返回到主程序。图 3-4 的指令执行顺序如图 3-5 所示。

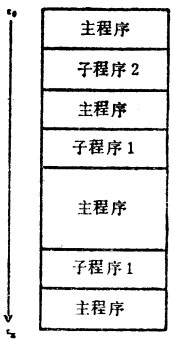


图 3-5 CALL 指令被使用时，程序执行的顺序

要被执行的第一个子程序是第 2 号子程序。主程序被启动后，立刻调入第 2 号子程序。第 2 号子程序的末尾的返回指令被执行之后，8080 执行主程序的一小部分。然后 8080 执行调入第 1 号子程序的 CALL指令。8080 微型计算机执行完第 1 号子程序之后，返回到主程序。然后它执行了主程序的主要部分后，再调入第 1 号子程序。请读者注意，虽然第 1 号子程序被调入了两次，但是在程序中，它只被写了一次。8080微型计算机第二次从第 1 号子程序返回时，执行主程序的其余的指令。

延时子程序

现在，我们要讲述典型的延时子程序。8080微型计算机以2 MHz的频率操作（周期为500 ns）时，用这个子程序产生0.200秒（即200毫秒）的延时。在程序指令的控制下，需要产生长时间的延时时，我们可以使用像这样的延时子程序。8080微型计算机的每一条指令需要的执行时间是已知的，我们可以利用这一点。图3-6列出了8080的每一条指令所需要的时钟周期。

为了确定执行一条指令所需要的时间，必须用执行这条指令所需要的时钟周期乘以微型计算机的周期时间。大多数8080微处理机集成电路都是用一个2 MHz的时钟驱动的，所以，8080的周期是500 ns，如果时钟频率是1 MHz，则周期是1000 ns,即1 μs；时钟频率是750 kHz，则周期是1.33 μs。

表3-1列出了8080的一些指令的助记符，这些指令我们在本章和前面两章已经讨论过了。该表还给出了以三种不同的

表 3-1　8080的一些典型指令所需要的执行时间

指　令	时钟周期	500 ns	1 μs	1.333 μs
JMP	10	5 μs	10 μs	13.33 μs
OUT	10	5 μs	10 μs	13.33 μs
RAR	4	2 μs	4 μs	5.33 μs
CPI	7	3.5 μs	7 μs	9.33 μs
INRC	5	2.5 μs	5 μs	6.66 μs

时间周期执行这条指令所需要的时间。

实际上，我们可以把执行一组 8080 指令 所需要 的时间与 8080 微型计算机通过这组指令循环 的次数 相乘。 完成这一任务，加 1 和减 1 指令是特别有用的。 在用来 延时 200 ms 的子程序 （例 3-2 中），逐渐将 E 寄存器 的内容减 1， 一直到零为止。E 寄存器的内容为 0 时，D 寄存器的内容开始减 1 。如果 D 寄存器减 1 ，并没有到零， 8080 则返 回到 E 寄存 器做减 1 循环操作，D 寄存器和 E 寄存器已经被 减到 零时， 8080 才从这个子程序返回。这种操作的流程图如图 3-7 所示。

助记符	说明	指令码(1) $D_7D_6D_5D_4D_3D_2D_1D_0$	时钟(2) 周期
MOV r_1,r_2	寄存器-寄存器传送	01DDDSSS	5
MOV M,r	寄存器-存储器传送	01110SSS	7
MOV r,M	存储器-寄存器传送	01DDD110	7
HLT	暂 停	01110110	7
MVI R,n	立即数送寄存器	00DDD110	7
MVI M,n	立即数送存储器	00110110	10
INR r	寄存器加 1 指令	00DDD100	5
DCR r	寄存器减“1”指令	00DDD101	5
INR M	存储器加“1”指令	00110100	10
DCR M	存储器减“1”指令	00110101	10
ADD r	把寄存器内容加到A	10000SSS	4
ADC r	把寄存器内容带进位加到A	10001SSS	4
SUB r	从A减去寄存器内容	10010SSS	4
SBB r	从A带借位减去寄存器内容	10011SSS	4
ANA r	寄存器内容与A的内容相“与”	10100SSS	4
XRA r	寄存器内容与A的内容“异或”	10101SSS	4
ORA r	寄存器内容与A的内容相“或”	10110SSS	4
CMP r	寄存器内容与A的比较	10111SSS	4

图 3-6 每条 8080 指令所需要的时钟周期数

续

助记符	说明	指令码(1) $D_7D_6D_5D_4D_3D_2D_1D_0$	时钟(2) 周期
ADD M	把存储器内容加到A	10000110	7
SUB M	从A减去存储器内容	10010110	7
SBB M	从A带借位减去存储器内容	10011110	7
ANA M	存储器内容与A相“与”	10100110	7
XRA M	存储器的内容与A“异或”	10101110	7
ORA M	存储器内容与A“或”	10110110	7
OMP M	存储器内容与A比较	10111110	7
ADI n	把立即数加到A	11000110	7
ACI n	把立即数带进位加到A	11001110	7
SUI n	A的内容减去立即数	11010110	7
SBI n	带借位从A减去立即数	11011110	7
ANI n	把立即数和A的相“与”	11100110	7
XRI n	把立即数与A的“异或”	11101110	7
ORI n	把立即数与A的“或”	11110110	7
CPI n	把立即数与A的比较	11111110	7
RLC	A的内容循环左移	00000111	4
RRC	A的内容循环右移	00001111	4
RAL	A的内容连同进位循环左移	00010111	4
RAR	A的内容连同进位循环右移	00011111	4
JMP nn	无条件转移	11000011	10
JC nn	进位转移	11011010	10
JNC nn	无进位转移	11010010	10
JZ nn	零转移	11001010	10
JNZ nn	非零转移	11000010	10
JP nn	正转移	11110010	10
JN nn	负转移	11111010	10
JPE nn	偶校验转移	11101010	10
JPO nn	奇校验转移	11100010	10
CALL nn	无条件调用	11001101	17
CC nn	进位调用	11011100	11/17
CNC nn	无进位访问	11010100	11/17

续

助记符	说明	指令码(1) $D_7D_6D_5D_4D_3D_2D_1D_0$	时钟(2) 周期
CZ nn	零调用	11001100	11/17
CNZ nn	非零调用	11000100	11/17
CP nn	正调用	11110100	11/17
CM nn	负调用	11111100	11/17
CPE nn	偶校验调用	11101100	11/17
CPO nn	奇校验调用	11001001	10
RET	返回	11001001	10
RC	进位返回	11011000	5/11
RNC	无进位返回	11010000	5/11
RZ	零返回	11001000	5/11
RNZ	非零返回	11000000	5/11
RP	正返回	11110000	5/11
RM	负返回	11111000	5/11
RPE	偶校验返回	11101000	5/11
RPO	奇校验返回	11100000	5/11
RST n	再启动	11AAA111	11
IN	输入	11011011	10
OUT	输出	11010011	10
LXI B, nn	立即数装入寄存器对B	00000001	10
LXI D, nn	立即数装入寄存器对D	00010001	10
LXI H, nn	立即数装入寄存器对H	00100001	10
LXI SP. nn	立即数装入堆栈指示器	00110001	10
PUSH B	寄存器对B的内容压入堆栈	11000101	11
PUSH D	寄存器对D的内容压入堆栈	11010101	11
PUSH H	寄存器对H的内容压入堆栈	11100101	11
PUSH PSW	A的内容和标识位压入堆栈	11110101	11
POP B	从堆栈弹出寄存器对B的内容	11000001	10
POP D	从堆栈弹出寄存器对D的内容	11010001	10
POP H	从堆栈弹出寄存器对H的内容	11100001	10
POP PSW	从堆栈弹出A的内容和标识位	11110001	10
STA nn	直接存储A	00110010	13

续

助记符	说明	指令码(1) $D_7D_6D_5D_4D_3D_2D_1D_0$	时钟(2) 周期
LDA nn	直接装入A	00111010	13
XCHG	交换寄存器对D和H的内容	11101011	4
XTHL	栈顶的内容和寄存器对H的交换	11100011	18
SPHL	寄存器对H的内容送到堆栈指示器	11111001	5
PCHL	寄存器对H的内容放到程序计数器	11101001	5
DAD B	把寄存器对B的内容加到寄存器对H	00001001	10
DAD D	把寄存器对D的内容加到寄存器对H	00011001	10
DAD H	寄存器对H的内容自加	00101001	10
DAD SP	把堆栈指示器的内容加到寄存器对H	00111001	10
STAX B	按B、C间接存储A	00000010	7
STAX D	按D、E间接存储A	00010010	7
LDAX B	按B、C间接装入A	00001010	7
LDAX D	按D、E间接装入A	00011010	7
INX B	B和C加1	00000011	5
INX D	D和E加1	00010011	5
INX H	H和L加1	00100011	5
INX SP	堆栈指示器加1	00110011	5
DCX B	B和C减1	00001011	5
DCX D	D和E减1	00011011	5
DCX H	H和L减1	00101011	5
DCX SP	堆栈指示器减1	00111011	5
CMA	对A求反	00101111	4
STC	进位置位	00110111	4
CMC	进位求反	00111111	4
DAA	对A作十进制调整	00100111	4
SHLD	直接存储H和L	00100010	16
LHLD	直接装入H和L	00101010	16

续

助记符	说明	指令码(1) $D_7D_6D_5D_4D_3D_2D_1D_0$	时钟(2) 周期
EI	开中断	11111011	4
DI	关中断	11110011	4
NOP	不操作	00000000	4

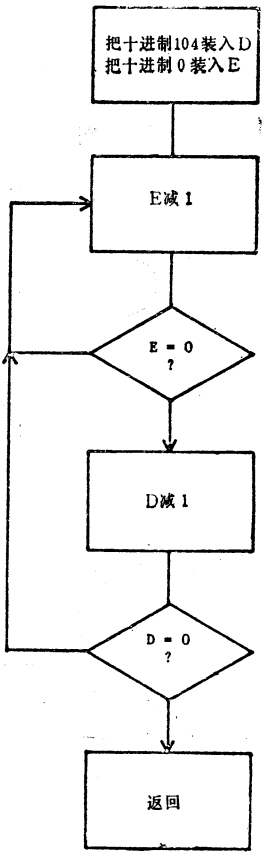


图 3-7 200毫秒延时子程序的流程图

在这个程序（图 3-7，例 3-2）中，所进行的判定操作是以D寄存器（或E寄存器）的内容是否已经被减到 0 为依据的。这种判定程序步是用流程图（图 3-7）的菱形来表示的。我们可以使用什么指令来进行该程序的实际的判定呢？我们可以使用JNZ 指令。8080 微型计算机不断地执行 200 毫秒的延时子程序指令，直到D和E两个寄存器的内容被减到 0 时为止。请读者记住，因为这个程序是延时子程序，所以在它的末尾必须有一条 RET 指令，供 8080 从该子程序返回用。

例 3-2 200 毫秒延时子程序

```
DELAY1, MVIE  /把 000 装入 E 寄存器。
        000   /(八进制数 000＝十六进制数 00)。
        MVID  /把十进制数 104 装入D寄存器
        150   /(八进制数 150＝十六进制数 68)。
WAIT,   DCRE  /E 寄存器的内容减 1。
        JNZ   /如果E的内容不等于 0，则执行
        WAIT  /JNZ 指令，返回到 WAIT。
        0
        DCRD  /现在 E＝0，所以，将D的内容减 1。
        JNZ   /如果D不等于 0，则执行 JNZ
        WAIT  /指令，返回到WAIT。
        0
        RET   /D＝E＝0，所以从该子程序返回。
```

在DELAY1 子程序(例 3-2)的始点，把 000(00)装入 E 寄存器，把 150(68)装入D寄存器。当 8080 执行到 WAIT 处的这条指令时，E 寄存器的内容减 1。因为E寄存器原来存储的内容是 000(00)，所以它被减到 377(FF)。由于这个结果不等于 0，所以 8080 返回到 DCRE 指令。最后当 E 寄存器的内容被减到零时，8080 执行 DCRD 指令。如果该操作的结果不等于 0，那么 8080 返回到 WAIT，DCRE 指令存储在这里。最后，D寄存器的内容减为 0 时，8080 则从该子程序返回。

假设读者需要 8080 产生 30 秒的时间延迟，那么可以用 200 毫秒的延时子程序来产生 30 秒的延迟时间吗？是的，我们可以用这个延时子程序，这只要让 8080 调入子程序 150_{10} 次就行了。例 3-3 就是产生 30 秒延迟时间的子程序。

例 3-3 30秒延时子程序

```
HAFMIN,MVIC   /把十进制数 150 装入C寄存器。
        226     /(八进制数 226，十六进制数 96)。
AGN 200, CALL  /调入 200 毫秒延时
        DELAY1 /子程序。
        0
        DCRC   /C 寄存器的内容减 1。
        JNZ    /如果C寄存器的内容不等于 0，
        AGN 200/则再调入 200 毫秒延时
        0      /子程序。
        RET    /30 秒后，返回主程序。
DELAYI, MVIE /把 000 装入 E 寄存器，
        000   /(八进制数 000，十六进制数 00)
        MVID/把十进制数 104 装入D寄存器。
        150   /(八进制数 150＝十六进制数 68)
WAIT,   DCRE /E 寄存器的内容减 1。
        JNZ   /如果E寄存器的内容不等于 0，
        WAIT /则执行 JNZ—WAIT 指令
        0
        DCRD/现在E＝000，D减 1
        JNZ   /如果E不等于 0，则执
        WAIT/行 JNZ—WAIT 指令。
        0
        RET   /否则，8080在延时 200 毫秒
              /后返回。
```

在例 3-3 中，用C寄存器保存一个定时字节。实际上，这个定时字节是个计数字节，它供 8080 调入 200 毫秒延时子程序的次数记数用。因为装入C寄存器的内容是八进制数 170

(78)，所以调用了DELAY1子程序 150 次。我们可以把例 3-3 列出的子程序简化，如例 3-4 所示。

例 3-4 列出的延时子程序可用来产生 0.200～51.2 秒的延迟时间，这是它的优点之一。那么，怎样才能产生 0.200～51.2 秒的延时呢？如果首先把一个计时字节装入C寄存器，然后在 HAFMIN 处调入 GENDLY(通用延时)子程序，就能够产生 0.200～51.2 秒的延迟时间。如果调入 GENDLY时，C寄存器的内容是 001(01)，那么该子程序需要 0.200 秒的执行时间，8080 才能返回到调入GENDLY 时的主程序。如果C寄存器装入的内容是 002(02)，那么，该子程序需要 0.400秒的执行时间。请读者注意，调入延时子程序(GENDLY)之前，必须把一个数据值装入C寄存器。

例 3-4 30秒延时简化的子程序

```
HAFMIN  MVIC/把十进制数 150 装入C寄存器。
        226    /(八进制数 226，十六进制数 96)。
GENDLY  MVID/把十进制数 104 装入D寄存器。
        150    /(八进制 150，十六进制数 68)。
        MVIE/把十进制数 0 装入E寄存器。
        000    /(八进制数 000，十六进制数 00)。
WAIT    DCRE/E寄存器的内容减 1。
        JNZ    /如果E的内容不等于 0。
        WAIT/则返回到DCRE 指令。
        0
        DCRD/E=0 所以将D减 1。
        JNZ    /如果D不等于 0，
        WAIT/则返回到 DCRE 指令。
        0
        DCRC/已经延迟了 200 毫秒，所以
```

```
JNZ   /将C减1，如果C
WAIT/的内容不等于0，则返回到DCRE指令。
0
RET   /现在，8080已经延期了30秒。
```

在如何使用这个子程序的问题上，例3-4这个程序给了读者很大的灵活性。你可以把一个数装入C寄存器后，再调入这个子程序，使它产生不同的延迟时间。为了产生51.2秒的延期时间，应该把多大的数装入C寄存器呢？为了产生51.2秒的延迟时间，必须把000(00)装入C寄存器。这就是说，要把这个子程序的0.200秒延时子程序段执行256次（0.200秒×256＝51.2秒）

把一个数据值装入寄存器或存储单元，从而这个子程序可以存取这个数，我们把这种操作就叫做给子程序传送变量。用8080微型计算机很容易进行这种操作；其目的往往是为了提高软件的灵活性。

关于例3-2，3-3和3-4所列出的这些延时子程序，需要读者记住的最重要的一点就是：所产生的延迟时间取决于读者的8080微型计算机的时钟频率。当在时钟周期为500 ns的8080微型计算机上执行这个DELAY 1子程序(例3-2)时，我们可以得到0.200秒的延迟时间。如果微型计算机的周期比500 ns快或者慢，那么，你必须把装入D寄存器和E寄存器的数值增大或者减少。

读者也可以用别的指令，另外来编写一个延时0.200秒的子程序，完成例3-2这个子程序的功能吗？当然可以。你可以采用INR和DCR指令。此外，还有别的办法吗？有。你可以用寄存器对加1和减1指令，如例子3-5所示。

例3-5 0.200秒延时子程序，程序中使用一条寄存器对

减1指令。

```
DELAYI  LXID    /把 16，666(八进制数 101，032，
        032     /十六进制数 411 A)装入
        101     /寄存器对D
DECIT   DCXD    /寄存器对D的内容减 1 。
        MOVAD   /把寄存器对D的内容送到A寄存器，
        ORAE    /然后把它与E寄存器的内容相“或”。
        JNZ     /如果结果不等于 0，
        DECIT   /则返回到 DCXD 指令，
        0       /再把寄存器对D的内容减 1。
        RET     /当寄存器对D的内容减为
                /000000(0000)时，则从该子程序返回。
```

在例 3-5 中，寄存器对D装入了 $16,666_{10}$。我们已经计算过了，这个数是为了产生 0.200 秒延期时间，而必须执行这个子程序(例 3-5)中的循环程序的次数。

把 101032(411 A)装入寄存器对D后，利用 DCXD 指令，将寄存器对D的内容减 1 。然后把D寄存器的内容送到A寄存器，把E寄存器的内容在A寄存器中与它进行“或”操作。根据该逻辑操作的结果，决定是否执行 JNZ—DECIT 指令。为什么要执行这些指令呢？请读者记住，寄存器对的加 1 和减 1 指令并不影响 8080 微处理器的任何标识位。因此，为了使该程序能够确定寄存器对D的内容是否为零，必须执行 MOVAD 和ORAE 这两条指令。如果寄存器对D的内容不等于 0，那么，执行这条指令的结果也不是零，(A寄存器中不是零)，就执行 JNZ指令。

如果需要的话，在例 3-5 这个延时子程序的末尾的 RET 指令，可以用DCRC，JNZ—DECIT 和 RET 指令来代替。这

就是说必须用C寄存器的计时字节来调入这个延时子程序。只有C寄存器的内容减到零时，8080微处理机才从这个延时子程序返回。

为了得到0.200秒的延迟时间，一定要把$16,666_{10}$这个数装入寄存器对D，这是怎样确定的呢？我们把指令序列写下来之后，执行在这个循环(DECIT，例3-5)内的指令所需要的周期的数目，可利用图3-6确定，DCXD指令需要5个周期；MOVAD指令需要5个周期，ORAE指令需要4个周期，JNZ指令需要10个周期。8080微型计算机系统的时钟频率是2 MHz，执行这24个周期则需要12 μs。用这个时间，除所需要的延期时间，就确定了必须执行这个子程序的循环的次数。因此，200,000 μs除以12 μs等于16666。把DELAY1子程序(例3-5)的DECIT执行16,666次，实际产生199,992 ms(毫秒)的延迟时间。但是，我们并没有考虑执行LXID和RET这两条指令所需要的时间。这两条指令需要20个周期，即10 μs(微秒)。所以，调用这个子程序实际产生的延迟时间是200.002 ms。这个数字的精度比大多数实际应用所需要的精度高。

如果使用例3-4所列出的这个延时子程序，可以得到最大的延迟时间仅仅是51.2秒，而有时需要更长的延迟时间。为了得到更长的延迟时间，只要多次调入HAFMIN或GENDLY子程序(例3-4)即可。例如，为了产生一小时的延迟时间，可以把30秒延时的子程序(HAFMIN，例3-4)调用120_{10}次。例3-6示出了如何实现一小时的延期时间的程序。

例3-6 调用HAFMIN子程序，产生一小时延时的程序

```
START  LXISP      /把堆栈指示器置位，
```

```
        200         /因为需要调入子程序
        000         /(八进制数 000200，十六进制数 0080)。
        MVIB        /把计数器置为 120₁₀
        170         /(八进制数 170，十六进制数 78)
AGAIN   CALL        /调入 30 秒延期时间
        HAFMIN      /的子程序。
        0
        DCRB        /计数器 120₁₀ 减 1。
        JNI         /如果计数数不等于零
        AGAIN       /则执行 JNZ—AGAIN指令，
        0           /然后再执行 HAFMIN 子程序。
        HLT         /计数器被减到零时，暂停。
```

在例 3-6 中，B 寄存器被装入的数是 120_{10}(八进制数 170，十六进制数 78)。这个数是调用 HAFMIN 子程序的次数。在例 3-6 中，计时字节 120_{10} 被装入 B 寄存器，因为 HAFMIN子程序需要使用C，D和E这三个寄存器。如果不用B寄存器，而是用H或L寄存器，也会是同样容易的。

现在，读者已经看到了许多延时子程序的例子。读者还懂得了：如果知道 8080 执行程序的周期时间和每一条指令所需要的时钟周期，就可以计算出延迟时间。因此，如果读者还不能编写延时子程序，用来产生 4.505 ms，或 23 天，5 小时，32 分和 19 秒的延迟时间，则是没有理由的。

条件调用指令和返回指令

我们已经讨论了转移指令与 CALL（调用）指令和 RET

（返回）指令。正如条件转移指令一样，还有一些条件调用指令和条件返回指令，表 3-2 归纳了这些指令

表 3-2　转移(Jump)，调用(CALL)和返回(Return)指令一览表

条　件	Jump			Call			Return		
无条件	JMP	303	C3	CAU	315	CD	RET	311	C9
非　零	JNZ	302	C2	CNZ	304	C4	RNZ	300	C0
零	JZ	312	CA	CZ	314	CC	RZ	310	C8
无进位	JNC	322	D2	CNC	324	D4	RNC	320	D0
进　位	JC	332	DA	CC	334	DC	RC	330	D8
奇校验	JPO	342	E2	CPO	344	E4	RPO	340	E0
偶校验	JPE	352	EA	CPZ	354	EC	RPE	350	E8
正结果	JP	362	F2	CP	364	F4	RP	360	F0
负结果	JM	372	FA	CM	374	FC	RM	370	F8

请读者记住，转移指令和调用指令都是三个字节指令，而所有返回指令都是单字节指令。大多数时间，读者也许会使用条件转移指令，无条件调用指令和无条件返回指令。但是，如果条件调用指令和条件返回指令用得恰当，它们的作用是很大的。

假设读者有许多信息，例如若干组 ASCII 字符，存储在 8080 微型计算机的存储器中，并且，希望用电传打印机把它们打印出来，或者在 CRT 上显示。首先，为了把那些 ASCII 字符保存在存储器里，你利用了程序，用电传打印机或 CRT 键盘把那些字符输入到存储器里。这个输入程序的特点之一是：当回车符被打印时，回车符和换行都被打印出来。这样就防止一行被打印在另一行的上面。然后，只有信息字符和回车字符被存储在存储器里。这就允许每行少用一个存储器单元来存

储这个 ASCII 字符信息，因为不需要保存换行字符。

现在，读者必须写一个程序，用来打印出被存储的字符。但是，当回车字符从存储器取出，并且被“打印”，即输出给电传打字机时，为了给其余的信息字符提供新的一行，还必须把换行字符也打印出来，请读者记住，换行符被“省略”了，并不作为 ASCII 字符存储在存储器里。在每一个回车符之后，将产生一行换行的程序，如例 3-7 所示。

例 3-7 在存储器中没有存储换行符的 ASCII 字符打印程序

```
START,  LXISP     /堆栈指示器置位，
        STACK     /因为要调用子程序。
        0         /这是一个读/写存储器地址。
        LXIH      /把这个按顺序存储ASCII
                  /字符的地址
        STRING    /装入寄存器对H
        0
NXTCHR,MOVAM      /从存储器取出一个字符。
        CALL      /电传打印机打印这个字符。
        TTYOUT    /或把它在 CRT 上显示 。
        0
        CPI       /被打印的这个字符
        215       /是回车符吗？
        CZ        /如果是，则调用
        LFALSO    /打印换行的子程序。
        0
        INXH      /把H和L寄存器的存储器地址加“1”
        JMP       /然后，从存储器取另一个字符，
        NXTCHR    /并打印这个字符。
        0
```

```
LFALSO, MVIA     /把这个 ASCII 字符(换行)
        212      /装入A 寄存器里。
        CALL     /然后，调入电传打字机
        TTYOUT   /或CRT打印子程序。
        0
        RET

TTYOUT, MOVBA    /把这个字符存入 B寄存器里。
TTYO,   IN       /输入这个UART(通用同步接收器/发送器)
        001      /的状态字。
        ANI      /只保存发送器的标志位。
        004      /如果A＝004,则发送器(打字机) 准备好。
        JZ       /如果A＝000，发送器(打印机)忙。
        TTYO     /所以继续等待发送器空。
        0
        MOVAB    /然后打印A寄存器的内容。
        OUT      /把这个字符从B送到A后，再把它输出给
        000      /UART。
        RET      /把这个字符仍然留在A，返回。
```

我们已经把 STRING 这个符号分配给用来存储 ASCII 信息的第一个字符的存储器单元，这些 ASCII 字符信息是按顺序存储在存储器单元的。因此， STRING 这个 16 位符号地址被作为 LXIH 指令的第二个字节(STRING)和第三个字节 (0) 存储，如果读者要把这个程序输入到 8080 微型计算机的存储器，你必须为 ASCII 信息的第一个字符确定一个 16 位地址，然后把这个地址存储在存放 LXIH 指令的操作码的存储器单元之后。此外我们也给读/写存储器的某个存储器单元分配了一个符号地址 (STACK)。当 8080 执行 LXISP 指令时，它必须

把这个 16 位地址装入堆栈指示器 SP。

另外还有四个符号地址，我们在例3-7，对它们做了定义。这四个符号地址是 NXTCHR,LFALSO,TTYOUT 和 TTYO。这些符号地址都是用来存储转移指令和调用指令的地址字节。

当 8080 微型计算机执行例 3-7 的程序时，首先把一个读/写存储单元的地址装入堆栈指示器 SP。之所以装入堆栈指示器 SP，是因为主程序要调用子程序，因此必须把返回地址保存在堆栈。然后寄存器对 H 装入了一个地址，这个地址指示到存储在存储器的 ASCII 信息的第一个字符。在符号地址 NXTCHR 处，8080 把 ASCII 字符从存储器移到 A 寄存器，然后，8080 调入 TTYOUT 子程序。这个子程序把这个字符打印在电传打字机上，或者显示在 CRT 屏幕上。在本章的另一节，我们要讨论输入/输出（I/O）设备的同步问题，因此，这里我们将不讨论 TTYOUT 子程序如何操作的问题。

当程序执行控制从子程序返回到主程序时，被打印的这个字符仍然保留在 A 寄存器里。因此，当 8080 微型计算机从子程序返回时，A 寄存器的内容与数值215(8D）进行比较，215(8D)是 ASCII 回车字符的数据值。如果刚刚打印的这个字符是回车字符（它用等式或“真零”标识表示），那么，条件调用指令 CZ 调用 LFALSO（打印换行）子程序。只有从存储器读出回车 ASCII 字符值，并且打印它时，才调入这个子程序。子程序 LFALSO 使一换行被“打印”在电传打字机上或显示在 CRT 上。8080 从 LFALSO 这个子程序返回以后，即 8080 跳过调用指令（CZ），它执行 INXH，使寄存器对 H 的内容增1，然后 8080 返回到符号地址 NXTCHR，执行这里的指令。因此，8080 把程序控制送回到这条 MOVAM 指令。

在存储器里不存储换行 ASCII 字符，那么，实际上可以

节省多少个存储器单元呢？正如我们在前面所提及的一样，如果在存储器里存储 80 行字符，那么，为了存储换行 ASCII 字符，则需要增添 80 个存储器单元。但是，并没有节省 80 个存储器单元，因为我们必须编写一个特别子程序(LFALSO)，在主程序中增添一些指令，如 CPI 215 和 CZ LFALSO 0。存储这三条指令需要 11 个存储器单元。所以，如果我们在存储器里存储 80 行字符，则可节省 69 个存储单元。如果只把 11 行或小于 11 行 ASCII 字符存储在存储器里，那么，在存储器里应该存储换行字符，而不另编写专用子程序，(例如，我们在例 3-7 的做法)，这样做则可更有效地利用存储器。

例 3-7 的程序什么时候才使电传打字机停止打印字符呢？即使 80 行 ASCII 字符都被打印完毕之后，电传打字机还是不停止打印。这种情况之所以出现，是因为在所有的信息字符都打印完毕后，并没有给 8080 编上程序指令，使它停止打印操作。这个程序无休止地循环。8080微型计算机没有办法从这个子程序退出，或者把程序控制转到别处。

为了说明某条条件返回指令的用处，我们编写了二进制—ASCII 码的十六进制的字符转换子程序。这个子程序用来把 A 寄存器的低四位(D_3～D_0) 的 4 位数值转换成相应的 16 进制字符的 ASCII 字符值。我们假设：A 寄存器的其余各位都等于零。我们选择了一些 ASCII码，二进制数和 16 进制数这三者的等效值列入了表3-3。

我们编写了例 3-8 的子程序，用来把A寄存器的二进制数内容转换成以 ASCII 码的十六进制数表示的字符。

例 3-8 二进制与 ASCII 码的十六进制的转换子程序

```
BINHEX  ADI   /调入这个子程序时，
        260   /把 260 加到A 寄存器的内容上。
```

表 3-3 十六个字符的 ASCII 码，二进制数和十六进制数

ASCII	二 进 制	十 六 进 制
260	0000	0
261	0001	1
262	0010	2
263	0011	3
264	0100	4
265	0101	5
266	0110	6
267	0111	7
270	1000	8
271	1001	9
301	1010	A
302	1011	B
303	1100	C
304	1101	D
305	1110	E
306	1111	F

```
CPI   /然后把A 寄存器的内容
272   /与 272 (十六进制BA)比较。
RC    /如果A的内容小于 272 (十六进制BA)，
      /则 8080 返回。
ADI   /如果 8080 不返回，
007   /把 007 (十六进制 07)加到A的内容上，
RET   /然后把一个ASCII 字母字符存入A，返回。
```

要把存入A寄存器二进制数值的 D_3—D_0进行转换，8080必须调用例 3-8 所列出的子程序。当 8080 从该子程序返回时，A寄存器将存储一个 ASCII 字符的相应的十六进制数字符。

当 8080 进入这个子程序时，立即把数 260 (B0)加到A寄

存器的内容上。现在，A寄存器的内容，可能是表3-4所列出的16个数中的任何一个。

表 3-4　　把260(BD)加到A寄存器的内容上

在程序的始点时 A寄存器的内容	加260后 A寄存器的内容		ASCII 字　符
二进制	八进制	十六进制	
0000	260	BD	0
0001	261	B1	1
0010	262	B2	2
0011	263	B3	3
0100	264	B4	4
0101	265	B5	5
0110	266	B6	6
0111	267	B7	7
1000	270	B8	8
1001	271	B9	9
1010	272	BA	:
1011	273	BB	;
1100	274	BC	<
1101	275	BD	=
1110	276	BE	>
1111	277	BF	?

我们只要把260（B0)加到A寄存器的内容上，二进制数0000到1001(0～9)就可以正确地被转换成相应的ASCII字符。但是，十六进制的字符A到F（二进制1010～1111）的ASCII字符是不正确的。所以，我们利用CPI 272这条指令来判定A寄存器的内容是不是大于272(BA)，或者等于272。因为A寄存器的内容是260～271（十六进制数B0—B9，ASCII码0—9)，所以，如果进位标识位置于逻辑1，则8080将从该子程序返回。如果A寄存器的内容大于或等于272，则把数值

007(07)加到A寄存器的内容上。结果，就会产生二进制1010-1111的相应的 ASCII 字符。把 007(07) 加到A寄存器 的 内容上之后，8080 把这个 ASCII 字符(A-F)存入A寄存器，然后返回到主程序。

有许多不同的方法可以用来把一种数制转换成另一种数制。在数制转换这一章，还要介绍一些数制换算的方法。但是，现在，读者已经看到了如何应用条件调用指令和返回指令的一些实例。

基本指令的运用

到现在为止，我们已经讨论了下面几类指令：

1．传送指令：寄存器—寄存器传送指令，寄存器—存储器传送指令和立即传送指令。

2．加1指令和减1指令：寄存器加1指令，寄存器减1指令，存储器加1指令，存储器减1指令。

3．输入指令和输出指令：IN 和 OUT

4．简单的寄存器对指令：LXI，INX 和 DCX。

5．逻辑指令和算术指令：ANA，XRA，ORA，CMP，ADD，ADC，SUB 和 SBB

6．条件转移指令，条件调用指令和条件返回指令，无条件转移指令，无条件调用指令和无条件返回指令。

输入/输出设备的同步

我们只要应用上面六种指令，就能编写出程序来，很快地，有效地，以最少的程序执行时间完成某个任务。但是，为了使读者弄清楚 8080 微型计算机是怎样“工作”的，我们必须把输入/输出(I/O)设备与它连接起来，这些设备可能包括微型计算机的面板，电传打字机，CRT 显示器，软磁盘，纸带阅读机，或纸带穿孔机。在本章的程序例子中，其中有许多程序例应用了电传打字机或 CRT 显示器作为输入/输出设备。这是因为这两种设备是很普通的外部设备。现在就让我们再来看看电传打字机的一段非常简单的输出子程序。

为了使用例 3-9 所列出的程序，我们假设电传打字机通过接口与 8080 微型计算机连接，也就是说，使用通用异步接收器/发送器(UART)将电传打字机与 8080 微型计算机连接。利用 UART 把 8080 微型计算机的数据传送给电传打字机，而且，UART 将可以接收电传打字机发送的字符，因此，8080 微型计算机能够输入数据。假设我们的程序例子所使用的电传打字机接收和发送 7 位 ASCII 字符。这些字符带有一位校验位(最高有效位)，此位总是等于逻辑 1 。

例 3-9 简单的电传打字机输出子程序

```
TTYOUT   MOVBA    /把这个字符保存在B寄存器里。
TTYO     IN       /输入这个UART状态字
         001
         ANI      /只保存发送器的标识位
         004      /如果A＝004，发送器(打字机)准备好。
```

```
JZ          /如果A＝000，发送器(打字机)不空。
TTYO        /因此，继续等待发送器(打字机)。
0           /完成现行任务后，
MOVAB       /才能打印A寄存器的内容。
OUT         /把这个字符从B
000         /送到A后，把它输出给UART。
RET         /把这个字符仍旧留在A，然后返回。
```

通用异步接收器/发送器(UART)用于通信，是非常普遍的，关于这个问题，我们已经写了一本专著。UART硬件所完成的功能，用汇编语言程序也能实现；这种汇编程序，人们常常叫它为软UART接口。完成这种功能的程序也已经发表了。

当8080微型计算机调入例3-9所示的子程序时，A寄存器必须存储一个8位字。这个8位字将要传送给电传打字机。在子程序TTYOUT处，8080必须把这个字符保存在B寄存器里。然后，这条IN指令把UART的发送器的状态标识位输入，判明发送器是否忙于发送字符。如果发送器忙，UART的发送器准备好标识位(与第001(01)号输入设备的D_2位连接)将是逻辑0。当UART现在不发送字符时，这个标识位将是逻辑1。UART的其它状态标识位被输入到这个状态字的其它位位置。

例 3-10 在电传打印机上打印B

```
PRINTB      IN          /输入UART的状态位。
            001
            ANI         /只保留发送器(打印机)的状态。
            001         /如果A＝000，发送器不空。
            JZ          /如果A＝004，发送器准备好。
```

```
PRINTB
0
MVIA      /发送器准备好，所以把B的
302       /ASCII值装入A。
OUT       /输出这个8位代码给发送器。
000
RET       /然后从这个子程序返回
```

这个状态字被输入以后，8080微型计算机执行ANI指令，该指令的立即数据字节是004(04)。这样，就屏蔽了这个8位字的其它各个状态位(发送器的状态位D_2除外)。8080执行ANI指令后，如果发送器不空，A寄存器的内容将是000(00)。如果发送器并不在向电传打字机发送字符，A寄存器的内容将是004(04)。这就是说，如果UART现在正在发送一个字符，那么，A寄存器的内容将是000(00)，因此8080执行JZ-TTYO指令。

ANI指令被执行之后，当UART的TRANSMITTERREADY(发送器准备好)标识位最后置逻辑1时，A寄存器的内容将会是004(04)。所以8080不再执行JZ-TTYO指令。相反8080要执行的下一条指令是存储在JZ指令之后的那条指令。这条指令就是MOVAB，它把B寄存器的内容复制到A寄存器里。然后，8080执行OUT指令，从而，UART能够并行地把A寄存器的8位数据内容锁存起来。当一个数据字被锁存到UART后，便开始自动地传送这个字符。这个操作把TRANSMITTER-READY(发送器准备好)标识位清零，以表明UART忙于发送字符。当这个8位字符已经被发送给电传打字机后，为了表示出READY(准备好)状态，发送器准备好标识位置于逻辑1状态。8080微型计算机执行这条OUT指令后，它

便执行 RET 指令，从而返回到调用 TTYOUT 子程序的主程序。

为什么 8080 要执行 IN 001，ANI 004，和 JZ-TTYO 这四条指令呢？这是因为这种指令序列能够使 8080 微型计算机与电传打字机同步。如果 8080 和电传打字机不同步，8080 每隔 100 μs 向电传打字机输出一个新字符（把这个数据字输出给 UART）。但是基本速度是 10 字符/秒的电传打字机打印一个字符需要 100 ms。这就是说 8080 微型计算机向这种慢速的机械打印机输出 1000 个字符，而电传打字机的打印机构才能打印完一个字符！由于两者的速度不同，所以，我们必须编写 TTYOUT 子程序，让 8080 等待 UART 的发送器(发送器的速度与打印机的速度一样慢)，完成现行字符的传送之后，才发送另一个字符。正如读者希望的那样，8080 微型计算机要花去大多数时间去等待发送器（电传打字机）准备好。微型计算机的大多数小系统都是这样。8080 微型计算机花去它操作时间的主要部分等待输入/输出(I/O)设备传送数据或者接收数据。

请读者注意，在例 3-9 中，首先要把发送给电传打字机的字符保存在 B 寄存器里。如果不把这个字符保存在 B 寄存器里，这条指令(IN 000)就会破坏这个字符，因为这条 IN 指令把数据字输入到 A 寄存器里。

假设把这个 ASCII 字符 B 打印在电传打字机上，读者可以利用例 3-10 所列出的子程序。

在例 3-10 所示的子程序中，8080 微型计算机需要等待 UART 的发送器(打印机)的标识位置逻辑 1。当这个标识位是逻辑 1 时，8080 把立即数据字节 302(C2)装入 A 寄存器（这个立即数据字节是字符 B 的 ASCII 值），然后把 A 寄存器的内容输出给 UART，以便发送给电传打字机或 CRT 显示器。有许

多不同的子程序可以用来使电传打字机或 CRT 显示器打印 或显示字符。可是，例 3-10 的子程序的灵活性是有限的。读者不希望必须用其可编程序的 ASCII 字符来复制这个子程序，每次打印一个字符。相反，读者最好使用例 3-11 所示的方法。

例 3-11 灵活地打印字符的方法

```
          .
          .
          .
          MVIA    /把要发送给打印机的 ASCII
          302     /字符值装入 A 寄存器里。
          CALL    /然后调用 TTYOUT 子程序
          TTYOUT/
          0
          .
          .
          .
TTYOUT    MOVBA   /把这个字符保存在 B。
TTYO      IN      /输入 UART 的状态字。
          001
          ANI     /只保存发送器的标识位。
          004     /A=004，发送器准备好。
          JZ      /如果 A=000，发送器不空。
          TTYO    /所以继续等待发送器
          0       /(打印机)空，才能打印
          MOVAB   /A 寄存器的内容。
          OUT     /把这个字符从 B 送到 A 后，
          000     /再把它输出给 UART。
          RET     /8080 返回，字符仍保留在 A。
```

如果使用例 3-11 所示的方法，那么，在 TTYOUT 子程序被调用之前必须把被打印的字符装入A寄存器。在这个例子中，我们使用 MVIA 指令来把这个 ASCII 字符装入A寄存器。如果我们已经把这个 ASCII 字符存储在C寄存器，或者存储在寄存器对H所寻址的存储器单元，那么，也可以使用诸如 MOVAC 和 MOVAM 这样的指令。

8080 微型计算机的用户几乎没有想到，在机械结构上应该把电传打字机的键盘与打字机分开，就象普通的打字机那样。即使电传打字机的键盘和打字机被装在同一个机箱里，它们也是两个独立的不同的输入/输出设备。这就是说，读者可以编写一个程序，把数据从键盘输入，然后，把同一个数据输出给打印机。这种操作就叫做字符回送，完成这种功能的程序如例 3-12 所示。

例 3-12 把键盘字符回送到打字机的程序

```
TTYIN    IN      /输入这个 UART 的状态字
         001
         RAR     /将接收器的标识位循环移入
         JNC     /进位。如果进位是 0，
         TTYIN   /没有键被按下，所以继续等待。
         0
         IN      /按下了一个键，所以把这个 ASCII
         000     /字符输入到A寄存器。
TTYOUT,  MOVBA   /把这个字符保存在B寄存器里。
TTYO,    IN      /输入这个 UART 的状态字
         001
         ANI     /只保存发送器的标识位。
         004     /如果A＝004，则发送器(打字机)准备好
         JI      /如果A＝000，则发送器(打字机)不空。
```

```
TTYO   /所以继续等待发送器(打印机)
0      /完成该操作，然后才能打印
MOVBA /A寄存器的内容。
OUT    /把这个字符从B寄存器
000    /送到A寄存器，再把它输出给UART。
JMP    /这个字符被输出后，8080转回到
TTYIN  /TTYIN，从而
0      /可以输入另一个字符。
```

在例3-12中，8080微型计算机等待着UART从电传打字机或从CRT键盘接收一个字符。当UART接收一个字符时，第001(01)号外部设备（状态字输入设备）的RECEIVER—FULL（接收器满）标识位（D_0位）将被置为逻辑1。在前面的程序例（例3-9和例3-10）中，用同一输入端口不同一位来检测输入这个状态字以后，这个8位字向循环右移一次。这一操作将会使A寄存器的最低有效位（UART的接收器满标志位）循环移入进位位置。如果进位标识位是逻辑0，那么，电传打字机的键盘上的键还没有被按下，接收器没满。如果是这种情况，进位标志位将是逻辑0，并且8080执行JNC指令。该操作将把程序控制回送到IN指令。继续执行状态字输入，标志位测试和JNC指令，直到电传打字机键盘上的一只键被按下后才停止。

当一只键被按下，并且这个键字符被UART接收后，RECEIVER—FULL（接收器满）标识位（逻辑1）将被循环移入进位位置。现在，该进位为“真”状态，表示这条JNC指令将不被执行，而是把UART所接收的这个字符输入到8080的A寄存器里，然后在TTYOUT处，把它保存在B寄存器里。

输入这个ASCII字符以后，程序等待着UART的发送器

准备好，而不发送字符给电传打字机的打印机。该程序进行此操作是通过按序列执行 TRANSMITTER—READY(发送器准备好）标识位的测试指令（我们在前面已讨论过了)来实现的。当 UART 的发送器准备好了时，已经被按下的这只键的 ASCII 值从 B 寄存器送入 A 寄存器。然后把这个 ASCII 字符值输出给 UART，它自动地把这个字符发送给电传打字机的打印机械 CRT 显示器的屏幕。

当这个 ASCII 字符从 UART 输入时，由 IN 命令使 RECEIVER—FULL（接收器满）标识位复位。这一操作是通过连接到 8080 微型计算机的外部设备来完成的。这就允许 UART 所接收的下一个完整的字符再次将 RECEIVER—FULL（接收器满）标识位置位，从而表示 UART 已经接收了前一个字符。UART是一种功能很强的器件。

为什么我们要把 UART 接收器的数据字输入到 A 寄存器，而不输入到 8080 微处理器的其它通用寄存器呢？这是因为，当 8080 执行 IN 和 OUT 这两条指令时，8080 只能把数据在外部设备和 A 寄存器之间传送，所以，为了发送这个 ASCII 字符，在把它输出给 UART 之前，必须从 B 寄存器移到 A 寄存器。

前面的用来回送电传打字机的字符的程序（例 3-12）是一种很有用的程序。我们只要把例 3-12 这个程序的最后那条指令（JMP—TTYIN）用一条返回指令（RET）代替，这个程序就变成了两个子程序。我们调用子程序 TTYIN，来输入一个键盘字符，并把它回送到打印机上。然后，当 8080 从这个子程序返回时，把它留在 A 寄存器和 B 寄存器里。我们调用子程序 TTYOUT，则就能很容易地把 ASCII 字符输出给电传打字机或 CRT 显示器。可是读者得记住，如果调用 TTYIN 子程序，能够输入一个键盘字符，但是，在一只键被按下前，

8080 将要在 TTYIN 循环中等待。**在这段时间，8080 微型计算机什么也不能做。**

我们为什么在例 3-12 所示的子程序的 TTYIN 程序段中要使用一条 RAR 指令来监控 UART 的 RECEIVER—FULL (接收器满)标识位，而用 ANI 指令来监控 TRANSMITTER—READY（发送器准备好）标识位呢？如果我们已经用了这条 ANI 指令来监控 RECEIVER—FULL （接收器满） 标识位，那么，这条指令需要两个存储器单元，一个存储器单元用来存储 ANI 指令的操作码， 一个存储器单元用来存储这条指令的立即数据字节(即屏蔽字节)。 RAR 指令只需要一个存储器单元，所以，节省了一个存储器单元。读者可以使用 RAR 指令，也可以使用 ANI 指令， 但是， 这两条指令的执行结果是相同的。

同样的理由，使得我们选择这条 ANI 指令来监控TRANSMITTER—READY（发送器准备好）标识位，而不采用 RAR 指令。因为这个程序需要三条 RAR 指令才能把 TRANSMITTER—READY（发送器准备好）标识位循环移入进位，需要用一条 JNC 指令来测该标志位是否为逻辑 0 状态。如果我们使用 ANI 指令，则只需要两个存储器单元， 因此， 节省了一个存储器单元。

但是， 使用 RAR 或别的循环移位指令检测这些标识位的状态，存在着一个主要的缺陷。如果读者希望编写一个尽可能好的程序软件，为什么在程序中不应用循环移位指令来检测这些标识位的状态呢？ 这个问题的答案是， 它与 RAR 指令及 ANI 指令是否影响的标识位无关，而是灵活性。

我们假设读者已经编写了一个程序，要把它出售，或者让其它程序设计员使用。在大多数情况下， 其它用户的 8080 微

型计算机系统的输入/输出（I/O）设备的地址与读者编写的这个程序的输入/输出设备的地址是不完全相同的，但是，对于你的程序的新用户来说，为了反映出他们自己的8080微型计算机系统所用的输入/输出设备的地址，改变你的程序的输入/输出指令的地址字节，是非常容易的。但是，假设另一个用户使用 D_4 和 D_5 这两位来分别指示UART的发送器和接收器的状态标识，而读者编写的程序是用 D_0 和 D_2 来指示这两个状态标识，如果读者在自己的程序中只使用了一条RAR指令，而其它用户需要ANI这条两个字节的指令来检测状态标识位 D_4 或 D_5 的状态，那么，插入这条ANI指令或许是很困难的。请读者注意，D_4 和 D_5 是从UART输入的。因此，当你需要具体测试某些标识位时，在你自己的输入/输出子程序中，应该使用这条ANI指令。如果你这样做了，任何用户应用此程序时，不仅能方便地改变输入/输出设备的地址，而且很容易改变这条ANI指令的屏蔽字节。具有这种灵活性的程序如例3-13所示。

例 3-13 在输入/输出（I/O）程序中应用ANI指令带来的灵活性

```
IN          IN
001         360
ANI         ANI
001         040
```

容易把左边的指令改变成右边的指令：

```
IN          IN
001         360
RAR         ANI
            040
```

不容易把左边的指令改变成右边的指令。

电传打字机输入/输出和字符处理

在例 3-12 中， 8080 可以从电传打字机或 CRT显 示 器的键盘接收一个 ASCII 字符， 然后把该字符回送到电传打 字机的打印机或 CRT 显示器的屏幕上。 现在可以编写这样一个程序，能够使你在电传打字机的键盘上打印信息，并且能把作为结果的 ASCII 字符保存在存储器里。例 3-14 列出了执行这些操作的程序。

例 3-14 所列出的这个程序把堆栈指示器 (SP) 指示到一个读/写存储器单元，因为在该程序中的将要执行 CALL 指令。然后， 把 16 个按顺序的存储器单元的第一个的地址装入 寄存器对 H，这 16 个存储单元将被用来存储 ASCII 字符。在这个例子中， C寄存器将用作为字符计数器， 我们用 020 (十六进制 10，十进制 16) 作为它的初始值。然后， 8080 微型计算机调用 TTYIN 子程序，就能从电传打字机的键盘输入字符。

例 3-14 向存储器输入和存储 ASCII 字符

```
INSAV, LXISP   /堆栈指示器置 1 。
       STACK
       0
       LXIH    /把一个读/写存储器地址
       STRING  /装入寄存器对 H，这个地址
       0       /可以用来存储 ASCII 字符。
       MVIC    /要输入和存储的 ASCII
       020     /字符的个数是16(十进制)。
SAVE,  CALL    /取一个电传打字机字符。
       TTYIN
```

```
        0
        MOVMA /把这个字符保存在存储器里。
        INXH  /存储器指示器加1。
        DCRC  /字符计数器减1。
        JNZ   /是16吗？不是，取另一个字符。
        SAVE
        0
          •
          •
          •
TTYIN,  IN    /输入 UART 的状态标识位。
        001
        ANI   /只保存接收器的标识位
        001   /如果 A=001，则按下了一只键。
        JZ    /如果 A=000，则没有按下键。
        TTYIN /所以继续等待，
        0     /一只键被按下。
        IN    /一只键被按下了，所以
        000   /向 A寄存器输入这个 ASCII 字符。
        RET   /把这个字符留在 A 寄存器，
              /8080 返回。
```

子程序 TTYIN 使 8080 微型计算机等待一只键按下，然后，它把这只键的 ASCII 值输入，送到 A 寄存器。于是，8080 微型计算机从这个子程序返回。通过 MOVMA 指令，把这个 ASCII 字符被保存在寄存器对 H 所寻址的存储器单元中。然后，8080 执行 INXH 指令，使寄存器对 H 的内容加 1，并且将字符计数器，即 C 寄存器的内容减 1。如果用这个程序，可以把多少个字符输入并且保存在存储器里呢？答案是 16 个

(十进制)字符，因为预置在C寄存器的数是十进制数16。这个程序将继续把电传打字机的键盘输入的ASCII字符保存在存储器里，直到C寄存器的内容逐渐减量，等于零时为止。这时，8080将不再执行紧跟在DCRC指令后面的JNZ指令，而开始执行该程序的其余指令。

当读者使用这个程序，在电传打字机的键盘上打入16个字符的信息时，也可以把这些信息打印出来吗？我们的回答是否定的，因为没有把这些ASCII字符发送给打印机的指令。因此，当你向电传打字机输入一个信息时，你看不到输入的信息被打印在打印机上。当然，我们可以修改这个程序，当信息输入时能够打印出来。例3-15所示的程序与例3-14的程序比较，除了有用来把输入给电传打字机的信息同时在电传打字机上打印出来的TTYIN子程序以外，其余都相同。请读者注意，这里用ANI指令，而不采用RAR指令。

既然，8080微型计算机执行例3-15这个程序能够使信息回送，那么，读者能看到例3-15中的输入和存储程序的另一个局限性？如果你要把长达43个ASCII字符组成的信息输入给电传打字机，并且要把例3-15的程序指令存储在只读存储器(ROM)里，那么，会发生什么现象呢？你无法改变字符的计数值，因为它是MVIC指令的立即数字节建立的，如果不用固定的计数值，则有利于寻找某种方法，向8080微型计算机发出信号，告诉它这个信息的最后一个字符刚刚被打印完毕。这就是说，我们可以把15个或56个字符所组成的信息输入给电传打字机。怎样可以完成这些操作，你能想得到吗？最简单的方法就是使用一个键盘字符作为一个软标志。在例3-16中，我们将把问号(？)的ASCII字符值(八进制277，十六进制BF)用作为软标志，向8080发出信号，告诉它，信息的最后一个

字符已经被输入给了电传打字机。

在例 3-16 中，堆栈指示器(SP)用一个读/写存储器地址预置后，就把一个读/写存储器地址装入寄存器对 H。这个读/写存储器地址将用来存储 ASCII 字符。然后 8080 调用 TTYIN 子程序。当电传打字机的一只键被按下时，这个键的相应 ASCII 码从 UART 输入，然后把它输出到 UART 的发送器。当 8080 从该子程序返回时，这个 ASCII 值留在 A 和 B 这两个寄存器。然后，8080把 A 寄存器的内容(被接收的这个字符)与 277(BF) 这个立即数据字节进行比较。277(BF) 是疑问号的 ASCII 字符的数值。如果被输入的这个字符不是问号，则比较的结果将不等于零，因此，8080 不执行 JZ-TERM 指令。相反，8080则执行 MOVMA 指令，将这个 ASCII 字符值存储在寄存器对 H 寻址的存储器单元，然后，8080执行 INXH 指令，将寄存器对 H 存储的存储器地址加 1；8080 再执行一条 JMP 指令，返回到 CHARIN，从而 8080 就能输入另一个字符。请读者注意，现在不论包含有多少字符的信息都可以输入，并存储在存储器里。只有当键盘上的问号键被按下，该键的 ASCII 值 277(BF)被 8080 接收时，8080 才执行 JZ-TERM 指令。当 8080 确实转移到 TERM 时，在最后这个字符被保存到存储器之后，它立即把 000(00)保存在存储器里。8080 执行这条 MVIM 指令时，把 000(00)保存在存储器里。疑问号的 ASCII 字符值也被保存在存储器里吗？不，疑问号的代码将并不保存在存储器里，而是将000(00)保存在存储器里。

例 3-15 回送信息的输入—存储程序

```
INSAV,   LXISP  /把堆栈指示器置位
         STACK
         0
```

```
        LXIH    /把一个读/写存储器地址
        STRING/装入寄存器对 H;
        0       /该存储地址能够存储一串
                 ASCII 字符。
        MVIC    /要输入和存储的 ASCII 字符的数量
        020     /是16(十进制)个。
SAVE,   CALL    /取一个电传打字机字符。
        TTYIN
        0
        MOVMA/把这个字符保存在存储器里。
        INXH    /存储器(堆栈)指示器加 1。
        DCRC    /字符计数器减 1
        JNZ     /是16吗? 不是, 取另一个字符,
        SAVE
        0
          •
          •
          •
TTYIN,  IN      /输入 UART 的状态标识位。
        001
        ANI     /只保存接收器的标识位。
        001     /如果 A=001, 则一只键被按下了。
        JZ      /如果 A=000, 则没有键被按下。
        TTYIN   /所以, 继续等待到
        0       /一只键被按下。
        IN      /一只键被按下了, 所以把该键
        000     /的 ASCII 字符输入到 A 寄存器。
TTYOUT, MOVBA /把这个字符保存在 B 寄存器里。
TTYO,   IN      /输入该 UART 的状态字。
        001
```

```
        ANI     /只保存发送器的标识位。
        004     /如果 A=004，发送器(打印机)准备好。
        JZ      /如果 A=000，发送器(打印机)忙。
        TTYO    /因此继续等待发送器(打印机)完成
        0       /现行操作，然后
        MOVAB/才能打印 A 寄存器的内容。
        OUT     /把这个字符从 A 寄存器送到
        000     /B 寄存器，输出给 UART。
        RET     /把这个字符仍旧留在 A 而返回。
```

当回车符被打入时，将会发生什么情况呢？回车符和换行将被打印出来吗？不能。只有回车符将被打印出来。这就是说，第二行和后续各行将接在前一行后面打印出来。为了解决这个问题，每一次输入回车符时，都必须将回车符和换行都打印出。这个问题与我们在前面的例子(例 3-7)所讨论的问题非常类似。正如在前面这个例子那样，例 3-17 也应该使用一条条件调用指令。

例 3-16 用问号中断正在输入的信息

```
IANDT    LXIH    /把一个读/写存储器地址
         SIRING  /装入寄存器对 H；
         0       /该地址可以用来存储 ASCII 字符，
CHARIN,  CALL    /从电传打字机或 CRT
         TTYIN   /取一个字符，然后打印
         0       /该字符。
         CPI     /这个字符是问号吗？
         277
         JZ
         TERM    /是的，则保存一个结束符，
         0
```

```
        MOVMA /不是，则把这个字符保存在存储器里。
        INXH  /存储器指示器加 1。
        JMP   /然后，从电传打字机或 CRT
        CHARIN /取另一个字符
        0
TERM,   MVIM  /现在，把 000 存储在存储器，它位于这些 ASCII
        000   /字符之后，因为
              /问号已被打入。
          •
          •
          •
        ETC
TTYIN,  IN    /输入 UART 的状态位。
        001
        ANI   /只保存接收器的标识位。
        001   /如果 A＝001，则一只键已被按下。
        JZ    /如果 A＝000，则没有键被按下。
        TTYIN /因此，继续等到一只键被按下。
        0
        IN    /一只键已被按下了，所以把这只键的
        000   /ASCII 字符输入到 A 寄存器。
TTYOUT, MOVBA /把这个 ASCII 字符保存在 B 中。
TTYO,   IN    /输入 UART 的状态字。
        001
        ANI   /只保存发送器的标识位。
        004   /如果 A＝004，发送器(打印机)准备好。
        JZ    /如果 A＝000，发送器(打印机)不空。
        TTYO  /所以，继续等待发送器完成现行任务，
        0     /后，才能打印
        MOVAB /A 寄存器的内容。
```

```
        OUT     /把这个字符从 B 移到 A 后，
        000     /再把它输出给 UART。
        RET     /把这个字符仍旧留在 A，然后返回。
```

在例 3-17 中，并没有把 TTYIN 子程序列出来，因为该子程序与例 3-16 所列出的 TTYIN 子程序完全相同。

既然，这些 ASCII 字符已经存储在存储器了，那么，怎样才能把这些 ASCII 字符信息在电传打字机打印或显示在 CRT 的屏幕上呢？在例 3-14 中，把计数用数据装入在 C 寄存器里。这个数字用来确定在电传打字机键盘上能够打印入多少个字符并存储在 8080 的存储器里。当这个计数用的数字为零时，则不能再输入字符。这个计数用的数字也可以用于程序中，用来把存储在存储器里的字符打印在电传打字机上，或者显示在 CRT 上，如例 3-18 所示。

例 3-17 输入回车符时打印换行

```
IANDT,  LXISP   /装入堆栈指示器，
        STACK   /因为要调入子程序。
        0
        LXIH    /把读/写存储器地址，
        STRING/装入寄存器对 H，这些地
        0       /址用来存储 ASCII 字符串。
CHARIN, CALL    /从电传打字机或 CRT
        TTYIN   /取一个字符，然后打印它。
        0
        CPI     /这个字符是疑问号吗？
        277
        JZ
        TERM    /是的，保存一个结束符。
        0
```

```
        MOVMA/不是，则把它保存在存储器里。
        INXH   /存储器指示器加 1。
        CPI    /回车符打入了吗？
        215    /(八进制215＝十六进制 8D)。
        CZ     /打入了，然后打印换行。
        LFALSO/这样，就避免了出现行与行
        0      /打印重叠的现象。
        JMP    /从电传打字机或 CRT
        CHARIN/取另一个字符。
        0
TERM,   MVIM   /紧接在 ASCII 字符的后面，把 000 保存在存储器里
        000
          .    /因为疑问号已被打入。
          .
          .
        ETC
LFALSO, MVIA   /把换行的 ASCII 码
        212    /装入A寄存器
        CALL   /然后，调入电传打字机或 CRT
        TTYOUT/打印子程序
        0
        RET
```

把一个读/写存储器地址装入堆栈指示器（SP），并且将存储 ASCII 字符的存储地址装入寄存器对 H 之后，再把要打印的字符的数目装入C寄存器。程序执行控制在 PRNT1时，把一个字符从寄存器对 H 寻址的存储器单元送入 A 寄存器，(这一操作是由 8080 执行 MOVAM 指令来完成的)。然后 8080 调入 TTYOUT 子程序，把 A 寄存器的内容输出给 CRT 显示

器或电传打字机。当 8080 从 TTYOUT 子程序返回时，寄存器对H的内容加“1”，而C寄存器的内容减“1”。如果C寄存器的内容没有减到零时，则执行 JNZ-PRNTI 指令。当十六个字符都被打印后，C寄存器的内容减到零，则 8080 不再执行 JNZ-PRNTI 指令。这时，8080 执行该程序的其余指令。

例 3-18 打印存储在存储器的 ASCII 字符

```
PRINT, LXISP   /把一个读/写存储器地址
       STACK   /装入堆栈指示器。
       0
       LXIH    /把存储 ASCII 字符的存储器地址
       STRING  /装入寄存器对 H。
       0
       MVIC    /把要打印的字符的
       020     /数目（十进制 16）装入 C 寄存器
PRNTI, MOVAM   /从存储器送一个字符到 A 寄存器
       CALL    /在电传打字机打印或 CRT 显示器
       TTYOUT  /上显示这个字符。
       0
       INXH    /寄存器对 H 中存储的存储器地
       DCRC    /址加 1，然后 C 寄存器存储的字符
       JNZ     /的数目减 1。
       PRNT1   /如果 C 的内容不等于零，
       0       /则 8080 返回到 PRNT1。
         •     /当 16 个字符都被打印完毕后，
         •      8080执行存储在这儿的指令。
```

例 3-18 所给出的这个程序与用了一个字符计数器的输入程序（例 3-14）有着相同的局限性。如果 ASCII 字符信息的

长度增加或减少，那么我们必须改变C寄存器存储的计数数字，才能与改变了长短的ASCII字符的新信息相适应。我们可以编写出具有较高的灵活性的打印程序吗？是的，我们能够。请读者回忆一下，在例3-16和例3-17中，当键盘上的疑问号键被按下时，000（00）被保存在ASCII字符信息的末尾。我们可以利用这一点，编制新程序，如例3-19所示。

请读者记住，没有把换行的ASCII字符存储在读/写存储器，只把回车符的ASCII字符存储在存储器里面。因此回车字符被输出给电传打字机或CRT显示器之后，例3-19的输出程序必须产生这个换行ASCII字符。完成这些操作所需要的指令，如例3-19所示。

例 3-19 打印以000结尾的ASCII字符信息

```
PRINT,  LXISP    /把一个读/写存储器地址
        STACK    /装入堆栈指示器。
        0
        LXIH     /把存储ASCII字符的地址
        STRING   /装入寄存器对H。
        0
PRINTI, MOVAM    /从存储器取一个ASCII字符。
        CPI      /这个字符是这段信息的结束符吗？
        000
        JZ       /是的，则执行JZ指令
        END
        0
        CALL     /不是，它不是000(00)，
        TTYOUT   /所以把这个字符
        0        /在电传打字机或CRT显示器打印。
        CPI      /被打印的这个字符
```

```
        215         /是回车符吗?
        CI          /如果是回车符,
        LFALSO      /则打印换行。
        0
        INXH        /表指示针增1,
        JMP         /然后取另一个 ASCII 字符。
        PRNT1
        0
END       .
          .
          .
        ETC
TTYOUT,MOVBA    /把这个字符保存在 B 寄存器里。
TTYO    IN      /输入 UART 的状态字。
        001
        ANI     /只保存发送器的标识位。
        004     /如果 A=004, 发送器 (打印机) 准备好。
        JZ      /如果 A=000, 发送器 (打印机) 不空
        TTYO    /所以继续等待发送器 (打字机)
        0       /空, 然后才能打印A寄存器
        MOVAB   /的内容。
        OUT     /把这个字符从 B 传送到 A,
        000     /然后输出给 UART。
        RET     /把这个字符仍旧留在 A, 8080 返回。

LFALSO, MVIA    /把换行的 ASCII 代码
        212     /装入 A 寄存器
        CALL    /然后调入电传打字机
        TTYOUT/或 CRT 显示器的打印子程序。
        0
```

RET

8080 微型计算机什么时候开始执行从符号地址 END 开始的这段程序呢？在打印操作之前，从存储器读出 000（00）这个字符时，8080 将开始执行这段程序。读者记得，键盘上的疑问号键被按下时，000 已经被存储在存储器。000 是不打印的一个字符；一段 ASCII 字符信息被存储在存储器里，我们几乎可以用任何一个不打印的 ASCII 字符作为它的结束符。

如果读者希望在你的一段 ASCII 信息中用一个疑问号，那么会发生什么现象呢？如果存储在存储器里的一般 ASCII 信息有一个疑问号，那么，这个疑问号将被例 3-19 的程序打印出来。可是，没有容易的方法能把 ASCII 疑问号字符的值保存在存储器里。在例 3-16 的程序中，按下疑问号键时，000（00）被保存在存储器里，因为我们选择了这个疑问号键作为文件输入的结束符键。我们不是用 ASCII 疑问号字符作为软标志，而是可以用电传打字机产生的一个控制字符的 ASCII 值。CTRL 键维持在按下状态时，只要按下键盘上的一只键，便产生控制字符的代码。例如，当 CTRL 键维持在按下状态时，只要按下 C 键，便会产生一个 CTRL/C 控制字符。所以，这些控制代码之一（不是疑问号的代码），可以用来中止输入子程序。我们对例 3-16 这个程序必须做出的唯一改变，就是改变 CPI 指令的立即数字节。这就是把 277（疑问号的 ASCII 值）改变成 ASCII 码 203，作为 CTRL/C 值。由于这一修改，当 CTRL/C 代码被产生时，只中止输入子程序。在进行这一修改以后，任何一串 ASCII 信息中都可能含有疑问号，因为 CTRL/C 输入时，只有结束符 000 被存储在存储器里。

总之，在本书其余部分将要用到的电传打字机或 CRT 显

示器输入/输出（I/O）子程序如例 3-20 所示。虽然这些子程序与本章已经讨论过的那些子程序稍有不同，但是，在确切地确定它们如何操作方面，读者应该是不会有什么困难的。

例 3-20 电传打字机输入/输出的通用子程序，带有回送的 TTYIN 和 TTYOUT

```
TTYIN,    IN      /输入 UART 的状态字
          001
          ANI     /只保存接收器的标识位。
          001     /如果 A=001，则一只键已被按下。
          JZ      /如果 A=000，则没有键被按下。
          TTYIN   /所以，继续等待一只键被按下。
          0
          IN      /一只键已经被按下，所以把这只键
          000     /的 ASCII 字符输入到 A 寄存器。
          ANI     /把这个 ASCII 字符的奇偶校验位(D7)
          177     /去掉(八进制 177=十六进制 7F)。
TTYOUT,   MOVBA   /把这个字符保存在 B 寄存器里。
TTYO,     IN      /输入 UART 的状态字。
          001
          ANI     /只保存发送器的标识位。
          004     /如果 A=004，发送器(打印机)准备好。
          JZ      /如果 A=000，发送器(打印机)忙。
          TTYO    /所以，继续等待发送器(打印机)
          0       /完成现行操作，然后才能
          MOVAB/打印 A 寄存器的内容。
          OUT     /把这个字符从 B 寄存器送到 A
          000     /寄存器后，把它输出给 UART。
          RET     /此字符仍旧保留在 A 中，
                  /8080 返回。
```

```
    没有回送的 TTYI
TTYI    IN      /输入 UART 的状态。
        001
        ANI     /只保留接收器的状态。
        001
        JZ      /如果 A=001，一只键被按下。
        TTYI    /如果 A=000，没有键被按下。
        0       /如果没有键被按下，继续等待。
        IN      /一只键被按下，所以输入
        000     /这个 ASCII 字符。
        RET     /把它存入 A 寄存器而返回。
```

电传打字机或 CRT 显示器有三个基本的输入输出子程序。当 8080 微型计算机必须接收一个 ASCII 字符，并把这个字符打印在打印机上时，则调入 TTYIN 子程序。请读者注意 TTYIN 子程序中的那条 ANI 指令。我们从外部设备的键盘输入一个 ASCII 字符之后，这条 ANI 指令把存储在A寄存器的一个字节的 D_7 位(校验位)清除(屏蔽)。这就是说，如果TTYIN子程序被用来输入 ASCII 字符，那么，这个子程序可以与用来发送带有奇校验位，偶校验位或没有校验位的 ASCII 字符的电传打字机或 CRT 显示器一道使用。这就为程序软件提供了很大的灵活性和适用性。

当字符必须在 CRT 显示器显示或者在电传打字机上打印和穿孔时，8080 则调用 TTYOUT 子程序。当 8080 调用这个子程序时，必须把要打印的这个字符存储在A寄存器里。当 8080 微型计算机从 TTYIN 子程序或 TTYOUT 子程序返回时，它把这个 ASCII 字符保存在A寄存器和B寄存器里。

当 8080 必须要接收一个 ASCII 字符，但是这个字符并不

表 3-5　　ASCII字　符　集

字　符	八进制	十六进制	字　符	八进制	十六进制
RELL	207	87	?	277	BF
LF	212	8A	ⓐ	300	C0
CR	215	8D	A	301	C1
SPACE	240	A0	B	302	C2
!	241	A1	C	303	C3
〃	242	A2	D	304	C4
#	243	A3	E	305	C5
$	244	A4	F	306	C6
%	245	A5	G	307	C7
&	246	A6	H	310	C8
,	247	A7	I	311	C9
(	250	A8	J	312	CA
)	251	A9	K	313	CB
*	252	AA	L	314	CC
+	253	AB	M	315	CD
,	254	AC	N	316	CE
−	255	AD	O	317	CF
.	256	AE	P	320	D0
/	257	AF	Q	321	D1
0	260	B0	R	322	D2
1	261	B1	S	323	D3
2	262	B2	T	324	D4
3	263	B3	U	325	D5
4	264	B4	V	326	D6
5	265	B5	W	327	D7
6	266	B6	X	330	D8
7	267	B7	Y	331	D9
8	270	B8	Z	332	DA
9	271	B9	[	333	DB
:	272	BA	\	334	DC
;	273	BB	]	335	DD
<	274	BC	↑	336	DE
=	275	BD	←	337	DF
>	276	BE			

被打印(回送)时，8080 应该调用 TTYI 子程序。这个子程序并不改变这个字符的校验位，并且当 8080 微型计算机从这个子程序返回时，它只把这个字符存储在A寄存器里。

电传打字机程序和终端程序

电传打字机的测试程序

电传打字机或 CRT 的测试程序是可以编写得最简单和最有用的程序之一。在这个电传打字机测试程序中，所有的有效打印字符都可以输出给电传打字机的打印机或 CRT 屏幕。完成这一任务，我们需要知道电传打字机或 CRT 显示器能够接收、打印或显示的有关 ASCII 字符的一些知识。我们在表 3-5 中，归纳了 ASCII 字符集。请读者注意，大多数打印的ASCII 字符的值都在 240 (SPACE) 和 337 (←) 之间。只有回车符(CR)，换行(LF) 和 BELL (振铃) 例外；CR 的 ASCII 码值是 215 (八进制)，8D(十六进制)；LF 的 ASCII 码值是 212 (八进制)，8A(十六进制)；BELL 的 ASCII 码值是 207(八进制)，87(十六进制)。我们将要研究这几个字符。很可惜，没有容易的方法，可以打印 CR，LF，BELL 这三个字符，因此，我们只好使用例 3-21 所给出的程序。

例 3-21 怎样打印 CR，LF 和 BELL

```
START,  LXISP    /把读/写存储器的一个地址
        STACR    /装入堆栈指示器。
        0
TEST,   MVIA     /把这个 ASCII 字符(回车符)
```

```
215       /装入 A 寄存器。
CALL      /然后，在电传打字机或
TTYOUT /CRT 显示器上打印回车符
0
MVIA      /把这个 ASCII 字符(换行)
212       /装入 A 寄存器里。
CALL      /然后在电传打字机或 CRT
TTYOUT /显示器上打印该换行符。
0
MVIA      /把这个 ASCII 字符 BELL(振铃符)
207       /装入 A 寄存器
CALL      /然后在电传打字机或
TTYOUT/CRT 显示器上打印 BELL 这个振铃符
0
```

这段程序中所用的 TTYOUT 子程序，我们在本章前面已经讨论过了。把一个读/写存储器地址装入堆栈指示器之后，要打印的这个字符被送入A寄存器，然后才调用 TTYOUT 子程序。这个过程需要重复三次，CR，LF 和 BELL 这三个字符每个需要一次。

8080 微型计算机怎样把其余的 ASCII 字符都打印出呢？当然读者并不需要去编写一个有 64 条或 67 条 MVIA 指令的程序，使每条 MVIA 指令后面都跟上一条三字节的访问指令，用来调用 TTYOUT 子程序。因为正在被打印的全部 ASCII 字符的 ASCII 值都在 240(A 0)和 337(DF)之间，所以，只要产生全部打印字符，例 3-22 所列出的程序就可以用来测试电

传打字机或 CRT 显示器。

在打印 CKT(回车符)，LF(换行符)和 BELL(振铃符) 之后，8080 把 ASCII 值 240(A 0)装入A寄存器。 ASCII 值 240 (A 0)是 SPACE(空格符) 的代码。然后把A寄存器的 内 容与立即数字节 340(E 0)进行比较。 如果A 寄存器的内 容 是 340 (E 0)，那么，8080 微型计算机将执行 JZ 指令。该操 作 就 会把程序控制转到该程序的始点，即转到 TEST。如果A寄 存器的内容不等于 340(E 0)，那么，8080 调用 TTYOUT 子 程序，打印A寄存器的内容。 然后，8080 执行 INRA 指令， 将A寄存器的内容加 1、A 寄存器的初始值是 240(A 0)，所以，第一次它增量到241(A1)。什么时候A寄存器的内容才不增量 呢？当A寄存器存储的数值是 340(E 0) 时，才不对 它 进行增量操作。电传打字机的输出是怎样的格式呢？电传打字机的打印出格式如下：

```
!"#$%&'(  )*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]↑←
!"#$$%&'(  )*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]↑←
!"#$%&'(  )*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]↑←
```

这个程序的优点之一是：它可以用 来 提 高 或 调 整 驱 动 UART 接口的时钟频率，该接口被用来把电传打字机(或 CRT 显示器)和 8080 微型计算机连接起来进行通信。读者可以简单地把时钟频率提高或降低到错误开始在电传打字机上出现的这一点。一旦最高和最低的频率被确定之后，就可以把时钟调整到中心频率。虽然，并不是 8080 微型计算机都使用 UART 把诸如电传打字机或 CRT 显示器一类串行的外部设备和它 进行

通信，但是，许多8080微型计算机系统是用UART作为接口，完成与串行的外部设备进行通信任务的。

例 3-22 电传打字机或CRT测试程序

```
START,  LXISP    /把读/写存储器的一个地址
        STACK    /装入堆栈指示器。
        0
TEST,   MVIA     /把这个ASCII字符(回车符)
        215      /装入A寄存器。
        CALL     /然后在电传打字机或CRT上
        TTYOUT   /打印该回车符。
        0
        MVIA     /把这个ASCII字符(换行)
        212      /装入A寄存器。
        CALL     /然后在电传打字机或CRT上
        TTYOUT   /打印该换行符。
        0
        MVIA     /把这个ASCII字符(振铃)
        207      /装入A寄存器
        CALL     /然后电传打字机或CRT上
        TTYOUT   /响铃。
        0
        MVIA     /要打印的第一个字符是
        240      /空格(八进制240=十六进制A0)。
NXTCHR, CPI
        340      /最后这个字符已经被打印了吗?
        JZ
        TEST     /是，再开始这一过程。
        0
        CALL     /不是，打印这个ASCII字符。
```

```
TTYOUT
0
INRA        /要打印这个 ASCII
JMP         /字符的值加 1，
NXTCHR      /然后打印之。
0
```

纸带阅读机/穿孔机的测试程序

电传打字机上的纸带穿孔机操作不正常时，读者怎么知道呢？8080 微型计算机在给 300 英尺长的纸带穿孔时，需要检查它的错误。为了帮助读者测试纸带穿孔机，你可以使用例 3-23 所列出的程序。编写这个程序时假设，纸带穿孔机与电传打字机的打印机构通过机械连接方式连接的大多数情况下，纸带穿孔机和电传打字机都采用这种连接方式。这就是说，对读者而言，打字机和纸带穿孔机是相同的一种外部设备，因此也可以使用 TTYOUT 子程序，把A寄存器的内容在纸带上穿孔，如果被测试的穿孔机不是用机械方式与电传打字机的打印机连接，就必须给出一个子程序，用来把A寄存器的内容在纸带上穿孔。当 8080 从这个子程序返回时，A寄存器的内容保持不变。

例 3-23 的程序被启动时，首先将在纸带上穿上 100 个(十进制)空格符(八进制 000，十六进制 00)。这是因为把一个读/写存储器地址装入堆栈指示器 (SP) 后，8080 调用了 BLANK (空格)子程序的缘故。这个子程序把C寄存器用作为空格符计数器，该计数器初值为十进制数 100 (八进制 144，十六进制 64)。然后把A寄存器清除为 000 (00)，再把A寄存器的内容在纸带上穿孔，直到C寄存器的内容减到零时才停止。穿孔操

作停止时，8080 不执行 JNZ-BLANK 1 指令，所以，它把程序控制送回到主程序。请读者注意，这个 BLANK 子程序的 XRAA(A寄存器和A寄存器的异或操作)指令的用途。XRAA 指令用来把A寄存器的内容清零(八进制 000，十六进制 00)。

当程序控制返回到主程序后，8080 调入 TTYOUT 子程序，把A寄存器的内容在纸带上穿孔。A寄存器的内容仍旧是零。但是，8080 微型计算机从 TTYOUT 子程序返回时，执行 INRA 指令，将A寄存器的内容加 1，成为 001(01)。然后测试零标识位，看A寄存器的内容是否已经从 377 增量到 000 (FF-00)，如果A寄存器的内容不是零；则执行 JNZ-PUNCH 指令，把A寄存器的增量后的内容打印在纸带上。

最后，A寄存器的内容从 377 增量到 000(FF-00)时，8080 不再执行 JNZ-PUNCH 指令，而是调入 BLANK 子程序。这样，又会把 100 个(十进制数)空格符（从八进制 000 或十六进制 00 开始）在纸带上穿孔。当这个任务完成时，8080 微型计算机将停止操作。

例 3-23 纸带穿孔机的测试程序

```
PTEST,  LXISP     /把读/写存储器的一个
        STACK     /地址装入堆栈指示器。
        0
        CALL      /给 100 个空格(000)符穿孔。
        BLANK
        0
PONCH,  CALL      /将A寄存器的内容穿孔。
        TTYOUT    /(穿孔机与打印机进行机械连接)。
        0
        INRA      /把要穿孔的字符加 1 。
        JNZ       /是 000 吗？不是，给另一个字符穿孔。
```

```
        PUNCH
        0
        CALL        /是 000，001—377 个字符已穿
        BLANK       /孔完毕，所以再穿孔 100 个空格符。
        0
        HLT         /然后暂停。
BLANK,  MVIC        /把十进制数 100 装入 C 寄存器，
        144         /(八进制数 144＝十六进制数 64)
        XRAA        /执行这条指令，把 A 寄存器置零。
BLANKI, CALL        /把 A 寄存器的内容在
        TTYOUT      /电传打字机的穿孔机上穿孔。
        0
        DCRC        /字符计数器减 1。
        JNZ         /如果 C 的内容不等于零，
        BLANKI      /给另一个空格符穿孔。
        0
        RET         /在纸带上已经打印了 100 个空格符
                    /后，8080 返回。
```

如果正在被测试的电传打字机在每英寸纸带穿孔 10 字符，将会得到一条纸带，这条纸带的开始一段是10英寸长的空格符，中间是 25.6 英寸的一段纸带，000-377(00-FF)的全部字符，都穿在这段 25.6 英寸的纸带上；最后一段又是 10 英寸长的空格符。对于大多数程序员来说，要从这条纸带上找出任何穿孔错误(除了最简单的)也是十分困难的。为了对这条纸带上任何遗漏或不正确的字符进行测试，我们可以使用例 3-24 所列出的纸带阅读机程序。我们假设纸带阅读机正在正确地操作。

例 3-24 纸带阅读机的测试程序

```
READER,  LXISP      /把读/写存储器的
```

```
        STACK   /一个地址装入堆栈指示器。
        0
FRONT,  CALL    /读一个字符(纸带阅读机
        TTYI    /和键盘是同一种
        0       /输入/输出设备)
        CPI     /读了一个空格符吗?
        000
        JZ      /是的，读了一个空格符，
        FRONT   /忽略纸带的开始一段上所有的空格符。
        0
        MVIB    /它不是 000，把 B 寄存器置为
        001     /应该被读的第一个非零值
NEXT,   CMPB    /被读的这个字符与 B 的内容相等吗?
        JNZ     /不相等，则有错误。
        ERROR
        0
        CALL    /从纸带上取下一个字符。
        TTYI
        0
        INRB    /然后把这个比较字符加 1。
        JNZ     /B 寄存器已经从
        NEXT    /377 增加成为 000 了吗?
        0       /没有，测试下一个空格字符。
        HLT     /是的。纸带上的非零值的字符
                /已经测试完毕，所以暂停。
ERROR,  LXIH    /把读/写存储器的一个地址
        TEMPO   /装入寄存器对 H，从而
        0       /可以保存两个字符。
        MOVMB   /保存这个比较字符。
        INXH    /存储器地址增 1，
```

```
MOVMA    /然后保存这个被读的字符，
MVIA     /现在，在电传打字机上打印一个星号*
252      /表示出现了一个错误。
CALL
TTYOUT   /你应该检查读/写存储器
0        /的内容，弄清出现了什么错误。
HLT
```

当例 3-24 中的 LXISP 指令被执行时，首先要把一个 16 位的读/写存储器地址装入堆栈指示器（SP）。许多电传打字机的键盘和纸带阅读机似乎是同一个输入/输出设备。这就是说，在 8080 微型计算机的控制下，从 UART 输入一个数据字时，电传打字机不能确定这个字符到底是从键盘输入的，还是从纸带阅读机输入的。为此，使用 TTYI 子程序从纸带阅读机输入一个 8 位的字符。8080 读一个字符后，便把它与立即数字节000(00)进行比较。如果从纸带读出的这个数据字是零，那么，CPI 指令被执行后，零标识位将成为“真”。所以，8080 执行 JZ—FRONT 指令。这一段程序具有将纸带开始一段的空格符都读出和忽略它们的功能。从纸带上读第一个非“零”字符时，则不再执行这段 JZ—FRONT 程序。

第一个非零值的字符应该是 001 (01)。因此，当 8080 执行 MVIB 指令时，把 001 (01) 装入 B 寄存器。然后，8080 把 B 寄存器的内容与 A 寄存器的内容进行比较；A 寄存器存储的这个字符是刚刚从纸带上读入的。从纸带上读入的第一个非零值字符的值应该是 001 。如果 A 寄存器和 B 寄存器的内容不相等，则 8080 执行 JNZ—ERROR 段程序。

程序控制转到 ERROR 处时，8080 把一个读/写存储器 16 位地址装入寄存器对 H。然后，把 A 寄存器和 B 寄存器的内容

保存在后续存储器的存储单元中。然后把星号(*)的 ASCII 值装入A寄存器；接着 8080 调用 TTYOUT 子程序,在电传打印机上打印这个字符。这个任务被执行后，读者才能知道已经出现了错误。如果确实出现了错误，你将会发现 8080 正希望从纸带上读入的这个字符被存储在 LXIH 指令所指定的存储器单元；而实际上从纸带读入的字符被存储在较高一个地址上，即下一个连续存储器单元。

如果正确地读入了第一个非零值的字符，8080 将从纸带上读入第二个字符，然后把B寄存器的内容加 1 。并且重复上面所述的比较操作过程。只有B寄存器的内容从 377 增量成为 000(FF—00)时，8080 将才停止执行 JNZ—NEXT 这段程序。当这种情况出现时，8080 微型计算机将暂停。在最后一个非零值字符(377，FF)从纸带上读入后，如果穿孔机所穿出的纸带是正确的，并被阅读机正确地读出，8080 微型计算机将暂停。当然，如果纸带阅读机和穿孔机不是在正确地进行操作,那么，将会出现问题。为了使这两个程序或其中任何一个能够执行，所用的外部设备必须正确地工作，更精细的测试过程是可能有的，但是，现在我们的目的只是向读者介绍诸如电传打字机输入/输出设备的基础知识。

电　锁

这个程序使用了 9 个数字作为一个电锁的键。我们可以用任何 9 个数字的数；但是用社会保险帐号(SSAN)特别容易记住。我们能够用任何一种方法或外部设备来把这个键代码表示的 9 个数字送入 8080 微型计算机，以便锁定程序进行处理,这

些外部设备包括：键盘(由十个键构成)，指旋轮开关，磁标记阅读机，或电传打字机。为了使这个电锁程序尽可能简单，我们将用电传打字机来打入这个键代码所包含的 9 个数字。

这个键的 9 个数字要存储在连续的存储器单元。用电传打字机送入这个键代码后，存储在存储器里的数将与电传打字机输入的数进行比较。如果比较的结果不一致，就会“响起”警铃，因为有人在“撬”这个锁。如果正确地送入包含 9 个数字的键代码，那么，某处的门就会打开。我们用二个发光二极管来显示报警状态和“开门”状态。例 3—25 给出的程序是用来完成这些任务的。

8080 执行电锁程序(ELOCK)时，它首先把读/写存储器的一个地址装入堆栈指示器，因为要正确地调用子程序需要使用堆栈。这个存储单元已经分配给符号 STACK。然后，把数字计数数装入 B 寄存器，计数数是这个键代码所包含的数的位数。在实际应用中，这个数目可以增量或减量，但是我们已经决定把八进制数 011 (十六进制数 09，十进制数 9)装入 B 寄存器。这个数是社会保险帐号(SSAN)的位数。然后，把一个存储器单元的地址装入寄存器对 H，这个地址是用来存储键代码的第一个数字的地址。我们把这个存储器单元分配给符号地址 KEY。这个键代码所包含的其余各位数依次被存储在连续的存储器单元，从符号地址 KEY 开始，逐次向较高的单元存储。这个键代码包含的各个数字以 ASCII 字符值形式存入存储器里。

读者通过检查这些存储器单元，就能够确定电锁的键代码吗？这些存储器单元所存储的这个键代码如下：

0，1，2，3，4，5，6，7，8，

把这个键代码的第一个数字的地址装入寄存器对 H 以后，

8080 调用 TTYI子程序。当电传打字机的键盘上的一只键被按下后，8080 才从 TTYI 子程序返回。8080 从该子程序返回时，A寄存器所存储的 ASCII 值的 D_7 位置为零。这就从 ASCII 值中消去了校验位。

现在，A寄存器存储了这个键盘字符，把这个字符的 ASCII 值与寄存器对H所寻址的存储器单元的内容进行比较。寄存器对H寻址的存储器单元存储了键代码的第一位数字。所以，被输入的第一只键代码与八进制数 060 （十六进制数 30）进行比较。如果这两个值不相等。则 8080 从JNZ 转移到 PICK，执行指令。8080 的程序控制到达 PICK 时，A寄存器的最低有效值(LSB) 置逻辑 1，然后，把输出送给第 157（6 F）号外部设备。可以用这个逻辑电平锁定一个“门”，点亮 LED显示器或者报警。所以出现这种情况，是因为所按下的电传打字机键盘上的一只键与按顺序存入存储器的相应键代码不一致。

例 3-25 电锁程序

```
ELOCK,  LXISP    /使堆栈指示器置 1
        STACK
        0
        MVIB     /把 B 寄存器用作为位数计数器
        011      /(十进制数 9)
        LXIH     /寄存器对 H 指示到
        KEY      /被存储在存储器的“键”。
        0
NEXTIN, CALL     /从电传打字机取一个字符
        TTYI
        0
        ANI      /去掉这个 ASCII 字符的
        177      /校验位。
```

```
        CMPM    /把这个字符与存储器
        JNZ     /的内容比较，不一致，
        PICK    /则有人"撬"这个锁!
        0
        INXH    /如果与第一个数字是一致的，则寄
        DCRB    /存器对 H 指示到下一个数字。
                /位数计数器减 1 。
        JNZ     /九位数都输入了吗?
        NEXTIN  /没有，取另一个键代码。
        0
OPENIT, MVIA    /九位数字都被输入，并且与存入
        001     /存储器的键代码都一致，
        OUT     /因此，把"门"打开
        156
        HLT
PICK,   MVIA    /有人撬这个锁，于是
        001     /报警。
        OUT
        157
        HLT
TTYI,   IN      /输入 UART 的状态字。
        001
        ANI     /只保存接收器的状态。
        001
        JZ      /如果 A 的内容＝001，一只键被按下，
        TTYI    /如果 A 的内容＝000，没有键被按下。
        0       /如果没有键被按下，则继续等待。
        IN      /一个键被按下，所以输入
        000     /这个字符的 ASCII 值。
        RET     /8080 把该 ASCII 值留在
```

```
                    /A寄存器而返回
KEY,    060         /这只键所含的九位数字按
        061         /适当的顺序存储在此处。
        062
        063
        064
        065
        066
        067
        070
```

如果A寄存器存储的键代码及其相应的存储器单元的内容是一致的，8080则不执行JNZ-PICK指令。相反8080执行INXH指令，使寄存器对H的存储器地址增1；8080执行DCRB指令，使位数计数器(B寄存器)减1。如果B寄存器的内容减1后，并不到0值时，则重复执行上面的指令序列。只有当B寄存器的内容最后被减到零值时，8080才不执行JNZ NEXTIN指令。这种情况出现时，电锁键所包含的九位数字已经全部正确地送入了。所以，A寄存器的最低有效位(LSB)置逻辑1，然后把输出送给输出端口156(6E)。我们可以使用这个逻辑电平来把门打开，关断报警系统，或者点亮发光二极管(LED)显示器。然后8080微型计算机回到暂停状态。

这个程序所带来的问题之一是：当键代码被送入8080微型计算机时，如果出现了错误，我们则毫无办法把它改正。即使你知道正确的键代码，也可能在按电传打字机的键时搞错。所以，如果有一只应急暂停键使你能再启动程序，则这只键是很有用处的。同时，给你试三四次，以便打入正确的键码组合，这对于8080微型计算机来说是有利的。这样做尽管会增

加“撬锁”的可能性，但是，这种可能性仍然是很少的，即只有2.5×10^{7}分之一。例 3-26 所列的程序就具有这两个特点。

读者从例 3-26 可以看到，修改这个程序是很简单的。我们已经使用 101(41)这个数作为 CPI 指令的立即数字节，从而把电传打字机键盘的A键分配给应急暂停操作用。101(41) 这个数便是 ASCII“A”的值。当 8080 执行例 3-26 这个程序时，把读/写存储器的一个地址装入堆栈指示器(SP)，然后，把键代码可以打入的次数(本程序例为四次)的计数数装入C寄存器。该程序的 PICK 程序段，也已经被修改过了，所以，只有在C寄存器的计数数值被减到零值后，才响起报警信号。

请读者注意，应急暂停键(A)可按照需要经常使用。但是，这个程序仍然只能为你正确地输入键代码提供四次按键的尝试。当键A被按下时，8080 返回到 RSTRT。在这里，8080 把位数计数数字装入B寄存器，这就是说，读者可以在键盘上输入另一只键代码。

例 3-25 改进的电锁程序

```
ELOCK,    LXISP    /堆栈指示器置位。
          STACK
          0
          MVIC     /把报警之前尝试的
          004      /次数数据装入C寄存器。
RSTRT,    MVIB     /用B作为位数计数器
          011      /(十进制数 9 )
          LXIH     /寄存器对H指示到被
          KEY      /存储在存储器的“键”
          0
NEXTIN,   CALL     /从电传打字机
          TTYI     /取一个字符
```

```
          0
          ANI       /去掉校验位。
          177
          CPI
          101       /应急暂停键被按下了吗?
          JZ        /已经被按下,
          RSTRT     /再启动这个子程序。
          0
          CMPM      /把它与存储器单元的内容比较。
          JNZ       /结果不一致,则出了错误,
          PICK      /所以,判断已经出了多少错误。
          0
          INXH      /如果第一位数字是一致的,寄存
                    /器对H现在指示到下一位数,
          DCRB      /位数计数器(B寄存器)减"1"。
          JNZ       /九位数字都输入完了吗?
          NEXTIN    /没有,取另一个键代码。
          0
OPENIT,   MVIA      /九位数字都被输入了,并且与这个
          001       /"键"一致,然后把门打开。
          OUT
          156
          HLT
PICK,     DCRC      /已经试了四次吗?
          JNZ       /没有,再试。
          RSTRT
          0
          MVIA      /试了四次,仍旧是错误的,
          001       /然后报警。
          OUT
```

```
          157
          HLT
TTYI,     IN        /输入 UART 的状态字。
          001
          ANI       /只保存接收器的标识位。
          001
          JNZ       /如果 A＝001，一只键被按下，
          TTYI      /如果 A＝000，没有键被按下。
          0         /如果没有键被按下，则继续等待。
          IN        /一只键被按下了，所以
          000       /输入这个字符的 ASCII 代码。
          RET       /把它留在A寄存器，
                    /8080 返回。
KEY       060
          061       /这是一个包含 9 个数字的键，
          062       /这九个数字存储在
          063       /连续的存储器单元。
          064
          065
          066
          067
          070
```

如果读者正在输入适当的键代码，随后，你忘记了已经输入了哪些数，那么只有在这种情况下，才能使用应急暂停键。如果你试图输入代码 123456789，8080 微型计算机把这些数的任何一个与键代码的第一位数字 0 比较，都不能得到一致的结果。因此，即使只输入一个 9 位数，也会把四次尝试权用完。这是因为这些数被输入时，8080 微型计算机要把它们进行比

较。把 1 这个数打入后，将 1 与 0 进行比较，因为它们不相等，所以，8080 返回到这个程序的 PICK 程序段。在 PICK 处，将尝试次数计数器(C 寄存器)的内容减 1。因为现在 C 寄存器的内容是 003 (03)，所以 8080 返回到 RSTRT。程序执行到达 RSTRT 时，B 寄存器的位计数数再被置预置值，然后 8080 第二次调入 TTYI 子程序。这时，如果读者错误地把键代码 123456789 的数字 2 打入机器，也要把它与存储在存储器的键代码的第一个数 0 进行比较。

下一次通过循环时，8080 微型计算机应该把打入的数字 3 和 4 与这个键代码的第一个数 0 进行比较。由于比较的结果不会一致，所以计数器(C 寄存器)被减到 0，并且响起警报。

从电传打字机输入这些 ASCII 字符为什么要使用 TTYI 子程序，而不使用 TTYIN 子程序呢？这个问题读者知道吗？这两个子程序有什么区别呢？使用 TTYIN 子程序，不仅让你从电传打字机键盘上输入字符，而且也把这些字符打印出来。TTYI 子程序只让你从电传打字机的键盘输入字符，这些字符并不被打印。因此，你输入这只键代码时，如果有人正从你的肩上面看过去，那么，他们记住这个键代码则是困难的。如果这个键代码被打印了，那么，谁都可以更为容易地记住这个键代码。

小　结

总之，在本章和前面两章，我们已经讨论了下列指令：

1. 63 条 MOV 指令。
2. 8 条 MVI 指令。

3．8 条加 1 (INR)指令和 8 条减 1 (DCR)指令。

4．64 条算术和逻辑指令：ADD，ADC，SUB，SBB，ANA，XRA，ORA，和 CMP。

5．8 条立即数算术和逻辑指令：ADI，ACI，SUI，SBI，ANI，XRI，ORI，和 CPI。

6．两条输入/输出(I/O)指令：IN 和 OUT

7．两条停机指令：HLT 和 NOP。

8．四条寄存器对装入指令：LXIB，LXID，LXIH，和 LXISP。

9．四条寄存器对加 1 指令(INX)和四条减 1 指令(DCX)。

10．九条转移指令，其中一条无条件转移指令和八条条件转移指令。

11．九条调用指令，其中包括一条无条件调用指令，八条条件调用指令。

12．九条返回指令，其中包括一条无条件返回指令，八条条件返回指令。

就本书所讨论的范围来说，只要读者遵循本书所给出的程序实例，你会得心应手地运用这二百零二条指令。甚至你能用这些基本指令编写出功能很强的程序来。在下一章，我们要讨论 8080 的较高级的指令。

第四章　8080/8085的高级指令

我们人为地把8080的指令分成了基本指令和高级指令两个部分。第一代8位微处理器的代表Intel可以执行的大部分指令和第二代微处理器8080的一些指令，在基本指令这一章(第二章)已经讨论过了，这些基本指令也是最常用的指令。一个程序的60%～80%是由它们所构成的，程序的其余指令是由高级指令组成的，这些指令正是本章描述的内容。

寄存器对的各种操作

为了把数据传送到存储器单元，或者从存储器单元送出来，我们使用了寄存器对H来提供这个存储器单元的地址。你也已经看到：正如寄存器对H(H和L寄存器)一样，B寄存器和C寄存器，D寄存器和E寄存器也能作为寄存器对使用。可以与这些寄存器对协同使用的指令有INX，DCX和LXI这三种指令，这些指令我们已经在前面讨论过了。8080微处理器也能使用寄存器对B、寄存器对D来对存储器寻址，但是，用于这两个寄存器对的数据传送指令的功能不如用于寄存器对H的数据传送指令的强。

如果使用寄存器对H，8080微处理器可以把存储器单元的内容和A寄存器的内容进行相加，相减，“与”操作，“或”操作，异操作，或比较操作；也可以把存储器单元的内容加1或减1。

8080 也可以把寄存器对H的内容作为数据传送类（MOV)指令的数据源地址，或数据的目的地址。**把寄存器对B或寄存器对D的内容作为存储器单元的地址使用时，8080只能使数据 在A寄存器和某个存储器单元之间传送**。我们把这些数据传送指令列入表 4-1。

表 4-1　寄存器对 B 和寄存器对 D 的数据移动指令

LDAXB	把寄存器对 B 寻址的存储器单元的内容装入 A 寄存器。
LDAXD	把寄存器对 D 寻址的存储器单元的内容装入 A 寄存器。
STAXB	把 A 寄存器的内容存入寄存器对 B 寻址的存储器单元。
STAXD	把 A 寄存器的内容存入寄存器对 D 寻址的存储器单元。

虽然，8080 并没有诸如 LDAXH 和 STAXH 这样的指令，但是，有没有一些其它指令来完成这两条指令的功能呢？回答是肯定的。MOVAM 和 MOVMA 能够实现这一功能，现在，读者已经看到了下述指令，它们是用来操作寄存器对 B 和寄存器对 D 的：

LXIB	LXID
INXB	INXD
DCXB	DCXD
STAXB	STAXD
LDAXB	LDAXD

这些指令的用途之一是将数据块从一个存储器区传送到另一个存储器区。为了完成这一任务，一定要给出两个地址：一个是数据的源地址，另一个是数据的目的地址。读者还必须有某种方法，用来表示数据块已经传送了。这一功能是由字节计数器来完成的。把连续存储器单元所存储的一个数据块送到存

储器的另外一部分的程序，如例 4-1 所示。

在例 4-1 中，你能够确定正在被传送的数据来自何处和送到什么地址单元吗？第一个数据字节是从第 021100 (1140) 号存储器单元送到 021200(1180)号地址单元。被传送的最后一个数据字节的地址是什么呢？要把最后这个数据字节从第 021022 (1112)地址单元送到第 021222(1192)号地址单元。我们可以使用这个程序传送 521 个数据字节或 623 个数据字节吗？不能。这个程序能传送的数据字节最多为 256 个。这是为什么呢？因为我们用C寄存器来存储字节的数目。为了传送 100 个数据字节或 200 个数据字节，必须把这两个数的等效的二进制数(十进制数 100 和 200 分别等于二进制 数 01100100 和 11001000) 装入C寄存器。但是，为了传送 256 个数据字节，必须把000(00)装入C寄存器。传送第一个数据字节后， 8080 执 行 DCRC 指令时，将C寄存器的内容从000(00)减量成 377(EF)。 因 为该结果不是零，所以， 8080 微型计算机将返回到“TRNSFR”。然后 8080 继续循环执行“TRNSFR”， 一直到另外的 255 个 数 据字节被传送后才停止。8080 在停止执行循环时，C寄存器的内容将最后减为零。

为了克服例 4-1 这个程序只能传送 256 个字节的局限性，读者可以在这个程序的基础上增添几条指令，使它变成为例4-2那样的程序，则它更有通用性。

例 4-1 传送数据块的程序

```
MOVE, LXIH/把这个数据源
      100   /地址装入 H 寄存器对。
      021
      LXID/把这个数据
      200   /的目的地址装入 D 寄存器对。
```

```
        021
        MVIC/装入要被传送的
        023/数据字节的数目。
TRNSFR, MOVAM/利用寄存器对 H，取一个数据字节。
        STAXD/利用寄存器对 D，存储一个数据字节。
        INXH/源地址加"1"。
        INXD/目的地址加"1"。
        DCRC/字节计数器减 1。
        JNZ/19 个(十进制)数据字节都被传送完了吗?
        TRNSFR/没有，则执行这条转移指令。
     0
     · /  是，都送走了。继续向前执行程序。
     •
```

如果使用寄存器对 B 作为字节计数器，执行例 4-2 这个程序就能把若干个数据字节从一个存储区移到另一个存储区，最多可传送 65536 个字节，为了传送这么多数据字节，应该把000000(0000)装入寄存器对 B，循环第一次，就能把寄存器对 B 从 000000 减到 377377(0000—FFFF)。因为把寄存器对 B 作为字节计数器，所以使用 DCXB 指令，把字节计数器的内容减 1。这条指令并不影响任何标识位，因此，8080 必须执行 MOVAB 和 ORAC 这两条指令，以便确定什么时候寄存器对 B 的内容是 000000。当寄存器对 B 的内容是 000000(0000)时，8080 已经传送完了所需要的数据字节。在例 4-2 这个程序里，数据字节的数目是 012023(OA13)，即十进制数 2579。因此，8080 只有传送了 2579 个数据字节后，它才停止执行 TRNSFR 这个循环。

假设读者已经把一个数据组存储在 005000～005200 (0500～0580)这些存储器单元，而读者要把这些数据移到 005100～

005300(054～05CO)。这些存储器单元，为了传送这个数组的数据，你可以使用例 4-1 或例 4-2 这两个程序吗？不能，绝对不能！因为两个数组是重叠的，问题就在这里。读者要把005000(0500)存储器单元的内容送到 005100 (0540) 存储器单元，可是，此存储器单元还有要传送的数据。换句话说，把数据从 005000(0500)地址单元，移到 005100(0540) 地址单元，需要把被传送的数据送回到原存放数组的存储器单元，可是，这个单元的内容还没有被送走，在这个被传送的数据写入另一个存储器单元之前，你大概不容易把这个数值保存在 005100 (0540)这个单元。这个问题，如图 4-1 所示。

例 4-2 改进的数据块传送程序

```
MOVE, LXIH    /把数据源的始地址
      100     /装入寄存器对 H。
      021
      LXID    /把数据的目的始地址
      100     /装入寄存器对 D。
      051
      LXIB    /装入要移送的字节
      023     /的数目。
      012
TRNSFR, MOVAM/利用寄存器对 H 取一个数据字节。
      STAXD   /利用寄存器对 D 存储一个数据字，
      INXH    /源地址加 1。
      INXD    /目的地址加 1。
      DCXB    /字节计数器减 1。
      MOVAB   /计数器的内容是零吗？
      ORAC
      JNZ
```

```
TRNSFR      /不是，则返回到 TRNSFR。
0
•           /是，继续往下执行程序。
•
•
```

读者能够想一个简单的方法来解决这个问题吗？传送这些数据的最简单的方法是，首先把原来的那个数组的最高地址（005200，0580）存储的数据送到 005300(05CO)，这个地址单元是新的数组的最高地址。这个操作完成之后，8080应该从这个最高地址起，继续向下建立这个新的数组。这就避免了把来自原数组的数据又写入到组列中去。例 4-3 给出的这个程序就是用来完成这个任务的。它把第一个数据字节从 005200 这个地址单元送到 005300(0580～05C 0)，把最后一个数据字节从 005000 移到 005100(0500～0540)。

当这些数据组重叠时，你怎么才能把一个数组从较高的地址移到较低的地址呢？假设一个数组数据存储在 030100～030250(1840～18A8)这些存储器单元。如果你必须把这个数组的数据向地址较低的存储单元送到 030000～030150(1800～1868)的地址单元，那么；开始和最后被传送的地址应该是什么呢？这个问题正好与我们刚在讨论的那个问题相反。如果把第一个数据字节从 030100 这个地址送到 030000(1840～1800)，这个阵列的数据一个也不会被丢失。如果，你把 030250 这个存储器单元的内容送到 030150(18A8～1868)，那么数据将会丢失，因为 8080 正在把数据往回传到原来的数组中。正如读者可以看到的，确切地知道正在被传送的数组是从哪里送来的，送到何处去，这是十分重要的。只有在这之后，你才能确定所采用的合适的指令序列。根据这两个例子，**如果你把一个数据组数据**

从较低的存储器地址移到较高的存储器地址，应该首先传送存储在最高地址的这个数据。如果数组是重叠的，也不会丢失任何数据。**如果读者正在把一个数组数据从较高的地址单元送到较低的地址单元，那么，首先传送被存储在最低地址单元的数据字节。**即使有数据字节重叠，也不会丢失任何数据。这些程序例子(例 4-1～例4-3)用来阐明“STAX”类型指令的作用。

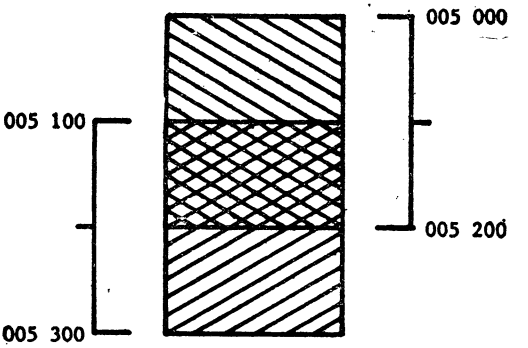


图 4-1 传送叠加数组数据的问题

例 4-3 把重叠的数组数据从上向下传送

```
MOVE, LXZH    /装入这个数据组的始
      200     /地址。
      005
      LXID    /装入这个数据组
      300     /的目的地址。
      005
      LXIB    /装入要被传送
      200     /的字节数。
      000
DOWN, MOVAM/从原数组取一个数据字节。
```

```
STAXD  /把这个数据字节存入新数组。
DCXH   /源地址减 1 。
DCXD   /目的地址减 1 。
DCXB   /字节计数器减 1 。
MOVAB  /字节计数器的内容已经是 0 吗?
ORAC
JNZ
DOWN   /不是，则返回。
0
  .    /是，继续执行下面的程序
  .
  .
```

堆 栈 指 示 器

还有一个寄存器叫堆栈指示器(SP)，前面，我们只简要地讨论了一下。这是一个 16 位的寄存器,它存储一个 16 位地址，这个地址指示到存储子程序的返回地址的读/写存储器段。为了把一个 16 位地址装入堆栈指示器，8080 应该执行一条 LXI SP 指令，该指令有三个字节，指令码用八进制数 061（十六进制数 31）表示。这条指令的第二字节和第三字节包含读/写存储器单元的地址。在功能上，这条指令与其它 LXI 指令相同。你也可以让 8080 执行 INXSP 指令（操作码是八进制数 063，或十六进制数 33），或者让 8080 执行 DCXSP 指令（操作码是八进制数 073，或十六进制数 3 B），使堆栈指示器加 1 ，或者减 1 。正如执行其它寄存器对的加 1 和减 1 指令一样,当 8080 执行 INXSP 和 DCXSP 这两条指令的任何一条时，都不会影

响到标识位。堆栈指示器象寄存器对B和寄存器对D一样，是一个专用性很强的寄存器。与堆栈指示器操作有关的指令并没有LDAX或STAX指令。到现在为止，把我们已经讨论过的寄存器对指令归纳起来，如表4-2所示。

表 4-2　　寄存器对指令一览表

寄存器对指令	B	D	H	SP
LXI	是	是	是	是
INX	是	是	是	是
DCX	是	是	是	是
LDAX	是	是	是*	不是
STAX	是	是	是*	不是

* 表示有等效的指令；LDAXH=MOVAM，STAXH=MOVMA

就堆栈指示器和堆栈指示器指令而言，我们还没有讨论返回地址存入堆栈的方法。我们只知道：8080执行条件调用指令或无条件调用指令时，必须把一个返回地址存入读/写存储器部分，即堆栈，这是由程序设计员事先把它划分为堆栈区的。8080执行LXISP指令时，确定这个堆栈区。当执行一条返回指令时，从堆栈弹出一个返回地址；8080执行这条三字节的调用指令之后，继续开始执行该程序的其余部分的指令。

DAD类指令

DAD类（双寄存器对加）指令，用来把指定的寄存器对B，D，H或者堆栈指示器的内容加到寄存器对H的内容之上。

16 位数加的结果自动地被存储在寄存器对H里。这类 DAD 指令都是单字节指令，表 4-3 列出了它们的操作码。

表 4-3　　DAD 类 指 令

八 进 制		十六进制
011	DADB	09
031	DADD	19
051	DADH	29
071	DADSP	39

在基本指令这一章（第三章），我们讨论了单精度（八位）加和带进位加指令。8080 可以用这些指令来实现下列操作：(1) 加任何一个通用寄存器的内容；(2) 加寄存器对H寻址的存储器单元的内容；(3) 把立即数字节加到A寄存器的内容上。但是，你有时可能需要操作大于 8 位（十进制数 0～255）的数据字。我们常常使用双精度的数，即 16 位。因为一个 16 位的数据字可以表示 0～65，536（十进制）之间的任何一个数。如果读者希望把两个 16 位数相加，你可以应用例 4-4 所列出的程序。

例 4-4　两个 16 位数相加的程序

```
ADD 16,  MOVAC    /取一个 LSBY。
         ADDL     /加另一个 LSBY。
         MOVLA    /LSBY 相加的结果保存在 L 寄存器。
         MOVAB    /取一个 MSBY。
         ADCH     /加另一个 MSBY。
         MOVHA    /把结果的 MSBY 保存在 H 寄存器里。
```

这个程序由六条指令组成，用来完成将寄存器对B的内容

加到寄存器对H的内容上的操作。相加的结果是16位，把它存在寄存器对H里。请注意在例4-4中ADCH指令的使用。这条指令把H寄存器的内容和来自另一个加操作的结果保存在A寄存器里。虽然，我们在下一章要讨论算术程序，但是，你认为目前这个程序可以简化吗？这个程序可以简化，使用一条DADB指令就行了。

这条DADB指令象其它DAD类指令一样，当两个LSBY相加时，所产生的进位要加到高位字节上。如果两个MSBY相加产生了进位，那么进位标识位将被置逻辑1。在基本指令这一章(第三章)，讨论了两种加法指令，即ADD和ADC，但是，双精度（16位）加指令只有一种，它就是DAD指令。如果一定要把大于16位的数相加，那么，这种指令不可能实现。这是因为8080执行一条DAD类指令，完成了第一个16位数加操作以后，并没有16位带进位加的指令可供执行。因此，在这个程序中，必须增添一些指令，用来把第一个16位数的加操作所产生的进位加到将要进行相加的一组16位数上。

这条DADH指令特别有趣，假设下面这个16位字存储在寄存器对H里面：

H寄存器	L寄存器
00000000	00000010

执行一条DADH指令后，寄存器对H的内容应该是什么呢？H寄存器的内容将仍旧是00000000，但是L寄存器的内容应该是00000010，如果另一条DADH指令被执行，H寄存器的内容仍旧不变，但是L寄存器的内容将是00001000。这是怎么回事呢？起初，L寄存器的内容是002(02)。8080执行第一条DADH指令后，L寄存器的内容是004(04)。第三条DADH指令被执行之后，L寄存器所包含的内容是010(08)。根据这

条指令的执行结果，读者应该可以得出结论：每当执行一条DADH指令后，寄存器对H的内容将增加一倍。请读者记住，当你把一个数自加时，你应使该数加倍。你也可以认为是这个数乘以2。如果把000001(0001)装入寄存器对H，那么，执行若干条DADH指令后，寄存器对H的内容会发生什么变化呢？让我们来考虑表4-4所示出的结果。

根据表4-4所示的结果，读者不仅可以把这条DADH指令看作为双精度指令，也可以看作为逻辑左移操作指令，在逻辑左移操作中最高有效位（MSB）被丢失，或者移入进位。这种逻辑左移操作不是循环移位操作，因为并不把数据循环移入16位字的最低有效位。8080微型计算机执行例4-5的程序后，寄存器对H的内容应该是什么呢？

表4-4 执行若干条DADH指令后寄存器对H的内容

H寄存器	L寄存器
00000000	00000001
00000000	00000010
00000000	00000100
00000000	00001000
00000000	00010000
00000000	00100000
00000000	01000000
00000000	10000000
00000001	00000000
00000010	00000000
00000100	00000000
00001000	00000000

8080把例4-5这个程序循环执行共20次（十进制）。第

十五次执行这条DADH指令后，寄存器对H的内容如下所示：

H寄存器	L寄存器
10000000	00000000

8080第十六次执行这条DADH指令后，H寄存器和L寄存器将存储000(00)。但是进位将是逻辑1。这两个寄存器将为000(00)，以供执行该程序的其余部分的指令用。请读者记住，L寄存器的八位产生的一位进位存储在H寄存器。

例 4-5 执行二十次DADH指令的程序

```
BY 2,     LXIH     /把 000001 装入寄存器
          001      /对H，(八进制数 000001，十六进制数 0001，
          000      /二进制数 0000000000000001)
          MVIB     /把计数数字装入 B 寄存器
          024      /(十进制数 20，八进制数 24，十六
                   /进制数 14)
DOUBLE, DADH       /把寄存器对H的内容相加，并将
                   /结果存入H、L。
          PCRB     /计数器的内容减 1。
          JNZ      /如果 B 的内容不是零，则执行
          DOUBLE   /JNZ—DOUBLE。
          0
          HLT      /当 B 的内容＝000 时，
                   /8080 执行暂停操作。
```

执行其余的n次循环后，H寄存器，L寄存器和进位的内容如表4—5所示。

DADSP指令是一条有趣的指令，因为这条指令给你提供这种能力：确定堆栈指示器（寄存器）现行存储的一个地址。在多数程序中，这条指令操作并不是重要的，因为又有一些特殊事情才使用堆栈。但是调试，系统监控和操作系统等程序经

表 4-5　　执行 JNZ-DOUBLE 的循环后H寄存器，L 寄存器和进位的内容

进　位	H	L
0	00001000	00000000
0	00010000	00000000
0	00100000	00000000
0	01000000	00000000
0	10000000	00000000
1	00000000	00000000
0	00000000	00000000
0	00000000	00000000
0	00000000	00000000
0	00000000	00000000

常把堆栈指示器现在指示到的存储器单元的内容告诉用户：它是否已经超越了程序的范围，或者是否已经超越了数据存储区，或者它是否已经超过了一定大小。为了确定堆栈指示器的现行内容，你可以给 8080 编程序，这个程序如例 4-6 所示。

例 4-6　确定堆栈指示器，寄存器（SP）里存储的地址

```
LXIH   /把八进制数 000000
000    /(十六进制数 0000)
000    /装入寄存器对H。
DADSP  /把堆栈指示器的内容加到
·      /寄存器对H的内容上，
·      /然后把结果存入寄存器对H。
·
```

8080 按照例 4-6 所示的指令序列执行程序以后，寄存器

对H的内容应该将是一个地址；这个地址就是现在存储在堆栈指示器里的内容。这个指令序列不会改变堆栈指示器的内容。同8080的其它许多程序指令一样，一条指令执行之后，寄存器的内容并未改变，它只是用作为源寄存器罢了。8080执行例4-6的程序指令之后，它还可以使用堆栈。但是，H寄存器的内容将是堆栈指示器的现行地址。当8080执行例4-7所列出的指令序列以后，寄存器对H的内容将是什么呢？

这些指令被执行之后，寄存器对H的内容将是0 02303（02C3）。最初，堆栈指示器置于0 02300（02C0）。8080三次执行LNXSP指令以后，堆栈指示器所存储的地址是002303（02C3）。8080执行DCXSP指令，使堆栈指示器的内容减1，即从002303（02C3）变为002302（02C2）。然后，把000001（0001）装入寄存器对H；8080再执行DADSP指令，把堆栈指示器的内容加到寄存器对H的内容上。因此，002302＋000001得到002303（02C2＋0001＝02C3），这个数保存在寄存器对H。

例 4-7 使堆栈指示器加1和减1

```
LXISP   /装入堆栈指示器寄存器。
300
002
INXSP   /堆栈指示器的内容加1。
INXSP   /堆栈指示器的内容加1。
INXSP   /堆栈指示器的内容加1。
DCXSP   /堆栈指示器的内容减1。
LXIH    /把001装入L寄存器，
001     /把000装入H寄存器。
000
DADSP   /把堆栈指示器的内容
```

```
•       /加到寄存器对H的内容上，
•       /然后把结果保存在寄存器对H。
•
```

直接装入指令和存储指令

在上一个程序例子（例 4-7），读者看到了8080可以把堆栈指示器的内容加到寄存器对H的内容上。但是，假设读者已经把一个16位的数存储在寄存器对H里，那么，8080通过执行下面的指令序列，仍然能确定这个堆栈指示器地址吗？

```
LXIH    /把八进制数
000     /000000(十六进制数 0000)
000     /装入寄存器对H。
DADSP   /把堆栈指示器的内容，
•       /加到寄存器对H的内容之上，
•       /并且把结果保存在寄存器对H里。
•
```

如果在执行上面这些指令之前，把寄存器对H的内容保存起来，那么，就可以确定这个堆栈指示器地址。读者可以把寄存器对H的内容保存在另一个寄存器对里，但是，这个寄存器对常常是不可使用的。在把堆栈指示器的内容加到寄存器对H的内容之前，可以把寄存器对H的内容保存在读/写存储器。那么，读者怎样才能实现这个操作呢？现在让我们来研究例4-8的指令。

例 4-8 把寄存器对H的内容保存在读/写存储器里

```
MOVEL   /把低位地址保存在E寄存器。
MOVDH   /把高位地址保存在D寄存器。
```

```
LXIH    /把地址 060015 装入寄存器对H，
015     /八进制数 060015＝十六进制数 300D。
060
MOVME   /把低位地址存入 060015(十六进制 300D)，
INXH    /然后把这个地址加 1。
MDVMD   /把高位地址存入 060016 (十六进制数 300E)。
LXIH    /现在，把 000000 装入寄存器对H，
000     /(八进制数 000000＝十六进制数 0000)。
000
DADSP   /把堆栈指示器的内容加到
·       /寄存器对H的内容上，然后把结
·       /果保存在寄存器对H。
·
```

把寄存器对H的内容存储在存储器的哪些地址单元呢？L寄存器的内容将存储在 060015 (300D) 号存储器单元，H寄存器的内容存储在 060016 (300E) 号存储器单元。当然这个办法有缺点，因为它需要使用寄存器对D，才能最后把寄存器对H的内容保存在读/写存储器。即使你使用 STAX 类指令，也必须使用寄存器对B或寄存器对D来保存L寄存器中的地址 (060015，300D)，为H寄存器保存地址 (060016，300E)。为了解决把寄存器对H的内容保存在读/写存储器的问题，我们可以使用 SHLD 指令。

SHLD (直接存储H寄存器和L寄存器的内容) 指令是一条三字节的指令，其中第二和第三字节包含一个 16 位的存储器地址。使用这条指令，我们能直接把寄存器对H的内容保存在读/写存储器里，而不需要使用其它寄存器对的内容作为地址。这条 SHLD 指令在程序中的应用如例 4-9 所示。

例 4-9 SHLD 指令的应用

```
SHLD /把寄存器对H的内容
015  /保存在由SHLD指令的第二，三个字节
060  /所指定的读/写存储单元。
```

既然在例 4-9 中，已经指定了一个 8 位的存储单元，那么 8080 微型计算机怎样才能把 H 和 L 这两个寄存器的内容保存起来呢？8080 把 L 寄存器的内容保存在 060015（300D）存储单元，把 H 寄存器的内容保存在下一个较高地址的连续存储单元，即 060016（300E）存储单元。

这条 SHLD 指令的 16 位地址指定读/写存储器(R/W) 存储单元还是只读存储器（ROM）存储单元呢？即使并不都是，但在大多数时候，SHLD 指令的 16 位地址应该是供读/写存储器用的。但是，在有的例子中，这个 16 位地址不是读/写存储器的地址，例如用这条 SHLD 指令与存储器映象输入/输出设备进行通信。为了把寄存器对 H 的内容保存在读/写存储器里，需要两个连续存储器单元，注意这一点是很重要的。因此，当我们应用 SHLD 指令来编写程序时，必须保留两个存储单元，供存储两个寄存器的内容使用。

把内容存入寄存器对 H，然后把堆栈指示器的内容加到寄存器对 H 的内容上，如例 4-10 所示，这些操作，现在大大地被简化了。

例 4-10 在执行 DADSP 指令之前，使用 SHLD 指令

```
SHLD  /把低位地址保存在060015(十六进制300D)。
015   /把高位地址保存在060016(十六进制300E)
060   /号存储单元。
LXIH  /把八进制数000000
000   /(十六进制数0000)装入
000   /寄存器对H
```

```
DADSP /把堆栈指示器的内容加到寄存器
•     /对H的内容上，并把结果保存在
•     /寄存器对H里
•
```

请读者注意，我们把寄存器对H的内容存入读/写存储器时，没有使用其它寄存器。

还有一条指令与这条SHLD指令类似，这条指令就是LHLD（直接装入H和L）指令。这是一条三个字节的指令，它把两个连续存储单元的内容装入寄存器对H。第一个存储器单元的地址是由这条指令的第二个和三个字节指定的。正如SHLD指令一样，被指定的地址实际上就是存储器单元的地址，这个存储器单元将用作为装入L寄存器的数据的源地址。被指定的这个地址加1以后，作为装入H寄存器的存储器单元的地址。为了把两个连续存储器单元的内容装入寄存器对H，8080可以执行LHLD指令（例4-11）。

例4-11 用一条LHLD指令对寄存器对H进行装入操作

```
LHLD  /把存储器单元 024020（十六进制 1410）
020   /的内容装入L寄存器，再把存储器单元
024   /024021（十六进制 1411）
•     /的内容装入H寄存器。
•
•
```

8080执行这条LHLD指令以后，寄存器对H存储的内容是什么呢？8080将把存储单元024020（1410）的内容装入L寄存器，把存储单元024021（1411）的内容装入H寄存器。例4-12所示的指令序列被执行之后，寄存器对H的内容将是什么

呢？

在例 4-12 所示的程序中，8080 把 060015 和 060016（300D和 300E）这两个存储器单元的内容装入寄存器对H。然后 8080 执行 LXIH 指令时，它把 000000(0000) 这个 16 位立即数字节装入寄存器对H，再把堆栈指示器的内容加到寄存器对H的内容上。正如读者猜想的一样，为什么在例 4-12 的程序中要使用 LHLD 指令，这并没有什么理由。最后一个例子如例 4-13 所示。例中的程序把数据保存在寄存器对H里，并能恢复寄存器对H的内容，同时还能确定堆栈指示器的内容。

例 4-12 测验有关 LHLD 指令的知识

```
LHLD   /把存储器单元 060015 和 060016 (十六进制 300D和 300E)
015    /的内容装入寄存器对H里。
060    /
LXIH   /把八进制数 000000 (十六进制数 0000)
000    /装入寄存器对H里。
000
DADSP  /把堆栈指示器的内容加到寄存器对H
•      /的内容上，并把结果保
•      /存在寄存器对H里。
•
```

8080 执行这个程序(例 4-13)的这一部分以后，寄存器对H的内容似乎没有变化。堆栈指示器的现行内容将被存储在读/写存储器的 024022 和 024023(1412 和 1413) 这两个存储器单元。在这个例子中，用存储器单元 024020 和 024021(1410 和 1411)作为临时存储单元，供寄存器对H使用。当 8080 把 SHLD 指令执行完了，而且这个程序被执行之后，这两个存储单元将仍旧存有寄存器对H的内容。请读者注意：例 4-13 这

个程序中的两条 SHLD 指令所指定的两个地址相差 2 。

例 4-13 确定堆栈指示器的地址，而不打扰寄存器对H的内容。

```
SHLD    /把寄存器对H的内容保存在存储器单元 024020(L)和
020     /024021(H)。
024     /(十六进制 1412 和 1413)
LXIH    /把 000000(十六进制 0000)
000     /装入寄存器对H。
000
DADSP   /把堆栈指示器的内容加到H和L的内容上。
SHLD    /把 SP 的低位字节保存在 024022 存储单元。
022     /把 SP 的高位字节保存在 024023 存储单元
024     /(十六进制 1412 和 1413)。
LHLD    /再从 024020 和 024021(十六进制数 1410 和 1411)
020     /这两个存储单元取回原先H和L
024     /这两个寄存器的内容。
.       /继续向下执行该程序的其
.       /余指令。
```

寄存器对 B 和寄存器对 D，没有与 SHLD 和 LHLD 的等效指令。但是，我们马上就要告诉读者一些容易的方法，用这些方法把这两个寄存器对的内容保存在读/写存储器里。

与 SHLD 和 LHLD 类似的指令有两条。这两条指令是 STA(直接存入A寄存器)和 LDA (直接装入A寄存器)。这两条指令都是三个字节的指令，它们的第二字节和第三字节包含一个 16 位存储器地址。

LDA 指令用来把该指令的第二和第三字节规定的存储单元的内容装入A寄存器。在前面一些程序例子中，我们必须把

一个存储单元地址装入寄存器对B，或寄存器对D，或寄存器对H，从而8080才能执行LDAXB,或LDAXD,或MOVAM指令，以便把一个8位数据字装入A寄存器。但是，LDA指令用它的第二和第三个字节指定存储单元，所以就没有必要用寄存器对为存储单元寻址。

8080可以使用STA指令来把A寄存器的内容存入该指令的第二和第三个字节所指定的存储单元。这就是说，不使用寄存器对来给存储单元寻址，8080也可以执行STAXB,STAXD或MOVMA这些指令。例4-14和例4-15是怎样应用STA和LDA指令的程序实例。

例 4-14 LDA指令的使用

```
LDA             LDA
132             COUNT
001             0
```

例 4-15 STA指令的应用

```
STA             STA
307             CHAR
034             0
```

在例4-14中，把001 132(015A)这个存储单元的内容装入A寄存器，这个存储单元是由LDA指令的第二和第三个字节寻址的。8080执行这条LDA指令时，这个存储器单元的内容保持不变。如例4-14所示，符号地址也可以作为LDA指令的第二和第三字节使用。对于这条具体LDA指令而言，符号地址COUNT所分配的存储器单元的内容，被装入A寄存器。

在例 4-15 里，A寄存器的内容被存储在存储单元 034307 (1 CC 7)，而这个存储单元是由符号地址 CHAR 指定的。当执行 STA 指令时，A寄存器的内容仍旧保持不变。对于 8080 微处理机的其它通用寄存器来说，它们都没有与 STA 或者 LDA 指令等效的指令。

假设你需要确定具体某个存储单元是否存储了 215(8D) 这个数据字，那么，你应该怎样才能完成这项任务呢？现在，你应该能够想到许多不同的办法来完成这个任务，例 4-16 就包括了三个这样的例子。

例 4-16 确定某个存储单元是否存储了数据字 215(8D)

```
LXIH        LXIH        LDA
123         123         123
062         062         062
MOVAM       MVIA        CPI
CPI         215         215
215         CMPM
```

如果采用例 4-16 所示的头两个方法，则用寄存器对H和A寄存器来确定寄存器对H所寻址的这个存储单元的内容是不是 215(8D)。在 LXLH 指令被执行之前，必须把寄存器对H的内容存储起来。为了完成这一操作，你可以使用一条 SHLD 指令和一条 LHLD 指令。然而，这就是说，采用这两种方法，需要增添两条指令。存储这两条指令需要六个存储单元。例 4-16 中的最后一种方法，是采用 LDA 指令，不仅指令较短，而且它只需要使用A寄存器，不需要A寄存器和寄存器对H两者都用。

利用堆栈来存储数据、地址和状态信息

在前面的所举的例子中，当 8080 调用子 程 序时，堆栈总是被用来存贮返回地址。 8080 微型计算机执 行 调用指令时，它把这条指令的返回地址、以两个 8 位值的形式自动地压入堆栈。 8080 执行返回指令时，它把这条指令的 最 后两个八位值自动地从堆栈弹出，然后装入程序计数器(PC)。

假设读者把数值存储在 B，C，D，E，H和 L 寄存器。如果你要 8080 调用转子子程序，最终执行完子 程序， 返回到主程序，那么，这些数值仍旧保存在这些寄存器里吗？这个问题，我们还没有办法作肯定回答。你应该以程序指令为基础，逐条逐条地检查子程序，注意哪些寄存器被使用了，从而确定其中某个寄存器的内容是否已经被改变了。如果读者已经安排 8080 调入第三章的一小时延迟子程序(例 3-6)，那么 B 、C、D、E 这四个寄存器的内容都被改变了，因这个延时子程序为达到延时的目的，使用了这些寄存器。

读者能克服这个严重的局限性吗？在子程序被调用之前，读者可以把各个寄存器的内容存储在读/写存储器的一个区内。8080 微型计算机从延时子程序返回后， 从读/写存储器的地址单元取回这些数据。完成这一操作所需要的指令是冗长的，它需要使用 8080 的两个通用寄存器， 来 为将要存储这些数据值的读/写存储器的地址单元寻址。

读者可以使用堆栈来暂时存储这些通用寄存器的内容，即使 8080 正在执行子程序时也可以，这便是 8080 微处理机的优

秀特征之一。实际上，你可以把各个寄存器的内容(A～E，H和L)和8080的五个标识位，根据需要，随时存入堆栈。虽然不能单独把某个寄存器的内容压入堆栈，或从堆栈弹出，但是，可以把寄存器对的内容保存在堆栈里。寄存器对B、D、H和**处理机状态字**(PSW)的压入指令和弹出指令的操作码如表4-6所示。

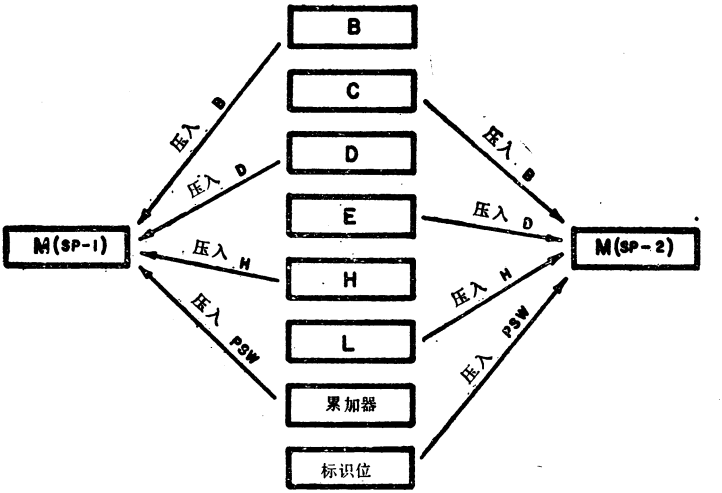
8080执行PHSHPSW指令或者执行POPPSW指令时，要把整个16位信息都压入堆栈或者都从堆栈弹出。这16位信息包括A寄存器的一个8位字节，和一个包含五位标识位（进位标识，零标识，符号标识、校验标识和辅助进位）的标识字以及3位零位。**因此，送给堆栈的位数或从堆栈弹出的数位总共为两个8位字节(16位)**。读者应该还记得：这种情况也正如调用和返回类指令一样；所有的地址都以两个八位字节存入堆栈或者从堆栈取回。

表 4-6　　寄存器对用的堆栈指令一览表

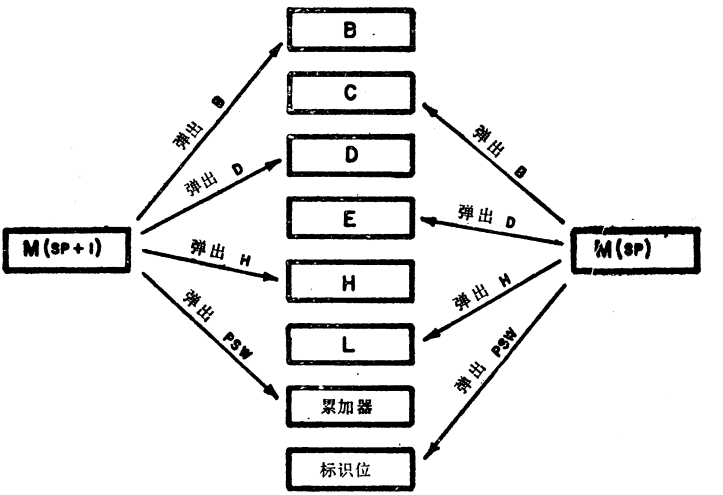
八　进　制	助　记　符	十六进制
305	PUSHB	C 5
325	PUSHD	D 5
345	PUSHH	E 5
365	PUSHPSW	F 5
301	POPB	C 1
321	POPD	D 1
341	POPH	E 1
361	POPPSW	F 1

所有的压入(PUSH)和弹出(POP)指令都是单字节指令

这些压入指令和弹出指令对于堆栈指示器M(SP)，堆栈指示器加1，M(SP+1)，堆栈指示器减1，M(SP−1)和堆栈



（A）任何一条指令被执行之后，堆栈指示器的地址被减去 2



（B）任何一条指令被执行之后，堆栈指示器的地址增 2

图 4-2　表明压入和弹出操作的方框图（每个操作涉及一个寄存器对和两个存储单元）

指示器减 2，M(SP－2)，所寻址的存储器单元的内容的影响如图 4-2 所示。

为了**有效地**使用这些指令，读者必须保证数据被压入堆栈和从堆栈中弹出的顺序正好**相反**。这是因为 8080 采用的堆栈是后进先出堆栈。这点可以从例 4-17 就很容易地看到。

例 4-17 所表示的压入堆栈和弹出堆栈的顺序是正确的；寄存器的内容被从堆栈弹出的顺序与它们被压入堆栈的顺序正好相反。例如，

例 4-17 适当地使用堆栈来存储寄存器的内容。

```
PUSHH    /把 H 和 L 寄存器的内容保存在堆栈里。
PUSHPS W/把 A 的内容和标识位保存在堆栈里。
PUSHB    /把 B 和 C 寄存器的内容保存在堆栈里。
CALL     /调入叫做"COUNT"的子程序。
COUNT
0
POPB     /从堆栈弹出 B 和 C 寄存器的内容。
POPPSW   /从堆栈弹出 A 寄存器的内容和标识位。
POPH     /从堆栈弹出 H 和 L 寄存器的内容。
```

寄存器对 H 的内容是最先被压入的，而最后被弹出的。在例 4-17 中，子程序 COUNT 并不影响寄存器对 D 的内容，所以，没有必要把寄存器对 D 的内容保存在堆栈里。如果 8080 在执行调入 COUNT子程序的操作时，可以检查堆栈的具体内容，那么，你所见到的结果如表 4-7 所示。我们假设，8080 在执行这个程序(例 4-17)之前，执行一条 LXISP 指令，将堆栈指示器置成 024 100 (1440)。

请读者注意，因为返回地址是最后进入堆栈的，所以，它必须最先弹出堆栈。我们为什么要把返回地址存储在堆栈呢？

表 4-7　　在 PUSHH，PUSHPSW，PUSHB 和 CALL 指令执行之后，堆栈的内容

八进制堆栈地址		十六进制堆栈地址	
预置的 SP——024	100	1440	
024	077H寄存器	143F	PUSHH
024	076L寄存器	143E	
024	075A寄存器	143D	PUSHPSW
024	074标志字	143C	
024	073B寄存器	143B	PUSHB
024	072C寄存器	143A	
024	071高位返回地址	1439	CALL
现行的 SP——024	070低位返回地址	1438	

返回地址被存储在那儿，是因为 8080 要执行一条调用指令。当 8080 执行这个子程序末尾的那条返回指令时，从堆栈弹出这个返回地址，然后装入程序计数器(PC)。这个地址将成为下一条要执行的指令的地址。8080 从堆栈取出这个返回地址以后，堆栈指示器将指示到存储单元 024 072 (143 A)。8080 从COUNT 子程序返回后，下一条要执行的指令是 POPB，所以，8080 从堆栈弹出 B 和 C 寄存器的数，装入这两个寄存器(寄存器对 B)。这时，堆栈指示器指示的存储单元是 024074 (143C)。8080 从堆栈取出 PSW (A 寄存器的内容和标识字)以后，堆栈将指示到 024 076 (143 E) 号存储单元。最后，8080 执行 POPH 指令时，堆栈将指示到的存储单元是 024 100 (1440)。只要压入(PUSH)指令和弹出(POP)指令用得适当，并且按适当的顺序执行，在使用堆栈进行存储操作时，就不会出现问题。

假设 8080 执行例 4-18 这个程序。最后一条指令 POPB 被

执行之后，B和C寄存器的内容应该是什么呢？这个程序似乎很复杂，其实却比较简单。8080 调用 GUESS 这个子程序之前，分别把数据值装入寄存器对B、D和H。然后 8080 再把这些数压入堆栈。在 GUESS 子程序里，8080 执行几条加 1 指令和减 1 指令，这些指令会改变寄存器 B. C.D.H 和 L 的内容。但是，因为 D.H 和 B 这三个寄存器对的内容都是堆栈取出的，所以，8080 返回到主程序时，GUESS 这个子程序如何改变这些寄存器对的内容实际上是无关紧要的。然而，需要做的事情全在于确定寄存器对被压入或弹出堆栈的顺序。

寄存器对D是最后被压入堆栈，最先从堆栈取出的寄存器对。所以，8080 执行 POPD 这条指令时，寄存器对D的内容是 020 020(1010)。8080 把寄存器对D的内容从堆栈取出后，再从堆栈取出寄存器对H的内容。把寄存器对H的内容压入堆栈的顺序是倒数第二，仅在寄存器D的前面。因此，8080 执行 POPH 指令时，寄存器对H的内容将是 123001 (5301)。寄存器对B是被最先将其内容压入堆栈、最后取出的寄存器对，所以，它的内容是 002100(0240)。

总之，我们可以把压入指令和弹出指令当作数据传送指令。象数据传送指令一样，当执行压入指令时，8080 可以把寄存器对的内容复制在读/写存储器的某一部分，即堆栈区。因此 8080 执行 PUSHB 指令时，寄存器对B的内容不变。也就是说，这条指令被执行之后，和这条指令被执行之前，寄存器对B的内容是相同的。同样，当 8080 从堆栈取出数据，并送回到寄存器对时，用作堆栈的读/写存储器的一个地址单元存储的数据仍旧是 8080 执行这条压入指令时写入存储器的这个数。

从上面的讨论，你已经看到，在编程序时，压入指令和弹

出指令带来了许多方便和灵活性。在 8080 调用子程序之前，可以把数据值暂时保存在堆栈里，然后，8080 将程序执行控制从子程序返回到主程序时，把暂时保存的数据从堆栈里取出来。当然，这些压入指令和弹出指令在程序的任何部分都可以使用，不一定非在调用指令的前、后执行这些指令。这样，则会带来一个有趣的问题。8080 是在调用指令之前和调用指令之后，执行压入指令和弹出指令呢，还是实际上在该子程序的开始和结尾执行压入指令和弹出指令呢?这些差别如例 4-19 所示。

例 4-18 把数值压入堆栈或从堆栈弹出

```
LXISP  /装入堆栈指示器，因为将要执行
STACK  /压入(PUSH)和弹出(POP)指令。
0
LXIH   /把 123  001 装入寄存器对H，
001    /把 001 装入L，把 123 装入H
123    /(八进制 123001＝十六进制 5301)
LXIB   /装入寄存器对B，把 100 装入C，
100    /把 002 装入B
002    /(八进制 002100＝十六进制 0240)。
LXID   /装入寄存器对D，E寄存器和
020    /D寄存器分别装入 020
020    /(八进制 020  020＝十六进制 1010)
PUSHB  /把B和C的内容存入堆栈。
PUSHH  /把H和L寄存器的内容存入堆栈。
PUSHD  /把D和L寄存器的内容存入堆栈。
CALL   /利用符号地址“GUESS”调用“GUESS”
GUESS  /子程序。
0
```

```
        POPD   /从堆栈弹出D和E寄存器的内容。
        POPH   /从堆栈弹出H和L寄存器的内容。
        POPB   /从堆栈弹出B和C寄存器的内容。
        •
        •
        •
GUESS,  INXH   /寄存器对H的内容加1。
        DCXB   /寄存器对B的内容减1。
        INRD   /D寄存器的内容加1。
        JNZ    /D已经被减量为000吗?
        GUESS  /没有，再执行GUESS循环。
        0
        RET    /是000，8080返回到主程序。
```

把压入指令和弹出指令适当地编入子程序，比在每次调用子程序的前后要反复执行这些指令的做法要好得多。请读者记住，这就是我们为什么事先编写好这个子程序的理由；这样可避免重复写这一特定的指令序列。由于在这个子程序编入了压入指令和弹出指令，所以，8080执行这个子程序时，它能够把寄存器对B和H的内容存储在堆栈里。正好在这个程序的返回指令执行之前，8080从堆栈取出寄存器对B和H的内容。无论什么时候使用堆栈，读者仍旧必须保证堆栈有充足的存储单元，供存储这些数据字用。这是通过8080执行LXSP指令，使堆栈指示器预置初值时实现的。因为数字值被压入堆栈时，将堆栈指示器的内容被减量，所以，一般规定，堆栈的起始地址从“空”读/写存储器单元最高有效地址开始，或者接近，其“下面”为堆栈区。

例 4-19 压入和弹出堆栈的时刻

```
        程序1                                    程序2
        •                                        •
        •                                        •
        PUSHH  /保存H和L的内容。                  •
        PUSHB  /保存B和C的内容。                  CALL
        CALL   /执行                             TEST
        TEST   /这个子程序。                      0
        0                                        •
        POPB   /取回B和C的内容。                  •
        POPH   /取回H和L的内容。                  •
        •                                        •
        •                                        •
TEST,   MVIB
        123             PUSHB   /保存B和C的内容
        LXIH            MVIB
        134             123
        036             LXIH
COUNT,  MVIM            134
        000             036
        INXH    COUNT,  MVIM
        DCRB            000
        JNZ             INXH
        COUNT           DCRB
        0               JNZ
        RET             COUNT
                        0
                        POPB    /取回B和C的内容
                        POPH    /取回H和L的内容
```

读者还知道，所有的子程序都必须包括：(1) 相同的压入

指令和弹出指令条数；(2) 一条条件或无条件返回指令。如果不满足这些条件，则堆栈可能越界；也就是说，堆栈可以很快地变得很大。如果出现这种情况，存储在读/写存储器的程序和数据，或者二者之一将会被破坏。超越堆栈几乎总是毁灭性的失败。遗憾的是，造成超越堆栈的情况非常容易出现。例 4-20 给出了一个典型的例子。

例 4-20 怎样才能不调用子程序

```
TEST,   CALL    /调用"测试"子程序
        TEST
        0
        INRA    /当 8080 返回时，将 A 寄存器
                /的内容加 1。
```

虽然大多数程序设计员不会犯这种具体错误，但是，当你的子程序有 15 级嵌套时，或者说当你的程序的长度有一、二千条指令时，则很容易犯这种错误。如果堆栈出现越界，读/写存储器的内容是什么呢？一般说来，读/写存储器填满了相同的 16 位数，这个 16 位数是调用指令的返回地址，即通过压入操作所存储的寄存器对数据，这个数据没有相应地"弹出"。在例 4-20，存储器填入的是存储单元的十六位地址，INRA 指令存储在这里。

再启动指令（单字节调用指令）

读者怎样才能使 8080 微处理机执行子程序呢？在读者编制的程序中，必须有一条调用指令供 8080 执行。这条调用指令是一条三字节的指令，它不同于转移指令；当调用指令被执

行时，返回地址存储在堆栈。还有一种单字节指令，叫做再启动指令，这种指令的作用与调用指令的作用相同，但是，不同的是：必须把再启动指令调用的子程序，从特定的地址开始，存入存储器。再启动指令有 8 条，把它们归纳起来，则如表 4-8 所示。

表 4-8　　再启动指令一览表

指　令	操作码 八进制	操作码 十六进制	被调用的存储地址 八进制	被调用的存储地址 十六进制
RST0	307	C7	000 000	0000*
RST1	317	CF	000 010	0008
RST2	327	D7	000 020	0010
RST3	337	DF	000 030	0018
RST4	347	E7	000 040	0020
RST5	357	EF	000 050	0028
RST6	367	F7	000 060	0030
RST7	377	FF	000 070	0038

* 与布线的复位一致。

8080 执行一条再启动指令时，它转移到表 4-8 所列出的特定的存储器单元；同时，把返回地址保存在堆栈。8080 执行调用指令时，所发生的事件的序列也是这样的。然而，用一条单字节指令，怎么能够指定一个 16 位的地址呢？那是不可能的。当任何一条再启动指令被执行时，8080 转移到的这个地址是固定的，并且你不能改变这个地址。不管 8080 执行哪一条再启动指令，8080 所转移到的这个地址的高位总是 000(00)。低位地址是变化的，其变化取决于执行哪一条再启动指令。读者还应该注意，如果采用 8 进制数制，再启动指令转移到的低位地址部分是很容易记住的；RST 0 的低位地址部分是 000；RST 1 的是010；RST 2 的是 020；RST 3 的是 030；等等。即

使再启动指令是单字节的调用指令，每执行一条这样的指令时，就有一个返回地址保存在堆栈中。例 4-21 就是使用了一条再启动指令的一例。

例 4-21 用再启动指令调用子程序

```
        •
        •
        MVIA   /把 001 装入 A 寄存器。
        001
        RST 7  /执行这条 RST 7 再启动指令。
        OUT    /把 A寄存器的内容
        123    /输出给 123 号输出设备。
        •
        •
        *000  070
SETUP,  LXID   /装入寄存器对 D。
        001    /把 001 装入 E 寄存器，
        000    /把 000 装入 D 寄存器。
        MVIH   /把 030 装入 H 寄存器。
        030
        RET    /返回到主程序。
```

例 4-21 是读者所见到的在程序中使用星号的第一个例子。这个星号用来表示程序的启动地址，或子程序的启动地址，或数据阵列的启动地址。当用汇编语言来汇编源程序时，星号使汇编程序在列**汇编程序表时间开始打印出新的地址**。这就是说，SETUP(例 4-21)这个子程序存储在存储器，是从 000070 (0038)这个地址开始存储的。

虽然，这个程序并没有特殊用途，但是，它确实具体体现

了调用子程序的再启动指令 RST 7 的应用。当 8080 执行这个程序时，RST 7 指令会把返回地址压入堆栈；这个返回地址用来存储 OUT 指令的存储单元的地址，然后，8080 转移到 000070(0038)这个存储地址，去执行该子程序。当然，在被再启动指令调用的这个地址上必定有一个子程序。当 8080 执行此子程序的末尾的那条返回指令时，从堆栈弹出返回地址，并且用作为下一条要执行的指令的地址。总之，你已经懂得了单字节再启动指令是怎样实现三个字节的访问指令的同样的功能的。实际上，我们可以用一条调用指令把例 4-21 这个程序改写成例 4-22 这个程序。

例 4-22 用调用指令代替再启动指令。

```
        •
        •
        MVIA
        001     /把 001 装入 A 寄存器。
        CALL    /调用 SETUP 子程序。
        0
        OUT     /把 A 寄存器的内容输出到
        123     /端口 123
        •
        •
   *000070
SETUP,  LXID    /装入寄存器对 D。
        001     /把 001(十六进制 01)装入 E 寄存器，
        000     /把 000(十六进制 00)装入 D 寄存器。
        MVIH    /把 030(十六进制数 18)装入 H。
        030
        RET     /返回到主程序。
```

三个字节的调用指令，需要两个存储器单元来存储这条指令所包含的子程序地址字节，这是使用三字节调用指令的缺点之一。用再启动指令而不用调用指令，能够节省的存储器单元的多少，对于你来说可能是重要的，也可能不是重要的。使用再启动指令的局限性之一，似乎是被调用的子程序所占用的存储单元必须是 8 个或者更少。这是因为在再启动指令转移到的地址之间，只有 8 个存储器单元。果真是这样吗？不，并不是。但是，当你希望的子程序所需要的存储器单元大于 8 个时，即使在一条再启动指令的转移地址之间只有 8 个存储单元，也可采用例 4-23 所示的方法，用再启动指令来完成调用功能。

如果读者在一个程序中，使用再启动指令，而不用调用指令，那么，这个程序可以节省两个存储单元。所节省的数量不是太大，除非你可以在整个程序中多处使用再启动指令。如果在一个程序中五十次使用再启动指令，那么，可以节省多少个存储单元呢？可以节省一百个存储单元。如果读者要把 1056 个字节的程序，挤入 1024 个字节的只读存储器(ROM)，那么，多次使用再启动指令是很重要的。再启动指令的应用的有关的问题之一是，通过中断与 8080 连接的输入/输出设备，也可以使用再启动指令。这就是说，再启动指令不能随意使用。这个问题我们将在中断这一章进行讨论。

例 4-23 长子程序与再启动指令的使用

```
   *001  000
START,  MVIA
        001      /把 001 装入 A 寄存器。
        RST 7    /执行再启动指令 RST 7。
        OUT      /把 A 寄存器的内容
        123      /输出给 123 号端口。
```

```
          *000  070
SETUP,    LXIB    /把 001 装入 C 寄存器，
          001     /把 000 装入 B 寄存器。
          000
          JMP     /转移到符号地址“MORE”，连接该
          MORE    /子程序的其余指令。
          0       /

     *001 200
MORE,     MVIH    /把 030 装入 H 寄存器。
          030
          MVID    /把 023 装入 D 寄存器。
          023
DECIT,    DCRD    /D 寄存器减 1。
          JNZ     /D 的内容是 000 吗?
          DECIT   /不是，则执行 JNZ-DECIT
          0
          RET     /D=000 时，8080 返回。
```

使用寄存器对 H 操作

当寄存器对B、D或H存储一个地址时，8080可以把数据送到存储单元，也可以从存储单元取出数据，其方法我们已经向读者介绍过了。把寄存器对H的内容送到存储器，或者从存储器取出寄存器对H的内容，所用的指令也已经讨论过了。

这些指令是 SHLD 和 LHLD。 把H寄存器和L寄存器作为寄存器对用，另外还有四条指令。

寄存器对交换操作

我们首先要讨论的是 XCHG 指令，它的操作码是 353(EB)。这条指令用来**将寄存器对 D 的内容和寄存器对 H 的内容被交换**。8080 执行这条指令之后，E寄存器的内容送到L寄存器，L寄存器的内容送到E寄存器，D寄存器的内容移到H寄存器，H寄存器对的内容移到D寄存器。请读者记住，这条指令把这两个寄存器对的 16 位内容互相交换。8080 执行例 4-24 的这个程序后，寄存器对D和H应该存储的内容是什么呢？

这条 XCHG 指令被执行以后，H寄存器存储的内容是 002(02)，L寄存器存储的内容是 001(01)，D寄存器存储的内容是 004(04)，E寄存器存储的内容是 003(03)。当然，如果这是所希望的最后结果，那么例 4-25 所列出的这个程序很容易地执行。

例 4-24 XCHG 指令的使用

```
SWAP,   LXID    /装入寄存器对 D。
        001     /把 001(十六进制 01)装入 E 寄存器。
        002     /把 002(02)装入D寄存器。
        LXIH    /装入寄存器对 H，
        003     /把 003(03)装入 L 寄存器。
        XCHG    /把D和E寄存器的内容与H和L
                /寄存器的内容交换。
        •
        •
```

例 4-25 用 LXI 指令取代 XCHG 指令

```
EQUIV,  LXID
        003
        004
        LXIH
        001
        002
```

读者还能想出其它两个方法，用来把寄存器对H和D的内容交换吗？如果你想不出，请看例 4-26。

例 4-26 XCHG 指令的等效指令

```
                MOVAH
          PUSHD MOVHD
XCHG = PUSHH = MOVDA
          POPD  MOVAL
          POPH  MOVLE
                MOVEA
```

读者已经知道如何使用 SHLD 和 LHLD 这两条指令来把寄存器对H的内容送到存储器，和从存储器取出寄存器对H的内容。假设你需要把寄存器对D的内容存入存储器，那么，应该怎样完成这一任务呢？你可以使用例 4-27 列出的程序指令，这个程序把寄存器对H的内容存储在堆栈里，并把分配给符号地址 STORE 的读/写存储器地址装入寄存器对H里。接着，把E寄存器的内容存入寄存器对H寻址的读/写存储单元。然后，8080 执行 INXH 指令，将存储地址加 1，再把D寄存器的内容存入存储器。接着 8080 从堆栈取出寄存器对H的原来的内容；8080 继续执行该程序的其余部分。

如果你使用 XCHG 指令，则可以编写出更简单的程序；如例 4-28 所示。

例 4-27 把寄存器对D的内容存入读/写存储单元

```
•
•
PUSHH   /把寄存器对 H 的内容保存在堆栈。
LXIH    /把被分配给符号地址"STORE"的
STORE   /地址装入寄存器对 H 里。
0
MOVME   /把 E 寄存器的内容保存在存储器。
INXH    /寄存器对 H 的存储器地址加 1
MOVMD   /把 D 寄存器的内容保存在存储器。
POPH    /从堆栈取出寄存器对 H 的内容。
•
•
```

例 4-28 把寄存器对D的内容保存在读/写存储器的改进方法

```
•
•
XCHG    /把寄存器对 D 和 H 的内容进行交换。
SHLD    /把寄存器对 H 的内容保存在两个连续
STORE   /的存储器单元，从"STORE"这个符号
0       /地址开始。
XCHG    /把寄存器对 D 和 H 的内容
•       /交换成原来的值。
•
```

例4-28 所列出的指令首先把寄存器对D的内容与寄存

器对H的内容进行交换。把要被保存在读/写存储器的数据值存储在寄存器对H。当8080执行SHLD指令时，再把寄存器对H的内容存入读/写存储器。L寄存器的内容存储在符号地址STORE指定的存储单元里；H寄存器保存在STORE+1这个存储器地址。把数据值存入存储器后，再把寄存器对D和H的内容进行交换。现在，寄存器对D和H的内容，与该指令序列被执行之前的一样。

例4-29 用LHLD指令，把存储器的内容保存在寄存器对D

```
•
•
XCHG   /交换寄存器对 D 和 H 的内容。
LHLD   /把符号地址"STORE"的内容
STORE  /装入寄存器对 H。
0
XCHG   /再交换寄存器对 D 和 H 的内容。
•
•
```

我们还可以用类似的指令序列来把两个连续存储器单元（例4-29和例4-30）的内容装入寄存器对D。读者将会看到，使用一条XCHG指令，可以节省三个存储单元；而且这个程序更容易理解，并且它所需要的执行时间较少。

例4-30 用传送指令把存储单元的内容装入寄存器对D

```
•
•
PUSHH  /把寄存器对 H 的内容保存在堆栈。
LXIH   /把符号地址"STORE"的存
STORE  /储器地址装入寄存器对 H。
```

```
0
MOVEM /把存储单元的内容装入 E 寄存器。
INXH  /存贮器地址加 1。
MOVDM /把此存储单元的内容装入 D 寄存器。
POPH  /从堆栈弹出寄存器对 H 的内容。
.
.
```

我们已经向读者介绍了几种方法，用来把寄存器对D和H的内容保存在读/写存储器，其中包括例 4-31 所示的方法。

例 4-31 把寄存器对D和H的内容保存在存储器里。

```
保存寄存器对 H 的内容        保存寄存器对 D 的内容
    SHLD                        XCHG
    TEMPO                       SHLD
      0                         TEMPO
                                  0
                                XCHG
```

但是，怎样才能把寄存器对 B 的内容保存在存储器里呢？请读者记住，寄存器对 B 和H没有交换指令。例 4-32 所列出的程序可以用来把寄存器对 B 的内容保存在存储器里。

例 4-32 把寄存器对 B 的内容保存在存储器里。

```
.
.
PUSHH  /把寄存器对 H 的内容保存在堆栈上。
PUSHB  /把寄存器对 B 的内容保存在堆栈上。
POPH   /弹出 B 的内容，送入寄存器对 H。
SHLD   /把寄存器对 H(B)的内容保存在
TEMPO  /读/写存储器里。
0
```

```
POPH    /恢复寄存器对 H 的内容。
•
•
```

在例 4-32 里，我们是把寄存器对H的内容压入堆栈；然后把寄存器对 B 的内容压入堆栈；再从堆栈取出，**送入寄存器对 H**。这种操作过程正如寄存器对-寄存器对数据传送指令的一样。当 8080 执行 SHLD 指令时，再把寄存器对H的内容（这个数值原来存贮在寄存器对 B）存入读/写存储单元。这时，寄存器对H的原来的内容仍旧存储在堆栈；原来在寄存器对 B 的 16 位数值现在被保存在寄存器对H里。只要把寄存器对H的内容从堆栈弹出，再送入寄存器对H，8080 微处理器就能继续向前执行这个程序的其余指令。这两个寄存器对的内容好象是原样子。请读者注意，寄存器对 B 和H并没有发生完全的交换。

例 4-33 寄存器对 B 的内容与寄存器对D 的内容交换，或者与寄存器对H的内容交换。

交换寄存器对 B 和 D 的内容	交换寄存器对B 和 H 的内容
PUSHB	PUSHB
PUSHD	PUSHH
POPB	POPB
POPD	POPH
或	或
PUSHD	PUSHH
PUSHB	PUSHB
POPD	POPH
POPB	POPB

如果读者确实需要把寄存器对 B 的内容或者与寄存器对D的内容交换，或者与寄存器对H的内容交换，那么，这是很容

易做到的。如果读者需要这种操作，PSW 寄存器对也可以交换。

改进的数据传送程序

在这一章的开始，我们讨论了使用程序来把数据块（数据陈列）从存储器的一个部分传送到另一个部分。例 4-34 就是所使用的程序之一。

假设读者需要使例 4-34 列出的这个程序：(1) 尽可能具有通用性；(2) 把这个程序存入只读存储器（ROM），并且仍旧能够改变始地址、末地址和字节数。这就是说，地址 021100 和 051100 (1140 和 2940)，或这个字节数 (012023，0 A 13)，不会总是所要使用的合适的地址或计数数值。读者需要做的是把首地址、末地址和字节数，存储在某些读/写存储器单元，然后用数据块传送程序对这些数据进行存取。例 4-35 这个程序是用来完成这项任务的。

例 4-35 这个程序的第一条指令是用来把连续两个存储单元的内容装入寄存器对H的；这两个存储单元是从符号地址BC开始的。然后，把这两个单元的 16 位数值压入堆栈，再从堆栈弹出，送入寄存器对B。现在已经把字节数（BC）装入寄存器B。然后，再把两个连续存储器单元的内容存入寄存器对H，这两个连续存储器单元是从符号地址 FA 开始的。这个符号地址是要传送数据的最后地址。然后交换寄存器对D和H的内容，从而把地址（FA）存储在寄存器对D里。接着把要传送的数据阵列的首地址（IA）装入寄存器对H。现在，我们已经把字节数，末地址和首地址都装入到了寄存器对B、D和H。当 8080 微处理器执行 TRNSFR 这个循环时，进行数据块的传送操作。

例 4-34 数据块传送程序

```
MOVE,    LXIH    /装入这个数据源的始
```

```
        100       /地址。
        021
        LXID      /装入这个数据
        100       /的目的始地址。
        051
        LXIB      /把要传送的字节的数目
        023/      /装入B寄存器。
        012
TRNSFR, MOVAM /利用H和L寄存器取一个数据字节。
        STAXD /利用D和E存储一个数据字节。
        INXH  /源地址加1。
        INXD  /目的地址加1。
        DCXB  /字节计数器减1。
        MOVAB /计数器的值已经是0吗?
        ORAC
        JNZ
        TRNSFR /不是，执行转移指令。
        0
        •         /是的，继续执行程序。
        •
        •
```

装入程序计数器

使用寄存器对H的另一条指令是PCHL指令，这条指令的操作码是八进制351（E 9）。这条指令是一个字节的转移指令。但是，它能指定任何一个16位地址！用一条单字节指令怎么能够规定任何一个16位地址呢？只要把寄存器对H的16位内容装入程序计数器(PC)就可以实现此任务；这是PCHL指令所执行的唯一操作。

例 4-35 用 LHLD 指令来存取地址和计数数

```
MOVE,   LHLD   /把分配给符号地址"BC"的存储
        BC     /单元的内容装入寄存器对 H。
        0
        PUSHH  /把这个数值压入堆栈，
        POPB   /然后从堆栈弹出，装入 B 和C。
        LHLD   /现在，把被存储在符号地址"FA"上的
        FA     /末地址装入 H 和 L。
        0
        XCHG   /把 D 和 E 与 H 和 L 的内容交换，把末地址存入
               D 和E。
        LHLD   /把始地址装入
        IA     /H和L。
        0
TRNSFR, MOVAM/利用H和L取一个数据字节。
        STAXD  /利用 D 和 E 存储一个数据字节。
        INXH   /源地址加 1 。
        INXD   /使目的地址加。
        DCXB   /使字节计数器减 1 。
        MOVAB/计数器的内容已经是 0 吗?
        ORAC
        JNZ
        TRNSFR/不是，执行转移操作。
        0
        .      /是的，继续执行程序。
        .
        .
BC,     023    /这是计数字节
        012
FA,     100    /这是末地址
```

```
        040
IA,     100     /这是首地址
        021
```

因为程序计数器(PC) 指示到要执行的存储下一条指令的存储单元，所以 PCHL 这条指令完成的功能与无条件转移指令JMP 所完成的功能是相同的。

读者可以把 PCHL 指令认为是：把H和L寄存器的内容装入程序计数器 (PC)。实际上，体现了这条指令的功能的简单程序实例即使有，也很少。但是，例 4-36 所列出的这个程序能使读者对怎样使用这条 PCHL 指令有所了解。

例 4-36 使用 PCHL 指令

```
LXIH
345
005
PCHL
```

如果执行例 4-36 所给出的指令序列，那么这条 PCHL 指令被执行以后，下一条要执行的指令的操作码被存储在哪个地址呢？这条指令的操作码一定存储在 005345(05 E 5)号地址单元；8080 执行 LXIH 指令，把这个地址装入寄存器对H。

堆栈的交换操作

XTHL 指令，其操作码是 343 (E 3)，它也是一条很复杂的指令。这条指令被用来把存储在堆栈的最上面的两个字节与寄存器对H的内容交换；堆栈存储的最上面的两个字节或者是某个寄存器对的内容，或者是一个 16 位返回地址。

如果读者需要确定或者检查存储在堆栈的最后两个 8 位字节，那么可以执行 POPB，或 POPD，或 POPH 指令。但是，

使用一条这样的指令，意味着我们必须把有关的数从堆栈弹出，送入一个“空”寄存器对。你不能凭执行一条压入指令来空出寄存器对，因为下一条弹出指令直接把这个被“保存”的数据再装回到寄存器对。此外，如果一条弹出指令被执行，堆栈指示器的内容将加 2 。因为这条 XTHL 指令把寄存器对H的内容与被存储在堆栈的最后两个地址进行交换，并不影响堆栈指示器的地址；只要执行另一条 XTHL 指令，就能把数据值恢复到它们的原来的“位置”。

假设读者这样编写一个程序，以便在该子程序被执行完毕后，能确定程序执行控制会返回到指令操作码的地址。为了这一目的，我们便可以使用这条 XTHL 指令。它使得程序能确定这个子程序被调用的那一点地址。

在例 4-37 中，每当 8080 转入 TEST 子程序时，存储在堆栈的返回地址和寄存器对H的内容进行交换。在这个程序例中，TEST 子程序的每一条 XTHL 指令被执行之后，寄存器对H的内容应该是什么呢？我们来观察一下从这个程序三个不同的点调入 TEST 子程序的情况。 8080 分别执行每条调用指令以后，寄存器对H的内容应该是 001203，001207，和 001213 (0183，0187 和 018 B)。 当执行此子程序时，8080 执行第一条 XTHL 指令后，寄存器对H所存储的返回地址可以被改变，仅仅检查一下这一个返回地址。这种操作是在 8080 执行在 TEST 子程序末尾的第二条 XTHL 指令把这个地址放回到堆栈之前进行的。如果寄存器对H保存的地址被改变了。那么，第二条 XTHL 指令将把一个修改了的地址放回到堆栈里。

例 4-37 XTHL 指令的应用：

```
*  001  200
START, CALL/调用该测试子程序
```

```
        TEST
        0
        INRA /A的内容加1。
        CALL /调用该测试子程序。
        TEST
        0
        INRB /B寄存器的内容加1。
        CALL /调用这个测试子程序。
        TEST
        0
        HLT  /暂停。
TEST,   XTHL /把寄存器对H的内容和堆栈上
        ·    /的最上面两字节交换。
        ·
        ·
        XTHL /把堆栈上的最上面的两字节与H和L交换。
        RET  /返回到主程序。
```

从寄存器对H装入堆栈指示器

LXISP，这条指令可以用于装入堆栈指示器。但是，如果这条指令用于一个存储在ROM只读存储器中的程序中，那么，则不能改变LXISP指令的两个地址字节。读者已经看到了一些使用LXISP指令的例子；SPHL指令所完成的功能与LXISP指令所完成的功能是相同的。SPHL指令的操作码是371（F9），这条指令与我们刚才讨论过的PCHL指令也很相似。读者可以把SPHL指令认为是：把H和L寄存器的内容装入堆栈指示器（SP）。

用这条指令，你可以把任何一个16位数装入寄存器对H，

然后执行一条 SPHL 指令，把这些数值装入堆栈指示器。当然，8080 执行 SPHL 指令之后，这个数仍然保留在寄存器对H里。例 4-38 所给出的指令序列利用一条 SPHL 指令来执行堆栈指示器的装入操作。

例 4-38 使用 SPHL 指令

程序一	程序二
LXISP	LXIH
300	300
012	012
	SPHL

我们使用这条 SPHL 指令，很容易把不同的 16 位数值装入堆栈指示器，以便满足具体应用的需要，即使包含 SPHL 指令的程序存储在只读存储器（ROM）里。你能编写一个程序，用来把存储在两个连续存储单元的一个 16 位地址装入堆栈指示器吗？

如果使用例 4-39 这个程序，你能把这个所需要的 SP（堆栈指示器）地址存入两个连续存储单元；具体地说，存储在 020234 和 020235（109 C 和 109 D）这两个连续存储单元。堆栈里的这个 16 位地址的低 8 位应该存储在 020 234(109 C) 存储单元，高 8 位地址应存储在 020 235（109 C）存储单元。

现在，读者已经看到了 8080 的许多新的指令，它们把寄存器H和L作为寄存器对使用。我们把这些新的指令进行归纳，列入表 4-9。

例 4-39 把 SP 装入 ROM 中的程序

```
SETSP, LHLD/把 020 234 和 020 235(109 C
       234   /和 109 D）的内容装入寄存器
       020   /对H。
```

表 4-9　　寄存器对 H 用指令一览表

LXIH	XTHL
⟨B_2⟩	XCHG
⟨B_3⟩	
	LHLD
INXH	⟨B_2⟩
DCXH	⟨B_3⟩
	SHLD
DADSP	⟨B_2⟩
DADH	⟨B_3⟩
DADD	PCHL
DADB	SPHL

```
SPHL /现在，把寄存器对H的内容
     /装入堆栈指示器。
```

A 寄存器的附加指令

还有两条单字节指令，用来对A寄存器的内容进行操作；处理进位的指令也有两条，也是单字节指令，这几条指令，我们将在本章予以讨论。处理A寄存器的内容的指令之一叫 CMA 指令，即A寄存器内容的取反码指令，它的操作码是 057 (IF)，当 8080 执行这条指令时，这条指令使A寄存器的内容按位取反，即转换。8080 执行例 4-40 的两条指令之后，A寄存器的内容是什么呢？

为了弄清 8080 执行 CMA 指令之后，A寄存器的内容是什么，我们可以把A寄存器的内容看成一个 8 位二进制数的字。因此，127 (57) 等于二进制数 01010111。8080 执行 CMA 指

令时，把A寄存器的内容取反，成为10101000(八进制数250，十六进制数A8)。应用CMA指令的比较复杂的程序如例4-41所示。

例 4-40 使用CMA指令的简单程序例。

```
    .
    .
MVIA/把127这个数值装入A寄存器里，
127   /(八进制数127＝十六进制57)。
CMA  /将A寄存器的内容取反。
      .
      .
```

例 4-41 另一个使用CMA指令的程序

```
USECMA, LXIB    /把012 000(十六进制0A00)这个数
        000     /装入寄存器对B。
        012
        DCRB    /B寄存器的内容减1。
        MOVAB/把B的内容送到A。
        CMA     /A寄存器的内容取反。
         .
         .
```

在使用CMA指令的例4-41中，当8080执行这条LXIB指令时，把012（0A）这个数装入B寄存器。然后，8080执行DCRB指令时，将B寄存器的内容减1，成为011（09)。取反操作完毕之后，A寄存器存储的内容是366（F6)。用二进制数表示，其结果如下：

00001001 取反变为11110110

该取反结果等于A寄存器内容的1的反码。它叫做1的反码，因为“1”取反成了“0”，而“0”取反成了1。算术运算经常需

要采用二进制数的补码。一个数的 2 的补码就是，当它与原来的数相同时，产生的结果等于 0 的数。例如，二进制数00000011 的 2 的补码是 11111101。实际上， 2 的补码比 1 的补码大 1，也就是说， 2 的补码等于 1 的补码加上 1。例如：

```
  00000011···二进制数，(3)
+ 11111101···二进制数，(3) 2 的补码
1 00000000···结果是 0
```

8080 的 CMA 指令使存储在A寄存器的那个数产生 1 的补码。你能够编写一个程序，使存储在A寄存器的数能产生 2 补码吗？这个程序非常简单，如例 4-42 所示。

随着例 4-42 的那些指令被执行，如果你可以检查A 寄存器的内容，那么你会看到：

A寄存器	执行指令后
× × × × × × × ×	•
× × × × × × × ×	•
× × × × × × × ×	MVIA
00000011	003
11111100	CMA
11111101	INRA
11111101	•
11111101	•

例 4-42 产生某个数的 2 的补码

```
•
•
MVIA/把八进制数 003 (十六进制 03)
003   /装入A寄存器。
CMA  /A的内容取反。
```

INRA /A的内容加 1。

•

•

另外，还有一条指令，用来对A寄存器的内容进行操作，它就是 DAA 指令。但是，这条指令我们留在下一章进行讨论。

进位指令

有关进位的指令有两条，它们是进位置位指令和进位取反指令。当 8080 执行进位置位指令时，它把进位置为逻辑 1。如果进位置位指令执行之前，进位是逻辑 0，那么，进位置位指令执行之后，进位将置为逻辑 1。如果在指令执行前进位已经是逻辑 1，那么，这条进位置位指令将对进位的状态不产生影响。

进位取反指令对进位的状态取反。如果进位是逻辑1，8080 执行这条进位取反指令后，进位取反，变为逻辑 0。如果进位是逻辑 0，那么，对它进行取反操作后，变为逻辑 1 状态。

进位置位指令，即 STC 指令，其操作码是 067(37)。进位取反指令，即 CMC 指令，其操作码是 077(3F)。这两条指令都是单字节指令，它们都影响到进位标志位的状态。

例 4-43 所列出的这个程序有一个子程序，它使用了一条 STC 指令，以便把已经出现的一个错误告诉主程序。

在这个例子中，这个 TEST 子程序把寄存器对 H 寻址的存储单元的内容传送到A寄存器，然后，把A寄存器的内容与立即数字节 233 (913) 进行比较。如果从存储器读出的这个数等于 233 (9B)，则 8080 微处理器从子程序返回主程序。作

为比较的结果，进位是逻辑 0 。

如果 8080 从存储单元读出的数不是 233 （9 B），那么，8080 把进位标识位置逻辑 1，然后从这个子程序返回主程序。当 8080 从子程序返回时，它要测试进位标识位的状态。如果进位标识位是逻辑 1，它表示存在着错误状态，那么，8080 调入 ERROR 子程序。否则，8080 跳过条件调用指令 CC，然后去执行存储在三个字节的 CC 指令后的那条指令。

我们把 STC 指令和 CMC 指令作为逻辑指令来考虑。因此，当 8080 执行这两条指令时，进位标识位的状态反映出这两条指令的执行结果。

CMC 指令是一条单字节指令，可以在算术程序中使用。假设你需要保证：8080 在执行程序的某一部分之前，进位标识位是逻辑 0。由于 8080 并没有进位清除指令，那么，你能够编写一个程序来完成这项任务吗？

例4-43 用这条 STC 指令来标出错误状态

```
    •
    •
CALL  /调用这个测试程序。
TEST
0
CC      /如果进位标志是逻辑 1，
ERROR/则调用 ERROR 子程序。
0       /否则，跳过这个程序。
    •
    •


TEST,  MOVAM /取存贮器的内容到A寄存器
       CPI     /把它与立即数据字节
```

```
233     /233（十六进制9B）比较。
RZ      /如果A的内容是233（十六进制9B），
STC     /则返回。如果A不是233(9 B)，则
RET     /把进位置逻辑1。现在返回到
        /主程序。
```

例 4-44 清除进位

```
        .
        .
    STC
    CMC
        .
        .
```

不论进位标识位的现行状态如何，这个简单的程序把进位标识置逻辑1，然后把它进行取反操作，变成逻辑0。如果你不用例4-44这个指令序列，你也可以执行一条单字节指令把进位清零，并且也能把A寄存器的内容清除为全零。这条指令是什么呢？如果8080执行这条XRAA（A寄存器的内容与A寄存器的内容异或）指令，A寄存器的内容和进位将被清除为逻辑0状态。SUBA（从A寄存器的内容减去A寄存器的内容）指令也有着相同的作用。

最后的结论

在本书所列出的许多子程序中，在程序的开头，我们没有引入压入堆栈指令，在子程序的末尾，也没有编入弹出堆栈指令。如果引入了压入和弹出指令，就会使得许多子程序表更长，即使我们已经在子程序的开头引入了压入指令，在末尾引

入了弹出指令，那么，你也会学不到什么东西。当然，如果在你自己的程序中，你需要使用本书中所给出的子程序，那么，你可以在这些子程序上增添压入指令和弹出指令。但是，有些子程序使用了条件返回指令。读者在给子程序增添上压入指令和弹出指令时，必须十分注意。实际上，如果增添了压入指令和弹出指令，常常要把条件返回指令改换成完成同样任务的其它指令。例如：

```
                        TESTA,  PUSHD
                                PUSHH
                                INRA
                                JNZ
                                ANOTO
TESTA,  INRA                    O
        RI                      POPH
        INXB 改变成：           POPD
        RET                     RET
                        ANOTO,  INXB
                                POPH
                                POPD
                                RET
```

读者知道，增添压入指令和弹出指令会增加简单的子程序的复杂性。当然，在这个具体的例子中，为什么一定要把寄存器对D和H的内容保存在堆栈里，这是没有确切的理由的。

到现在为止，我们已经讨论了8080微处理器用的244条指令。本书的其余章节将讨论如何把这些指令应用于各种软件问题。

第五章　算术子程序

我们从实验或从外部设备获得的数据，通常要把它们从一种测量单位转换成另一种测量单位。这就是说，用“磅/每平方英寸”所表示的压力，我们可以把它转换为毛（等于1毫米水银柱的压强），或者用英尺～磅表示的转矩必须换算成牛顿米。为了完成这类换算，我们可以使用算术例行程序。正如你已经所见到的那样，8080微处理器经常操作8位的数据字。但是，你还看到过，8080能够执行一些指令，处理16位的数据。这些指令包括LXZH，DCXB，INXD和DADH四条指令。

我们首先要讨论的内容是算术例行程序和算术子程序，用它们来处理定点数。所谓定点数就是小数点在固定的位置上，不管被运算的数的大小，都是如此。与定点数有关的指数是没有的。在本章的后面，我们将要简要地讨论浮点数；浮点数一定有指数。

整　数　加

8080微处理器能够执行的最简单的加法运算是两个8位二进制数相加。它必须把一个8位数存储在A寄存器，作为数据源。要相加的另一个数的源可以是：（1）任何一个通用寄存器（A～E，H和L）的内容，（2）寄存器对H寻址的存储单元的内容，（3）立即数字节。第二章的两个例子（例2-13和

例 2-15），已经告诉我们如何才能把两个 8 位二进制数加到一起的过程。第二章还给出了一个例子（例 2-17），该例示出了怎样用 ADD 和 ADC 这两种指令来把寄存器对 B 的 16 位数据加到寄存器对 D 的内容上的方法。在第四章中，我们已经讨论了 DAD 这种指令。这类指令可以用来把任何一个寄存器对的内容加到寄存器对 H 的内容上。例 4-5 示出了一个 16 位数的加法操作，那是**双精度**的加法，它把寄存器对 H 的一个 16 位数自加了数次。8080 微处理器也能够把 24 位，32 位或更长的数相加。我们通常把大于 8 位的数进行加法运算的程序叫做**多精度**程序或**多字节**程序。

假设我们要把两个 32 位数（4 个字节）加在一起，那么，8080 微处理器没有足够用的寄存器来存储这么多的信息（8 位字节）。我们怎样才能用 8080 完成这两个 32 位数的加法运算呢？我们可以把一个 32 位数（4 个 8 位字节）存入存储器，可以把另一个 32 位数存入 B、C、D 和 E 这四个寄存器。然后，我们可以用寄存器对 H 来给存储 32 位数的一个存储单元寻址。例 5-1 所列出的这个子程序，把四个连续存贮器单元的内容，按顺序分别加到 E、D、C 和 B 这四个寄存器的内容上；四个连续存储单元是从符号地址 DATA 开始的。8080 微处理器在访问这个子程序之前，一定要把一个需要相加的 32 位数装入 B、C、D 和 E 这四个寄存器，每个寄存器装入一个 8 位字节；还必须把另一个 32 位数装入四个连续存储单元。当 8080 从这个子程序返回时，这两个 32 位数相加的结果存放在 B、C、D 和 E 这四个寄存器里。

这个结果的高位有效字节（MSBY）和低位有效字节（LSBY）存储在什么地方呢？8080 从这个子程序返回时，把高位有效字节存储在 B 寄存器；低位有效字节存储在 E 寄存器。

请读者注意这种情况：8080 执行了 MOVBA 指令后，如果进位是逻辑 0，才从这个子程序返回。如果进位是逻辑 1，8080 不会执行 RNC 指令。相反 8080 将开始执行在 ERROR 这个符号地址的几条指令。符号地址 ERROR 的几条指令为什么包含在这个子程序中呢？假设 000 000 000 001(00000001) 这个数要加到 377 377 377 377(FFFF-FFFF) 这个数上面。那么，这两个数相加的结果应该是什么呢？进位应该是逻辑1；B、C、D 和 E 这两个寄存器对的内容都应该是 000(00)。当然，这两个数相加的结果不是 000 000 000 000(00000000)。因此，在这个子程序中必须包括符号地址的几条指令，以便用来保存相加的结果的最后进位，或者告诉 8080 微型计算机的操作员，这两个数相加的结果所需要的存储器空间比被分配用以存储相加的结果的空间大。在这个例子中，我们分配了四个寄存器（32 位）来存储相加的结果，但是，其结果确实是 33 位，其中包括 1 位进位。

很可惜，例 5-1 所示的那个子程序是很不灵活的。为了把字长为三个字节的两个数相加，应该把 32 位字长的两个数的高位有效字节都置成 000(00)。如果我们需要把字长是 5 个字节的两个数相加，那么，我们必须重新编写一个子程序。在例 5-1 中，需要一条 ADD 类指令和三条 ADC 类指令，才能把字长为四个字节的两个数相加。为了把字长为 5 个字节的两个数相加，是不是还需要增添一条 ADD 或 ADC 类的指令吗？应该增添一条 ADC 类指令。这个问题的最好的解决办法，是编写一个通用子程序，根据情况的需要，这个程序可以把或大或小的数相加。

例 5-1 32 位加法子程序。

```
/这个子程序用来把四个连续
```

```
/存储单元的内容，加到 B、C、D
/和 E 这四个寄存器所存储的
/一个 32 位数上。E 寄存器存储
/最低有效位字节，象
/存储地址的最低存储单元
/也存储最低有效位字节
ADDIT, LXIH    /把地址装入 H 和 L 寄存器，指示到
       DATA    /存入存储器的四个数据字。
       0
       MOVAM /取第一个 8 位字，
       ADDE    /将 E 寄存器的内容与它相加。
       MOVEA  /然后把相加的结果存入 E。
       INXH    /H 和 L 加 1，指示到下一个
               /8 位字。
       MOVAM /取第二个 8 位字。
       ADCD    /把 D 的内容和进位与它相加。
       MOVDA  /再把结果存入 D 寄存器。
       INXH    /H 和 L 加 1，指示到第三个
               /8 位字。
       MOVAM /把第三个字送到 A 寄存器。
       ADCC    /把 C 的内容和进位与它相加。
       MOVCA  /结果存入 C 寄存器 。
       INXH    /H 和 L 加 1，指示到最后一
               /个字。
       MOVAM /把最后一个字送到 A 寄存器。
       ADCB    /把 B 的内容和进位与它相加。
       MOVBA  /把结果的最高有效字节
       RNC     /存入 B 寄存器。如果进位
ERROR  .       /是 0，则返回。
       .       /如果进位是 1，加操作
```

```
        .       /的结果比 32 位（4个字节）大。
        .       /所以，执行一些“ERROR 指令”。
        RET     /然后返回。
DATA,   207     /这四个数字节
        051     /都存储在这儿
        372     /的存储单元。
        165
```

在 0.200 秒的延时子程序（例 3-4）表明：我们可以把自变数(执行 0.200 秒循环的次数)传递给这个子程序，我们也可以使用同样的方法来“告诉”多精度相加操作的子程序，将要被加的某个数有多少个 8 位数据字节。因为数据字节的数目将是可变的，所以按顺序把它们存储在存储单元时，有利于把两个数加在一起；因为在某些情况下，可能没有足够用的通用寄存器来存储一个加数的全部字节。所以，我们一定要寻找某种方法，同时存取两个不同的存储区，因为我们必须把要相加在一起的每一个数存储在存储器的两个不同部分。在存储器的两个不同部分同时进行存取的例子，我们已经给出了吗？回答是肯定的。我们在讨论用 STAXD 和 LDAXD 这两条指令，把数据值从存储器的一个部分送到另一个部分的问题时，已经给出了例子（例 4-1）。进行多精度的数加法操作时，数据传送的框图如图 5-1 所示。

这个程序将必须从存储器的一个部分取一个数据值，然后把另一个存储单元的内容加到这个数据值上，再把这两个数相加的结果存回到存储器里。这个结果正好存储在刚才被相加的一个数的存储单元。可以用来把这些可变字长的数相加的子程序如例 5-2 所示。

在例 5-2 所示的子程序中，我们用寄存器对 H 来为存储一

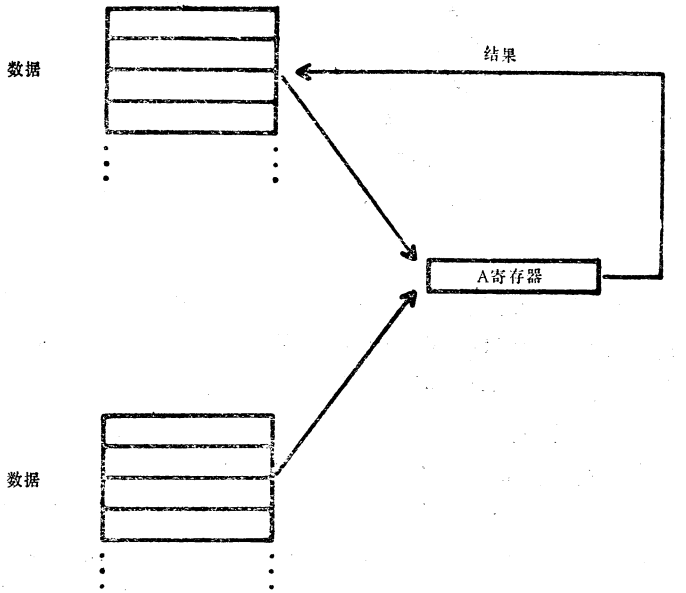


图 5-1　多精度字相加时，数据的传送

个加数的存储区寻址；我们用寄存器对D来给存储另一个加数的存储区寻址。把这两个数相加的结果，存储在寄存器对H所寻址的存储单元。请读者注意，为了对两个最低有效字节进行加法运算，在这个子程序的开头，仍然一定要使用一条ADD类指令，这是因为在第一次把两个最低有效字节进行加法运算时，未必涉及到进位。请读者记住，当用寄存器对1为数据传送指令寻址时，数据源（目的）必须是A寄存器。因此，在可以进行算术操作之前，必须把一个8位字节从由寄存器对D寻址的存储单元送到A寄存器里，然后利用这条ADDM或ADCM指令，把寄存器对H寻址的存储单元的内容加到A寄存器的内容上。再把这两个数相加的结果存入寄存器对H所寻

址的存储单元。因为寄存器对H所寻址的这个存储单元是数据的源和目的，所以，原先存储在这些存储器单元的数被加法运算的结果的一个字节代替。要是用稍有不同的方法进行加法运算，那么所得到的结果也是相同的。

例 5-2　多精度加法子程序

```
/我们必须按照下面的条件送入这个子程序。
/寄存器对H指示到一个数据字的最低有
/效字节；寄存器对 D 指示到另一个数据
/字的最低有效字节。把两个数据字的
/最低有效字节存储在最低存储单元。
/C寄存器的内容等于要被加的字节的数目。
/(十 进制数 1-256)。
START,  LDAXD  /把 D 和 E 作为指示器，取第
               /一个字节。
        ADDM   /把H和L寻址的存储单元的内
               /容与它相加。
STRT 1, MOVMA  /然后，把结果存回存储器。
        INXH   /指示器加 1，指示到这两个
        INXD   /数的下一个有效字节。
        DCRC   /字节计数器减 1。
        JZ     /字节都已被取完了吗？是的。
        CHKCRY/然后检查"溢出"错
        0      /的进位
AGAIN,  LDAXD  /否则，利用 D 和 E 作为指示器，
               /取下一个字节。
        ADCM   /带进位加存储单元的内容。
        JMP    /然后，存储该结果，使存储
        STRT1  /器地址加 1，字节计数
        0      /器的内容减 1。
```

```
CHKCRY, RNC     /如果进位是逻辑 0，则从
                •       /该子程序返回。否则，告诉
                •       /主程序，结果所需要的存储空
                •       /间比所分配的要大，
                RET     /然后，从该子程序返回。
```

我们如果使用例 5-3 所示的指令，那么这个结果应该存储在寄存器对D所寻址的存储单元，而不应该把结果存储在寄存器对H所寻址的存储单元，如例 5-2 所示。例 5-3 这个子程序的缺点是：它还需要一条指令才能完成同样的任务；更重要的是，它还需要使用另一个寄存器，即B寄存器来暂时存储一个要被加的数。

例 5-3 用寄存器对D源和目的存储器地址

```
•
•
MOVBM/把存储单元的内容装入 B 寄存器
LDAXD/把 D 和 E 寻址的存储单元的内容装入
      /A。
ADDB  /把 B 的内容加到A的内容上。
STAXD /把相加的和保存在寄存器对 D
  •   /所寻址的存储单元。
•
```

例 5-2 示出的这个子程序唯一的局限性或许是该程序使用了两条不同的加法指令。8080 调入一次这个子程序时，这条 ADDM 指令就执行一次，这是因为两个最低有效字节相加时，前一次加操作并没有产生进位的缘故。其它的加法操作都是由 ADCM 这条指令来执行的。然而，如果首先把进位清零，那么，8080 可以只使用 ADCM 这条指令，从而简化了这个子程序的循环，如例 5-4 所示。

8080 微处理器调入例 5-4 的子程序时，它将 A 寄存器和其本身的内容进行逻辑“异或”操作。该操作会把A寄存器的内容清为零。而且，更为重要的是，它还能将进位清除成逻辑零。因此，在这个子程序中，用了一条 XRAA 指令，所以，这条 ADCM 指令便成了唯一的所需要的加法操作指令。

8080 第一次调用这个子程序时，XRAA 指令使进位清零，所以，ADCM 指令能够把进位零加到两个最低有效字节的和上面。当然，8080 第二次通过循环时，把进位（逻辑 1 或逻辑 0）加到第二次加操作的结果上面，这是因为 8080 的程序执行控制现在转移到 STRT 1，而不是转移到 START（在 START，进位应该被清零了），8080 调入这个子程序来做加法运算时，能够进行运算的最大的数的字长为 256 个字节。为什么这个子程序把加法运算的数限制在这个长度呢？这是因为只有一个 8 位寄存器，即只有C寄存器作为字节计数器的缘故。但是，这个子程序仍然能够接收处理最大的数（2^{2048}，即字长是 2048 位的数；$2^{333}>1\times10^{100}$）。

例 5-4 从这个子程序(例 5-2)删去 ADDM 指令

```
/用下面的条件输入这个子程序。
/H和L指示到一个数的最低有效字节。
/D和E指示到另一个数的最低有效字节。
/这两个最低有效字节被存储在最低存
/储单元。C＝要加的字节的数目(1～256)。
START,  XRAA    /把A和进位清零。
STRT 1, LDAXD   /从D和E寻址的存储器单元
                /取一个字节
        ADCM    /加H和L寻址的存储器单元
        MOVMA   /的内容，然后把该结果存回
        INXH    /存储器。把两个数据缓冲器
```

```
        INXD    /指示器减1。
        DCRC    /字节计数器加1。
        JNZ     /全部字节都加过了吗？没有。
        STRT 1  /然后加下两个存储单元
        0       /的内容。
CHKCRY, RNC     /如果进位是逻辑0，
        •       /则返回。否则告诉主程
        •       /序，结果比分配的存储
        •       /空间大
        RET     /然后从这个子程序返回。
```

为了把一个字长为256个字节的数做加法运算，8080调入例5-4所示的START子程序时，C寄存器的内容必须是什么呢？C寄存器的内容必须是000(00)。把这两个数的最低有效字节相加并存入存储器以后，C寄存器的内容从000(00)减1到377(FF)。因为这个结果并不等于零，所以8080要把START这个子程序的循环再执行255次。这就是说，这个循环中的各条指令需要执行256次。

请读者注意，C寄存器的内容被逐渐减为0时，在这个子程序的末尾的几条指令测试进位是否是逻辑1状态。如果进位是逻辑0，8080从该子程序返回。如果进位是逻辑1，则8080必须执行其他一些指令，或者保存最终的进位，或者告诉8080微型计算机的操作员：被存储在存储器的结果是不正确的结果。

例5-4这个程序的唯一缺点是，存储结果的存储单元与存储一个被参与加法操作的数的存储单元相同。最好把结果存储在第三个存储区，这样才不至于使加数"丢失"。因此，寄存器对B，D和H都用来保持存储单元地址。这些地址是用来存储要参与加法运算的两个数的存储单元的地址，以及存储加法运

算的结果的存储器单元的地址。即使如此，仍然需要一个字节计数器——例 5-4 所示的子程序所用的字节计数器是 C 寄存器。只有A寄存器没有用来保持一个地址。但是，A寄存器不能用来保持字节计数数字，因为它要用来做两个数的加法运算。我们怎样才能编写这个改进的子程序呢？

当 8080 微处理器调用例 5-5 所示的子程序时，不仅必须分别把三个地址装入寄存器对B、D和H里，而且还必须把字节计数数装入A寄存器里。当 8080 开始执行这个子程序时，它必须立即执行 PUSHPSW 指令把 A 寄存器的内容和标识位压入堆栈。8080 执行了这条 PUSHPSW 指令后，A 寄存器的内容和进位清零。然后寄存器对D寻址的存储单元的内容移入A寄存器，再把寄存器对H寻址的存储单元的内容加到A寄存器的内容之上，并且把相加的结果存入寄存器对B寻址的存储单元。随后，8080 把三个存储地址都加 1。然后这条 XTHL 指令使寄存器对H的内容与堆栈上的最后两个字节进行交换。堆栈上的最后两个字节是什么呢？

例 5-5 使用三个存储地址的多精度相加的子程序

```
/根据下述条件送入这个子程序。
/H和L指示到第一个数的最低有效字节。
/D和E指示到第二个数的最低有效字节。
/把相加的两个数的最低有效
/字节存入最低的存储单元。B和C寄存器指示到用来存储结果的存储器
 单元。
/把要加的数据按八位字节逐次送入A寄存器。

                    /首先把字节计数数保存在A，
START,  PUSHPSW /然后把标识字存入堆栈。
        XRAA    /把A寄存器和进位清零。
```

```
AGAIN,  LDAXD   /取要被加的数的一个字节。
        ADCM    /把另一个字节和进位与它相加。
        STAXB   /然后，把其结果存入存储器里。
        INXH    /这三个存储地址都加 1 。
        INXD
        INXB
        XTHL    /把H和L的内容与栈顶的内容交换。
        DCRH    /(H＝A＝字节数)。使字节的数目减 1。
        XTHL    /恢复H和L、A和标识位。
        JNZ     /H(其实是A)的内容减 1，
        AGAIN   /成为 000 吗？ 如果不是，把另外
        0       两个字节相加。
        JC      /把各个字节都加完后，
        CHKCRY  /检查来自最后一次加
        0       /的"溢出"的进位。
        POPPSW  /进位是零，则从堆栈弹出 PSW，
        RET     /然后返回。
CHKCRY  .       /进位是逻辑 1，
        .       /所以，告诉主程序，结果所需要的
        .       /存储空间比所分配
        .       /的存储空间大。
        POPPSW  /然后，从堆栈弹出程序状态
                /字(PSW)，
        RET     /并从子程序返回。
```

当 8080 微处理机调用这个子程序时，它把一个返回地址压入堆栈。例 5-5 所示的子程序的第一条指令是 PUSHPSW，所以堆栈上的最后两个字节是 A 寄存器的内容和各标识位。8080 执行 XTHL 指令时，把寄存器对H的内容存入堆栈，把A寄存器的原来的内容装入H寄存器，从堆栈把各标识位装入

L寄存器。当8080执行DCRH指令时，H寄存器的内容，现在是字节计数器减1。8080将H寄存器的内容减1以后，寄存器对H的内容和堆栈上的最后两个字节再进行交换。这时，好象在堆栈和寄存器对H之间并没有发生什么操作似的，因为这两条XTHL指令没有**净效应**。但是8080执行DCRH（使字节数减量）指令的结果是由零标识位的状态来表示的。然后，8080执行JNZ指令，测试零标识位。如果计数字节的内容还不是零，则JNZ-AGAIN将被执行。

最后，字节计数器减到零时，8080不再执行JNZ-AGAIN的指令。这就是说，8080把多精度数的各个字节都已经加完了，所以，它需要从该子程序返回到主程序。但是，8080从这个子程序返回之前，必须测试进位标识位的状态，以便判断来自两个最高有效字节的加操作最后是否有进位。最后如果没有进位，那么，8080不执行从JC到CHKCRY的指令，而执行POPPSW指令，从堆栈弹出A寄存器的内容和各个标志位，然后执行RET返回指令。8080为什么要执行POPPSW指令呢？实际上我们并不关心从堆栈弹出的装入A寄存器的数值和各标志位。但是，如果8080没有执行这条POPPSW指令那么这条RET指令就把堆栈上的最后这两个字节作为一个返回地址使用。这样，就会在某个未知的地址上继续执行这个程序，把真正的返回地址保留在堆栈。因此，使弹出指令与压入指令一样多，或者**使压入指令和弹出指令一样多**，这是十分重要的。**堆栈操作并不是不重要！**

把两个最高有效字节（MSBY）进行相加以后，如果进位是逻辑1，那么，8080应该执行JC-CHKCRY的指令。在CHKCRY这个标号上，还需要一些指令来保存最后一位进位，或者告诉8080微型计算机的操作员，进位是逻辑1状态，存

储在存储器里的结果是不正确的。

我们把字节计数器临时存放在H寄存器，计数器可以减1而不影响进位标识位，该标识位只反映了最后一次进行加操作的结果，并且必须把它保存起来，供8080执行下一条ADCM指令处理，读者会问，这是什么原因呢？DCRH指令不影响进位标志的唯一的原因是：减1指令对于其他各个标识位都有影响，唯独进位标识位除外。读者要知道这个问题的唯一办法是，查阅8080生产厂的产品数据表，以弄清楚DCR类指令的特性。这是对读者提出的重要一点。当你要使用一条新的指令时，你应该查阅8080的技术性能说明书，确切地弄清楚这条指令的作用，而且8080执行这条指令时，该指令影响还是不影响哪些标志位。

到现在为止，你已经看到了有关多精度加法操作的许多子程序，你可以用这些子程序来解决你自己的软件设计的许多问题。关于如何把已经从这些加操作子程序所学到的知识应用于编制多精度减操作的子程序的问题，我们将在本章的下面一节讨论。

整 数 减

在第二章，我们给出了减法运算的许多程序实例；例如，例2-19，例2-21，例2-23，例2-25。这些例子告诉我们怎样能够把单精度的数和双精度的数相减。但是，由于进行减法操作的数越来越大，8080微处理机的通用寄存器将不能够存储要进行减操作的数，相反，通用寄存器将可以用来存储器要进行减操作的两个数的存储地址。

在例 5-2 中，8080 必须执行一条 ADDM 和一条 ADCM 指令，才能不把进位的内容加到两个最低有效字节（LSBY）的和上。例 5-4 所给出的子程序包含了尽量少的几条指令，因为 8080 执行一条 XRAA 指令，把进位位清为零。为了把执行多精度减操作的子程序（例 5-6 ）的进位清零，也可以执行一条 XRAA 指令。

例 5-6 多精度减法子程序

```
/按照下述条件送入这个子程序。
/H和L寄存器指示到一个数的最低有效字节。
/D和E指示到另一个数的最低有效字节。
/两个数的最低有效字节存储在
/最低存储单元。C＝要减的数
/的字节数(1-256)。
MSUB，XRAA      /把 A 寄存器和标识位清零。
MSUB1，LDAXD    /从D和E寻址的存储单元取一个字节。
        SBBM    /减去H和L寻址的存储单元的内容，
        MOVMA   /然后，把结果存回到存储器里。
        ZNXH    /两个数据缓冲指示器
        INXD    /加1。
        DCRC    /字节计数器减1。
        JNZ     /一个数的全部字节已被减完了吗?
        MSUB1   /没有，所以减去下两个存储器
        0       /单元的内容。
CHKCRY，RNC     /如果进位是逻辑0，则返回。
        •       /否则，告诉主程序，
        •       /最后已经出现了
        •       /借位。
        RET     /然后从该子程序返回。
```

当 8080 调用 MSUB 子程序后，执行这条 XRAA 指令时，

把A寄存器的内容和进位清零。然后把寄存器对D寻址的存储器单元的内容装入A寄存器。再从A寄存器的内容减去寄存器对H所寻址的存储单元的内容。8080第一次执行SBBM指令时，进位是逻辑0，所以进位标识的状态将不影响这两个最低有效字节(LSBY)相减的结果。如果8080再次执行MSUB1这个循环，那么将从A寄存器的内容减去进位；此外，该进位将会反映出是否已经出现借位。减操作被执行之后，该结果存储在寄存器对H所寻址的存储单元。这个结果字节写入刚从A寄存器的内容减去的那个数)。8080把该减操作的结果存入存储器后，把寄存器对D和H中的存储地址加1，字节计数器减1。这时，8080做出了判定，指明是否将再执行MSUB1这个循环。8080执行DCRC指令之后，如果字节计数器的内容不是零，那么，8080则执行JNZ-MS-UB1。如果两个最高有效字节的减操作并没有产生借位，且字节计数器被减为0，那么，8080将从该子程序返回主程序。如果进位是逻辑1，C寄存器的内容被减为零时，8080将不执行RNC指令。相反，8080将执行ERROR开始处的指令。如果确实出现了一位借位，则是从较小的数减去了较大的数。因此，可以用ERROR标号上的指令把一个自变量传回给主程序，或者告诉8080微型计算机的操作员：已经出现了一位借位。

多精度加法操作的子程序(例5-5)所采用的技术，同样也可以适用于多精度减操作的子程序。这样，减操作的结果可以保存在第三个存储区，所以，被减数和减数都能够保留下来。为了完成这一操作，寄存器对D存储一个存储地址，这个地址指向存储在存储器的被减数，寄存器对H指到减数，寄存器对B指到结果。再把A寄存器作为字节计数器用，要进行减操作的两个数的字节的数目存储在这个寄存器。例5-7所列出的子程

序归纳了这些特征，这个多精度减操作的子程序和多精度加法操作的子程序(例 5-5)只有一条指令不同：前者用了一条SBBM指令，而后者用了一条 ADCM 指令。

当 8080 调入例5 -7 所示的子程序时，A寄存器 必须 已经存入了参与减操作的数的字节的数目；寄存器对B、D和H必须已经存储了结果、被减数和减数的存储地址。PUSHPSW 是这个子程序的第一条指令，把存储在A寄存器里的字节数存入堆栈。然后，8080 执行 XRAA 指令，把A寄存器 和 进位清零，再把寄存器对D寻址的存储单元的内容送到A寄存器里，而后，又从A寄存器的内容减去寄存器对H所寻址的存储单元的内容。把减操作的结果 存储在寄存器 对B所寻址的 存储单元，然后，把这三个存储单元地址都加 1。

8080 执行 XTHL 指令，将堆栈上的最后两个字节（A 寄存器的内容和标识字）与寄存器对H的内容进行交换。因此，8080 执行这条 XTHL 指令 之后，寄存器 对H的 地址存入 堆栈，数值出现在A寄存器，标识位在寄存器对H里。这样，我们就能够使用堆栈存储单元代替寄存器。现在，字节数存贮在H寄存器里，当这条 DCRH 指令被执行之后，字节数被减 1。当 8080 执行第二条 XTHL 指令时，又把字节数与标识位字节一起放回堆栈。同时，8080 执行这条 XTHL 指令，把原来存储在寄存器对H的 地址取回。然后 8080 执行 TNZ～AGAIN 的指令，测试零标识位（用 DCRH 指令使零标识位置位 或清零)。

如果字节数还没有被减到零，那么，8080 返回到 AGAIN。这就使另外两个字节进行减操作。如果字节数被减为零时，那么，用程序指令来测试进位标识位。如果进位标识是逻辑 0，表示两个最高有效字节(MSBY)相减后，并没有出现借位，那

么，从堆栈弹出A寄存器的内容和标识字，然后8080返回到主程序。如果进位标识是逻辑1,这就是说，两个最高有效字节(MSBY)相减时出现了一位借位,则8080执行JC～CHKCRY。我们可以使用这些指令来告诉主程序：已经从一个较小的数减去了一个较大的数。

这个子程序(例5-7)不能用下列指令序列来结尾，你知道这是为什么呢？

```
•
•
XTHL
DCRH
XTHL
RZ
JMP
AGAIN
0
```

读者应该认识到，两个最高有效字节相减以后，这个指令序列并不检查借位，而且，在返回指令被执行之前，这些指令也不"清理"堆栈。当8080调用这个子程序时，A寄存器的内容和标识字被压入了堆栈。所以，在任何一条返回指令被执行之前，必须把这些数据值从堆栈弹出。如果不把这些数据从堆栈弹出来，当8080执行RZ指令时，会把它们作为一个16位的返回地址使用。这样，就会引起程序返回到程序中的一个未知点。

例 5-7 存取三个不同存储部分的多精度减子程序

```
/按照下述条件编写这个子程序，
/H和L寄存器指示到减数的最低有效字节；
/D和E寄存器指示被减数的最低有效字节；
```

```
/减数和被减数的最低有效字节被
/存储在最低存储单元。B和C寄存器指示
/存储结果的地址。然后
/相减的8位字节的数目转入A寄存器。
                    /首先把字节计数数存入A寄存器里，
START, PUSHPSW      /然后把标识字存入堆栈。
      XRAA          /把A寄存器和进位清零。
AGAIN, LDAXD        /取被减数的一个字节。
      SBBM          /减去减数的一个字节。
      STAXB         /然后把结果保存在存储器里。
      INXH          /把三个存储器地址都加1。
      INXD
      INXB
      XTHL          /把H和L寄存器的内容与栈顶的内容交换。
      DCRH          /(H=A=字节数)。字节数减1。
      XTHL          /恢复H和L、A和标识字。
      JNZ           /H(其实是A)减1，变为000了吗?
      AGAIN         /如果没有，那么
      0             /减另外两个字节。
      JC            /如果进位是1，则
      CHKCRY        /从较小的数减去了较大的数，
      0             /所以，告诉主程序：
      POPPSW        /进位是逻辑0，所以从堆栈弹出PSW，
      RET           /然后返回主程序。
CHKCRY  ·/进位是逻辑1，所以打印一个错误信息，
        ·
        ·/或者执行其他操作。
        POPPSW  /然后，从堆栈取PSW，
        RET     /然后从该子程序返回。
```

多精度加和多精度减程序和子程序都遵循一条简单的规则：**当多字节的数相加或相减时，必须首先加或减这数的最低有效字节（LSBY）**。这条规则在基本指令这一章已经应用过。当这样做时，进位可以被加到这个数的较高有效字节上，（或者从这个数的较高有效字节上减去借位）。

在本章的下一节，我们将要讨论单精度和多精度的乘法子程序。这些子程序比我们在本章和前面各章已经讨论过的加法程序和子程序，减法程序和子程序复杂得多。但是，用来执行乘法操作的方法和我们已经知道的方法非常类似。

整 数 乘

大多数程序设计员认识到：数的乘法运算可以用连续的加操作来实现；数的除法操作可以用连续的减法操作来实现。例5-8是个简单的乘法操作的子程序。这个子程序被用来将两个16位的数相乘，产生一个16位的结果。这个子程序就是通过连续的加法操作来完成两个数相乘的运算的。

一般地说，象例5-8这样的乘法子程序是不能使用的，因为它们可能需要较长的程序执行时间。我们假设需要把2046和3这两个数相乘，如果用连续的加法子程序来进行这两个数的乘法运算，那么，其中一个数必须用作计数器，来跟踪进行连续加的数。例如，我们可以把2046这个数目加三次：

$$(2046+2046+2046)=2046\times 3$$

这就是说，这个子程序的加操作循环程序只能执行三次。但是，这就需要把“3”这个数的等效的二进制数装入寄存器对D，把“2046”这个数的等效的二进制数装入寄存器对H。如果

把“2046”这个数的二进制数装入寄存器对D，“3”这个数的二进制数装入寄存器对H，那么，将会发生什么现象呢？因为寄存器对D中存放着计数器的值，这个子程序的循环程序应该被执行2046次，才能把“3”这个数目加2045次！仅仅交换用来存储这两个数的寄存器对，这个循环或者将要被执行三次，或者2046次。这是不能令人满意的，因为这样可能使延时时间太长。因此，通常需要使用更复杂的乘法操作子程序。

现在，让我们来分析下面两个数的乘法运算：

$$203 \times 114$$

这个乘法运算，可以被认为是：$(4 \times 203) + (10 \times 203) + (100 \times 203)$。请注意，当203被乘以10，或被乘以100时，这个例子中的“1”，相乘的结果向左移，以便增加结果的有效数字。我们之所以能够这样做，是因为参与乘法运算的数从右向左移时，这些乘数以十的幂增加，所以，其结果也一定是以十的幂增加。

例 5-8 用连续加做乘法

```
/这是一个没带符号的整数乘法子程序，它
 使用连续的加法运算。这样来编写这个子
/程序：
/H和L包含一个16位数，它需要参与
 乘法运算。
/D和E存储了另一个16位数。
/把一个16位数的结果存入B和C，然
 后8080从该子程序返回。
/L、E和C
/用来存储最低有效字节，H、D和B
/用来存储最高有效字节。
```

```
MULTP, LXIB    /把用来存储结果
       000     /的寄存器对置为 000000 。
MULA,  MOVAE   /把D和E作为字计数器使用。
       ORAD    /计数器＝ 000000 吗?
       RZ      /是的。把结果存入B和C
               /然后返回。
       XRAA    /不是。把A和进位清零。
       MOVAL   /取被相加的一个数的最低有效
               字节。
       ADDC    /加部分结果的最低有效字节。
       MOVCA   /然后把结果存入C寄存器里。
       MOVAH   /再取要加的数的最高有效字节
               (MSBY)。
       ADCB    /把B的内容和进位与它相加。
       MOVBA   /把结果的最高有效字节保存在
               /B里。
       DCXD    /将D和E的乘数减1。
       JMP     /然后判断被乘数是否已经自加
       MULA    /了所要求的次数。
       0
```

```
      203
    × 114
  -------
      812
     203
  +203
  -------
    23142
```

虽然，上面这个乘法的简化表示并没有把它们表现出来，但是可以把被乘数理解为在其后添“0”，就成了乘数的中间结果。这就清楚地表明了被乘数的 10^n：

```
  203
× 114
-----
  812
 2030
20300
-----
23142
```

这个思想也适用于二进制数乘法：

```
  0110                    6₁₀
× 0011                  × 3₁₀
------                  -----
  0110                   18₁₀
 0110
0000
+ 0000
-------
0010010=18₁₀
```

你可以看到，不论是用十进制数制，还是用二进制数制来进行乘法运算，相同数值的乘法运算，其结果都是相同的，因为作为结果“0010010”这个数等于(1×16)+(1×2)，即 18 。这是二进制数乘法的第一个例子。因为二进制的数字或者是 1，或者是 0，所以必须按位检查乘数，是 1 或是 0，乘数被检查，自右向左，开始检查，一次检查一位，是否是 1 。如果发现是 1，把被乘数加到在适当位置的累加的结果上。请读者记住，实际上，诸如十进制那种情况的乘法例子的“乘积”是没有的。1 就表明必须把被乘数加到累加的结果上。为什么？请记住，任何一个数乘以 1，其乘积与这个数本身相同，即：$1\times x = x$。

如果需要的话，我们也可以自左向右检查乘数，而不采用自右向左检查乘数。这就是说，首先检查乘数的最高有效位(MSB)。由于向最高有效位的右边各位检查，所以，也必须把乘积向右边移。这一过程如例 5-9 所示。

例 5-9 十进制和二进制乘法，应该首先检查乘数的最高有效位

43_{10} 00101011

53_{10} × 00110101

215 00000000

+ 129 00000000

2279_{10} 00101011

00101011

00000000

00101011

00000000

00101011

000100011100111 = 2279_{10}

检查这个二进制乘运算的结果，

$(1\times2048)+(1\times128)+(1\times64)+(1\times32)+(1\times4)+(1\times2)+$
$+(1\times1)=2279_{10}$。

把乘数向左移或向右移，这两个方法，哪一个比较好呢？实际上，只要部分结果或被累加的结果，即乘积被累加的数，向同一个方向移动时，乘数向左移还是向右移都没有关系。

在基本指令这一章，我们讨论了许多方法，它们可以用来检查某一位是不是逻辑 1，可以用来执行移位操作的循环移位指令，我们也讨论过了执行循环移位指令时，A寄存器的内容的变化情况，也表示出来了。因此，利用测试和移位方法，编制子程序，实现乘法运算，这是比较简单的。

例 5-10 两个 8 位数相乘的数的乘法子程序

```
/D寄存器的内容是乘数，E寄存器存储的内
 容是被乘数。16 位数结果
/将被存入寄存器对B，(B寄存器存储最高
/有效字节，C寄存器存储最低有效字节)。
/参与乘法运算的数必须在调用该子程
/序之前存入D寄存器和E寄存器。
MP 88,   LXLB      /把将要用来存储乘积的
         000       /寄存器对置为 000000。
```

```
        000         /(八进制 0000000 =十六进制 0000)
        MVIL        /把乘数的位数
        010         /装入L寄存器。
NXTBIT, MOVAD       /把乘数移入A寄存器。
        RAR         /把乘数的1位循环移入进位。
        MOVDA       /把乘数存回到D寄存器。
        JNC         /如果进位是逻辑0。
        NOADD       /只需对部分结果循环移位。
        0           /如果进位是逻辑1，则把被乘
                    /数加到该结果上。
        MOVAB       /取部分结果的最高有效字节。
        ADDE        /加这个被乘数。
        MOVBA       /把该结果存回到B寄存器。
NOADD,MOVAB         /现在，把这个16位的结果
        RAR         /向右循环移位一次
        MOVBA
        MOVAC       /现在，把最高有效字节的任何。
                     一位进位
        RAR         /循环移入最低有效字节的最
                    /高有效位。
        MOVCA       /把最低有效字节存回到C
        DCRL        /乘数的八位数据都已被检测完
                    /了吗?
        JNZ         /如果没有,那么
        NXTBIT      /返回，并检测另一位数据。
        0
        RET         /是的，各位数据都被检测完毕，
                    /把结果存入B和C，返回到主程序。
```

例 5-10 所示的这个子程序的第一条指令用来使寄存器对

B清零，该寄存器对用来存放乘法运算的结果的16位数的。然后，把010（十六进制08）装入L寄存器，这个数是这个子程序中的乘操作循环需要执行的次数。这个数也是乘数的位数，我们必须检测该位数的每一位是逻辑1，或是逻辑0。这些寄存器初始化以后，D寄存器的内容（乘数）被送到A寄存器，向右循环移位一次，然后再把该结果又存回到D寄存器。通过RAR指令的操作，A寄存器(乘数）的最低有效位被循环移入进位。如果把逻辑1移入进位，那么，必须把被乘数加到部分结果之上。因此，8080并不执行JNC-NOADD指令。如果把逻辑0移入进位，则执行JNC-NOADD指令，不产生加法操作。

如果逻辑1被循环移入进位，那么被乘数被加到存储在寄存器对B的部分结果上。B寄存器的内容被送到A寄存器，8080执行ADDE指令，将E寄存器的内容(被乘数)送到A寄存器，与它的内容相加，该结果的最高有效字节存放在B寄存器里。被乘数与部分结果的最高有效字节相加以后，存放在寄存器对B的整个16位数的部分结果都依次循环右移一位。然后，将存放在L寄存器的位数减1。如果这一操作的结果不是0，那么，在乘数中还有一些数据位必须进行测试。

如果乘数的一个逻辑0循环移入进位，那么，绝对不能把被乘数与寄存器对B的部分结果相加。因此，8080执行JNC-NOADD指令，在这里，即使没有把被乘数与B寄存器的内容相加，寄存器对B的内容也要循环右移一位。然后，存放在L寄存器的位数被减1；如果减1后的结果不等于零，那么，必须测试乘数中的另一位。

最后，当L寄存器的内容被减为零时，乘数中的所有各位被检测完毕，8080把乘积的16位数结果存入寄存器对B，然

后，从该子程序返回主程序。结果的最高有效字节存储在B寄存器里，结果的最低有效字节存储在C寄存器里。为了真正地理解8080的这些操作，让我们来检查在这个子程序被执行时，这些寄存器的内容。假设我们要做这两个数的乘法：32_{10}乘以5_{10}，首先让B＝00000000，C＝00000000，L＝010，D＝00000101，E＝00100000。

1．取乘数00000101，并使它循环移位（00000101循环移位成为00000010；进位是逻辑1）。

2．因为进位是逻辑1，所以把被乘数与部分结果相加，并且把部分结果存入寄存器对B。

```
  00000000      B寄存器(部分结果)
+ 00100000      D寄存器(被乘数)
  00100000
```

3．现在，把存放在寄存器对B的部分结果循环移位（00100000，00000000循环移位成为00010000，00000000）。

4．位数减1，现在，L寄存器存放3007(07)，这个数不是0。

5．取乘数00000010，并使它循环移位（00000010循环移位成00000001；进位是0）。

6．因为进位是0，所以只对部分结果循环移位；没有进行加（00010000 00000000循环移位成为00001000000）。

7．位数减1；现在，L寄存器存储006(06)，这个数不是0。

8．取乘数00000001，并把它循环移位（00000001循环移位成为00000000；进位是1）。

9．因为进位是逻辑1；所以把被乘数加到部分结果之上，然后把部分结果存入寄存器对B。

```
  00001000      B寄存器(部分结果)
+ 00100000      D寄存器(被乘数)
  00101000
```

10. 现在，对存放在寄存器对B的部分结果循环移位(00101000 00000000循环移位成为00010100 00000000)。

11. 位数减1；现在，L寄存器存储005(05)，这个数不是0。

因为在乘数中，再也没有数字"1"了，所以对于该子程序的其余部分，8080进行的唯一操作是把寄存器对B所存放的部分结果循环移位。循环移位指令对寄存器对B的影响，对L寄存器的影响，以及对位计数器的影响如下所示：

寄存器对B　　L寄存器

00010100,00000000 右移成 00001010 ,00000000； L＝004(04
00001010 00000000 右移成 00000101 00000000； L＝003(03)
00000101 00000000 右移成 00000010 10000000； L＝002(02)
00000010 10000000 右移成 00000001 01000000； L＝001(01)
00000001 01000000 右移成 00000000 10100000； L＝000(00)

L寄存器的内容已经被减为0，所以，8080微处理器应该从MP88这个子程序(例5-10)返回。它把结果的最高有效字节存放在B寄存器里；结果的最低有效字节存放在C寄存器里。尤如读者所希望的那样：$5_{10}\times32_{10}=160_{10}$，即 $5_8\times40_8=240_8$；$240_8=160_{10}=10100000_2$。所以，这个乘法运算所得到的结果是正确的。现在，关于被乘数为什么要被加到部分结果的最高有效字节的问题，想必你应该会认识到了。最后，把它循环右移，进入最低有效字节(LSBY)。

DADH这条指令把寄存器对H的内容向循环左移一位，这一操作已经被明确展示出来了。因此，这条指令可以用于8位

二进制数乘以 8 位二进制数的乘法子程序，如例 5-11 所示。

例 5-11 把 DADH 指令应用于两个 8 位二进制数相乘的乘法子程序

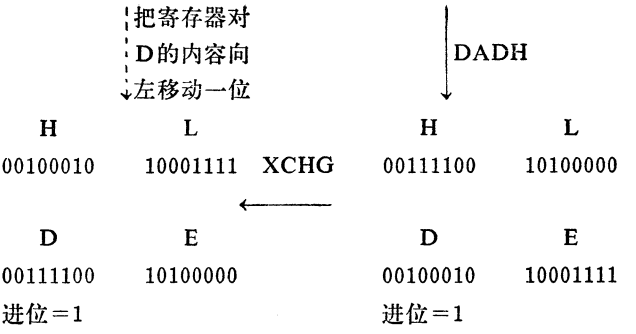
```
/C寄存器的内容是被乘数；
/D 寄存器的内容是乘数。
/这个 16 位的结果将被存放在寄存器对H
/(H存放的是最高有效字节； L
/存放的是最低有效字节)。
/8080 在调用该子程序之前，必须对C和D进行传送操作。
MPBBAA,  MVIA   /把乘数的位数
         101    /装入 A 寄存器里。
         MVIB   /把寄存器对 B 的最高有效
         000    /字节置为 000(00)。
         LXIH   /把寄存器对 H
         000    /的内容清除为 000000(0000)，
         000    /用来存放其结果。
NXTBIT,  XCHG   /把 D 寄存器的乘数送到 H 寄存器里。
         DADH   /把寄存器对 H 的内容向左移一位。
         XCHG   /取回乘数，送入 D 寄存器。
         JNC    /如果零被移入进位，
         NOADD/那么，只对其结果循环移位，否则，
         0
         DADB   /把被乘数加到其结果之上。
NOADD,   DCRA   / 8 位数据都已经被检测完了吗?
         RZ     /是的。然后，8080 从该子程序返回。
         DADH   /不。把 H 和 L 的结果循环移动
                / 1 位。
         JMP    /然后，检测乘数的下一位。
         NXTBIT
         0
```

在例 5-11 中，乘数被存放在D寄存器里；被乘数被存放在C寄存器里。当乘法操作完毕时，其结果被存在寄存器对H里。8080 调用该子程序时，把位数计数器的值（八进制 010，十六进制 08）装入A寄存器。把 000 装入B寄存器，它是寄存器对B的最高有效字节(MSBY)；把被乘数装入C寄存器，它是寄存器对B的最低有效位(LSBY)。这就是说，8080 可以执行一条DADB 指令，从而把被乘数加到寄存器对H的部分结果上。8080 执行 LXIH 指令，把寄存器对H清零，因为这个寄存器对将要用来存储部分结果和最终结果。

在 NXTBIT 这个标号上，8080 把寄存器对D的内容和寄存器对H的内容进行交换。原来乘数存放在D寄存器里，现在存放在H寄存器里。不管从E寄存器将什么传送到L寄存器都无关紧要。部分结果原来存放在寄存器对H里，现在存储在寄存器对D里。8080 通过执行 DADH 指令，把乘数向左移动一位，这是通过把寄存器对H的内容自加来实现的。当 8080 执行第二条 XCHG 指令时，把已被移位的乘数的各位存回到寄存器对D。8080 通过加法操作（执行 DADH 指令），完成这个移位操作之后，进位或者是 1，或者是 0，反映了在 XCHG，DADH，XCHG 程序序列之前，寄存器对D中最高有效位的状态。这种移动方式如例 5-12 所示。

例 5-12 用 XCHG 和 DADH 指令，移动寄存器对D的内容

H	L		H	L
00100010	10001111	XCHG	10011110	01010000
		⟶		
D	E		D	E
10011110	01010000		00100010	10001111



执行 DADH 指令之后，如果寄存器对D的最高有效位是0，则进位等于0。在这种情况下，不能把被乘数与寄存器对H所存储的部分结果相加。因此，8080 执行 JNC-NOADD 指令，从而把存放在A寄存器的数据的位数减1。如果计数的位数减到零时，则 8080 把该乘法运算的结果存入寄存器对H，然后从该子程序返回。如果位计数值不等于零，则 8080 执行 DADH 指令，把寄存器对H的内容(部分结果)向左移动一位。然后，8080 检测乘数的下一位。只有乘数中的 8 位都被检测以后，该子程序才执行完毕。

但是，如果由于把寄存器对D的最高有效位移入进位的结果，进位是逻辑0，如前所述，则寄存器对B的内容(被乘数)要加到寄存器对H的内容之上，即与部分结果相加。加操作之后，如前面所描述的那样，8080 继续执行这个子程序。根据这一描述，现在，读者应该懂得：当C寄存器存储被乘数时，为什么开始必须把B寄存器的内容置于000(00)。8080 把寄存器对B的内容与寄存器对H的内容相加时，寄存器对B的最高有效字节(MSBY)一定是零。

正如我们在前面所提到的那样，乘数向左移或向右移，这是无关紧要的。但是，我们必须把部分结果向同一个方向移

动。在例 5-10 中，我们把乘数和部分结果向右移。在例 5-11 中，我们使用了一条 DADH 指令，来把乘数向左移，然后，使用了一条 DADH 指令，把部分结果向左移。

作为整数乘法的最后一个例子，我们将分析 16 位乘以 16 位产生 32 位结果的乘法子程序。在前面的各个乘法例子中，乘数，被乘数和结果，都分别存放在 8080 微处理机的通用寄存器里。在例 5-13 这个新的乘法子程序中，要做到这一点，是不可能的，因为存放 32 位结果需要四个寄存器，存放 16 位乘数需要两个寄存器；存放 16 位被乘数也需要两个寄存器；另外还需要一个寄存器来存放乘数的位数。因此，我们可以采用几种办法，以便堆栈指示器寻址的读/写存储器单元可以用作暂时存储器。

在例 5-13 的子程序中，我们首先把乘数的位数装入A寄存器里，乘数的位数是 16 位（十进制），这个位数也是这个子程序中的循环，必须执行的次数。因为我们用寄存器对H存放 16 位数的结果，所以首先把 000000(0000)装入该寄存器对，然后，把这个数压入堆栈。现在，有四个 8 位存储单元，被预置为零。当然，由于 XCHG，DADH，XCHG，这种指令序列执行的结果，寄存器对D的内容已经向左移了一位，现在，进位是寄存器对D移位之前所存放的最高有效位(MSB)的值。

在该程序的这一点上，8080 根据进位的状态作出判断。如果进位是逻辑 1，那么，8080 必须把被乘数加到 32 位的部分结果之上，所以，必须把存放在寄存器对 B 的被乘数加到寄存器对H的内容上。如果该 16 位加的结果出现进位，那么，必须把 1 加到该结果的 16 位最高有效位上，而且通常存放在堆栈中。这就是为什么把 000000(0000)装入寄存器对H之后，才把寄存器对H的结果压入堆栈的原因。如果由于该加操作确实

出现了一位进位，那么，8080 执行 XTHL 指令时，寄存器对 H的内容和堆栈上的最后两个字节被进行交换。然后，8080执行“INXH”这条指令，把结果 16 位高有效位加 1，然后使用第二条 XTHL 指令把它压入堆栈。请读者注意，把 1 加到一个数上，正如使这个数增加 1 一样。还要请读者注意，在该程序的这一点上，不能用压入指令将结果的 16 位最高有效位压入堆栈，也不能用弹出指令将它从堆栈弹出而送入某个寄存器对，这是因为没有合适的寄存器对来保存这 16 位数的缘故。因此，我们使用 XTHL 指令。请记住，从堆栈传递的数都是 16 位的数。

例 5-13 16 位数乘以 16 位数的乘法子程序(32 位结果)

```
/这个子程序用来把寄存器对D的内容
/乘以寄存器对B的内容，把 32 位数的
/结果存储在寄存器对H里(两个最低有效字节)
/和寄存器对D里(两个最高有效字节)。
MP 1616,  MVIA     /把乘数的位数
          020      /装入A寄存器
          LXIH     /把寄存器对H和堆栈的最后一
          000      /个入口置为 000000(0000)。
          000
          PUSHH
NXTBIT,   XCHG     /取乘数，装入H和L寄存器里。
          DADH     /把最高有效位循环移入进位。
          XCHG     /把乘数送回到D和E寄存器。
          JNC      /如果进位等于零，不能把被乘数
          NOADD    /与H和L以及堆栈的
          0        /部分结果相加。
          DADB     /如果进位等于 1，把B和C的
```

```
                    /内容加到H和L，
                    /结果留在H和L。
          JNC       /应该把 1 加到堆栈
          NOADD     /存储的结果的最高
          0         /有效字节上吗?
          XTHL      /是的，把H和L的内容与堆栈
                    /入口的交换。
          INXH      /16 位最高有效字节加 1。
          XTHL      /然后，把它存回堆栈里。
NOADD,    DCRA      /使位数减 1。
          JNZ       /位数不等于零，所以
          NOTEND    /检测乘数的另一位。
          0
          POPD      /从堆栈弹出 16 位结果的最高有效
                    /字节。
          RET       /然后，32 位结果存入D与E
                    /和H与L，返回。
NOTEND,DADH         /把结果的最低有效字节循环左
                     移。
          XTHL      /取最高有效字节装入H和L。
          PUSHPSW/把位数和进位保存在堆栈里。
          DADH      /把最高有效字节间循环左移一
                    /次。
          POPPSW    /从堆栈弹出位数和进位。
          JNC       /最低有效字节有进位产生吗?
          NOCMSB    /没有，则不把 1 加到最高有效
          0         /字节上。
          INXH      /高位有效字节加 1。
NOCMSB,XTHL         /把高位有效字节存回堆栈.
          JMP       /然后检测另一位乘数。
```

NXTBIT

0

如果进位是逻辑 0，作为使寄存器对D所存贮的乘数旋转的结果，则不把被乘数加到局部结果上。不论该进位的状态是什么，是否把被乘数加到局部结果上，程序执行控制总是要到达在 NOADD 标号上的位数减 1 指令(DCRA)，在 NOADD 处，8080 处理位数。如果位数是零，从寄存器弹出高有效位 16 位，并装入寄存器对D里，然后，8080 执行 RET 返回指令。当这种情况出现时，作为结果的 16 位高有效位存放在寄存器对D里；作为结果的 16 位低有效位存贮在寄存器对H里。

如果位数不等于零，必须把 32 位部分结果向左移 1 位。8080 到达 NOTEND 标号时，它执行 32 位移位操作。这个 32 位结果的 16 位低有效位存放在寄存器对H；16 位高有效位存放在堆栈里。因为必须把部分结果向左移，所以首先必须把 16 位低有效位移位。这样，8080 给它们移位时，我们可以把来自最低有效位的任何进位移入到最高有效位 (MSB)。在 WOTEND 处的 DADH 指令，把 16 位低有效位向左移 1 位。在该程序的这一点时，由于执行 DADH 指令的结果，进位或者是逻辑 1，或者是逻辑 0。8080 执行这条 DADH 指令之后，执行 XTHL 指令，把作为结果的 16 位低有效位和 16 位高有效位进行交换。现在，问题是要把进位移入部分结果的最高有效位(MSB)。怎样才能完成这一操作呢？

这个问题是困难的，因为 DAD 类指令并不能把前面的进位加到现在正在做加法运算的数上。但是，DAD 类指令确实影响进位，因此，8080 在执行一条 DADH 指令，以使 16 位高有效位移位之前，把执行第一条 DADH 指令所产生的进位可以保存在堆栈里。因此，8080 执行 PUSHPSW，这条指令，

把进位保存在堆栈里。当 8080 执行 DADH 指令时，把 16 位高有效位向左移一位位置。现在必须把执行前一条 DADH 指令所产生的进位加到 16 位高有效位上。8080 从堆栈弹出A寄存器的内容和标识字节，然后，它根据进位的状态作出判定。如果进位是逻辑零，则没有进位被加到最高有效位上。如果进位是逻辑 1，则必须使寄存器对H的高有效 16 位加 1。8080的程序执行控制到达 NOCMSB(没有进位进入最高有效位)时，它把低有效的 16 位放回到堆栈里，8080 执行 XTHL 指令时，寄存器对H所存储的内容是已移位的部分结果的 16 位低有效位。8080 执行 XTHL 这条指令后，它转移到 NXTBIT，在这个标号上，8080 测试乘数的下一位。

虽然，乘法子程序的长度和复杂程度都是变化的，但是，它们在操作上都是非常类似的。在其他子程序中，我们都采用检测-移位法，只有在例 5-8 中，我们采用了连续加的方法。检测-移位法，我们称之为加-移位算位，这是比较适合的。用乘法程序进行计算时，你应该记住的一点是：结果的最大位数等于乘数的位数与被乘数的位数之和。这就是说：如果两个 16 位数相乘，可以得到 32 位数的结果。如果一个 16 位的数与一个 8 位的数相乘，其结果的数的位数可以达到 24 位。记住这一点是很重要的，因为这要你分配足够的存储空间来存储结果。**这就是说，如果例 5-8 的子程序用来将两个 16 位数相乘，那么，8080 只把结果的 16 位低有效位存入寄存器对 B 里，然后从该子程序返回。16 位高有效位(MSB)将被丢失。**

整 数 除

在本章有关乘法的这一节里，例 5-8 所示的程序是，采用将被乘数自加乘数所规定的次数的方法来完成乘法运算的。例如：

$$15=5\times3=5+5+5=3+3+3+3+3$$

除法运算，我们**可以采用从被除数减去若干次除数的方法**。每执行一次操作，而且不产生借位时，计数数加 1。结果计数次数等于被除数的**整数商**。当这种减法操作完成后，比除数小；而比 0 大的任何数就是被除数的**余数**。因此，在被除数 15 除以 3 的除法运算中：

$15-3=12$；	计数值＝1
$12-3=9$；	计数值＝2
$9-3=6$；	计数值＝3
$6-3=3$；	计数值＝4
$3-3=0$；	计数值＝5

在这个例子中，被除数的商等于 5，余数等于 0。在被除数 7 除以 4 的除法式中：

$7-4=3$；	计数值＝1
$3-4=-1$	产生了借位
$4+(-1)=3$	余数

这个除法的商是 1，余数是 3。当然，对于大数而言，这个过程很麻烦，所以，我们常常采用简便的方法来做除法运算。

做除法运算用十进制还是用二进制，都无关紧要，其结果都是相同的，因为 $00101010=(1\times32)+(1\times8)+(1\times2)=$

4210。但是，可惜需要用除法运算的大多数题目并不简单。请看用除法来解下式：

$$2151\overline{)12835}$$

首先可以试商 5 或者 6，

$$5\times 2151=10755$$

$$6\times 2151=12906$$

只要把商和除数相乘，从被除数减去其结果，便可以确定适当的商数。

$$\begin{array}{r} 12835 \\ -10755 \\ \hline 2080 \end{array} \qquad \begin{array}{r} 12835 \\ -12906 \\ \hline -71 \end{array}$$

从这个例子，我们很容易地看出第一次的商一定是 5。

例 5-14 十进制和二进制除法

$$\begin{array}{r} 00101010 \\ 101\overline{)11010010} \\ -0 \\ \hline 11 \\ -00 \\ \hline 110 \\ -101 \\ \hline 11 \\ -00 \\ \hline 110 \\ -101 \\ \hline 10 \\ -00 \\ \hline 101 \\ -101 \\ \hline 00 \\ -00 \\ \hline 0 \end{array} \qquad \begin{array}{r} 042 \\ 5\overline{)210} \\ -0 \\ \hline 21 \\ -20 \\ \hline 10 \\ -10 \\ \hline 0 \end{array}$$

从例 5-14 所示的二进制除法例子，我们可以看到，首先从被除数的第一位减去除数。如果减法运算的结果出现了借位

(如上面的例子所示)，那么，在商数上写入一个 0；然后，把除数加到减的结果上，这样，就恢复了被除数的原来的值。如果在减运算过程中没有出现借位，那么，则在商上写入 1，并且把这次减运算的结果供下一次减-检测过程使用。这一过程如例 5-15 所示。

例 5-15 利用减-检测的方法把 11010010 除以101

```
101/ 11010010
     0011010010          被除数
    -1010000000          除数
     1001010010          出现借位，商数位=0
    +1010000000          所以把除数与结果相加
     0011010010
    -  101000000         移下除数，再试，
     1110010010          出现了借位，商数位=0
    +  101000000         所以把除数与结果相加
     0011010010
    -   10100000         移下除数，再试，
     0000110010          无借位，商数位=1
    -    1010000         不加，移下除数，再试，
     1111100010          出现借位，商数位=0
    +    1010000         所以把除数与结果相加。
     0000110010
    -     101000         移下除数，再试，
     0000001010          无借位，商数位=1
    -      10100         不加，移除数，再试，
     1111110110          出现借位，商数位=0
    +      10100         所以把除数与结果相加。
     0000001010
```

```
  —      1010        移除数，再试。
  0000000000         无借位，商数位＝1。
  —       101        不加，移除数，再试，
  1111111011         出现借位，商数位＝0
  +       101        所以把除数与结果相加。
  0000000000
```

移除数和减运算已经进行了八次，因此，被除数已经被除尽，得到了商数，其结果等于 00101010_2，与前例 5-14 的结果一样。

我们可以编写的最简单的整数除法子程序之一是一个 8 位数除以另一个 8 位数。这个除法运算的结果和余数也是 8 位。在例 5-16 中，存放在 E 寄存器的被除数被存储在 D 寄存器的除数来除。8080 把商存储在 H 寄存器里，把余数存在 C 寄存器里，然后，从该子程序返回。

例 5-16　一个 8 位数除以另一个 8 位数的除法子程序

```
/这是一个除法子程序，用来把一个
/ 8 位数除以另一个 8 位数。
/必须把除数存入D寄存器，
/被除数存入E寄存器，才调用这个子程序。
/8080 从该子程序返回时，其
/结果将被存贮在H寄存器里。
DIV 88,   LXIH    /把 010 (十进制数 8)
          010     /装入L寄存器，把 000
          000     /装入H寄存器 (结果将存放在H)。
          MVIC    /把 000 装入C寄存器，因为部分被
          000     /除数将被存放在C寄存器。
NXTBIT,   MOVAE   /把被除数送到A寄存器里。
          RAL     /把A的最高有效位循环移入进位。
          MOVEA   /把被除数存回到E寄存器。
          MOVAC   /取存放在C寄存器的部分被除数，
```

```
        RAL     /进位移位，进入A的最低有
                /效位。
        SUBD    /从这个数减去除数。
        JNC     /如果进位为0，则没有
        NOADD   /产生借位。因此，把商循环移位。
        0       /否则，把除数加到
        ADDD    /减法运算的结果上。
NOADD,  MOVCA   /把部分被除数存回到C
        CMC     /进位取反，然后
        MOVAH   /把进位循环移入到H寄存
        RAL     /器的最低有效位。
        MOVHA
        DCRL    /8位都检测完毕了吗?
        JNZ     /没有，循环移入被除数的另一位。
        NXTBIT  /然后，再次试减。
        0
        RET     /把答案存放在H寄存器，然后返回。
```

这个子程序所执行的操作与例 5-15 所执行的操作是相同的。该子程序的第一条指令是 LXIH，它用来把 000010(0008) 装入寄存器对 H。实际上，它是把位数（被除数的位数）装入 L 寄存器。同时，8080 把 H 寄存器清零，因为用 H 寄存器存储商数，即除法运算的结果，C 寄存器也要作为暂时存储寄存器使用，所以，8080 也要把它清零。

在例 5-15 里，第一次减操作，实际上是从被除数的第 1 位减去除数。因为从其它最低有效位(LSB)减去 0，所以其它有效位没有受到影响，因而没有产生净效应。因此，除了从被除数的最高有效位减去除数以外，其它各位可以完全忽略不计，例如：

00000001　A寄存器的内容；部分被除数

−00000101　D寄存器的内容；除数

11111100　产生了借位

在这个程序中，正是它要完成的任务。我们把C寄存器(作暂时存储器用)的内容放入A寄存器，而且被除数的最高有效位(MSB)原先已被循环移入进位，现循环移入A寄存器的最低有效位(LSB)。这是"部分的"被除数。现在，从部分被除数减去D寄存器的除数。如果该减操作产生了一位借位，那么进位位将是逻辑1。

如果没有借位（进位＝0)，那么除数比部分被除数小，因此，为了表示这个结果，那么，商数必定是1，写在商的适当位置上。这是通过下列操作来实现的：把减操作的结果存入C寄存器(新的部分被除数)，然后把逻辑1循环移入部分结果，并把该结果存入H寄存器。

如果出现了一位借位，那么，除数比部分被除数大。在这种情况下，把除数加到减操作的结果上。这一操作完成后，把被除数的下一位最低有效位循环移入到部分被除数；这个过程还要重复七次，因为被除数有8位二进制数。

为什么一定要把H寄存器清零，这有什么道理吗？请读者记住：H寄存器用来保存除法运算的商数。乘法子程序必须把部分结果存储在一个寄存器或几个寄存器里，除法子程序与乘法子程序不一样，它利用循环移位指令，把来自进位的商循环移位，一次移动一位，进入H寄存器。为了完成这一操作，8080执行MOVAH，把H寄存器的内容移到A寄存器；执行RAL指令，循环移入A寄存器，然后执行MOVHA指令，把结果存回到H寄存器。因为这个指令序列被执行八次，所以，当8080微处理器调入这个子程序时，不论H寄存器的内容是

什么，都无关紧要，因为在这个程序的始点时，H寄存器所存储的内容不管是什么，在8080从该子程序返回之前，将会循环移位一遍。

我们的目标之一是希望使用尽可能少的存储单元、指令和寄存器，这一思想已经贯穿了全书。在下一个除法子程序，即例5-16所示的除法子程序的改进程序中，我们使用了A、B、C这三个寄存器，但是，没有使用寄存器对H。相反，把除法运算的商存回到E寄存器里。

这个子程序的改进型（例5-17）比例5-16少用四个存储单元和一个寄存器。我们应该指出的第一个不同点是：数据的位数被存放在B寄存器，这个数是十进制数9，而不是8。读者应该能观察得到，数据的位数是在该子程序的中间，而不是在末尾被减1。在该子程序的末尾，只给进位位作反码运算，然后，8080微处理机返回到NXTBIT。NXTBIT接近该子程序的始点。

例 5-17 一个8位数除以另一个8位数的改进型除法子程序

```
/必须把除数存入D寄存器，
/把被除数存入E寄存器，才能调用这个子程序。
/当8080从这个子程序返回时，
/结果将被存储在该寄存器里。
DIV 88 A,   LXIB    /把000装入C寄存器，以便
            000     /存储部分被除数，然后
            011     /把十进制数9装入B。
NXTBIT      MOVAE   /把被除数传送到A寄存器。
            RAL     /把A的最高有效位，
                    /循环移入进位。
            MOVEA   /把被除数存回到E寄存器里。
```

```
        DCRB    /8 位数据都已经被检测完了
                /吗?
        RZ      /是的，检测完了，结果保存在E。
        MOVAC   /不，没检测完，取部分被除
                /数，装入A。
        RAL     /部分被除数循环左移。
        SUBD    /从这个数减去除数。
        JNC     /如果没有出现借位，
        NOADD   /那么，不把除数加到减法
        0       /操作的结果上。
        ADDD    /把除数加到结果上。
NOADD,  MOVCA   /把部分被除数保存在C。
        CMC     /对进位进行取反操作。
        JMP     /然后，8080 返回，并且把进
        NXTBIT  /位循环移入E的最低有效位，
                /
        0       /把E的最高有效位
                /循环移入进位，
                /并测试另一位。
```

8080 微处理机不使用 MOVAH，RAL 和 MOVHA 这三条指令，只执行 CMC 这条指令，仍旧能使进位置零，或清除到适当的状态。然后，8080 执行 JMP-NXTBIT 指令序列。在 NXTBIT 处，8080 把 E 寄存器的内容送到 A 寄存器里；在 A 寄存器，把这个数循环右移一位。然后，把 A 寄存器的新内容存入 E 寄存器里。怎样把商保存在 E 寄存器里呢？当把 E 寄存器的内容传送到 A 寄存器，并移位时，A 寄存器的最高有效位被循环移入进位。同时进位，实际上代表商数的 1 位，被循环移入到 A 寄存器的最低有效位。因此，8080 执行八条循环移位

指令后，商数将被存放在E寄存器里。如果是这样，那么，为什么要把位数定到十进制9呢？让我们假设，处理位数减1用的DCR类指令，仍然编写在该子程序的末尾，并且数据位数开始定到十进制数8。我们仍然需要把循环移入E寄存器。这些指令被列于例5-18。

例 5-18 给DIV 88A子程序（例5-17）结尾的不适当的方法

```
        •
        •
        MOVCA   /把部分被除数存入C寄存器里。
NOADD,  CMC     /对进位进行取反操作。
        DCRB    /8位被除数都已经被测试完了
        JNZ     /吗？没有。再测试另一位。
        NXTBIT
        0
        RET     /是的。数据位数等于0，把答
                /案存入E。
```

看起来，这个指令序列会正常工作，因为DCRB指令并不影响进位。但是，8080最后一次执行循环时，将进位取反，以反映商数的最后一位最低有效位。然而，8080将把B寄存器的内容从001减成为000(0100)，并不把商的最后这一位循环移入商数。因为，该子程序的开始，把被除数的1位循环移入了进位。从进位把一位商数循环移入了商，所以，必须把DCRB指令移走，数据的位数必须从十进制8增至9。这就是说，这个子程序的循环移位指令部分（RZ指令前面的指令）要执行九次；这个子程序的减操作指令，只执行八次。除法运算的余数保存在寄存器里吗？是的，余被保存在C寄存器里。

使用同一类指令，编写一个 16 位数除以另一个 16 位数的子程序是容易的。对于这个子程序来说，必须同时存放下列数：

16 位除数，
16 位被除数，
16 位部分被除数，
16 位商数，
16 位余数，
8 位计数器的值。

由于这些存储要求，这些数的某一些，我们将用读/写存储器来存储。

例 5-13 一个 16 位数除以 16 位数的除法子程序

```
/这个子程序用于一个 16 位数除以另一个 16 位
/数，寄存器对 D 存放被除数，寄存器对 B 存
/除数。
/当 8080 从该子程序返回时，结果将被存储在
/寄存器对 D 里，余数存放
/在寄存器对 B 里。
DV 1616,    LXIH    /把符号地址 TEMP 的存储单元
            TEMP    /的地址送入寄存器对 H。
            0       /这是一个读/写存储器
                    /地址。
            MOVMC   /把除数的最低有效字节保存
                    /在存储器中。
            INXH    /把这个存储地址加 1 。
            MOVMB   /把除数的最高有效字节存入
                    /存储单元。
            INXH    /把这个存储地址加 1 。
```

```
        MVIM    /然后，把除数的位数
        027     /(十进制 7 )存入存储单元。
        LXIB    /把 000000(0000)
        000     /存入寄存器对 B 。寄存器对
        000     / B 将用来存部分被除数。
NXTBIT, LXIH    /把除数的位数(开始是 021)
        COUNT   /存入的存储地址
        0       /装入寄存器对 H。
        MOVAE   /取除数的 LSBY(最低有效字节)装入 A。
        RAL     /把 MSB(最高有效位)循环移入进位。
        MOVEA   /把被除数的 LSBY(最低有效字节) 存回 E。
        MOVAD   /然后，取被除数的 MSBY (最高有效字节)。
        RAL     /将 MSB(最高有效位)，循环左移，进入进位。
        MOVDA   /把被除数的 MSBY (最高有效字节)存入 D。
        DCRM    /把存储在存储器的位数减 1 。
        RZ      /如这个数等于 0，则返回。
        MOVAC   /否则，将存储在 B 、C 里的 被 除 数 的 MSB
                (最高有效位)
        RAL     /位循环左移，进入部分被除数。
        MOVCA   /
        MOVAB
        RAL
        MOVBA
        DCXH    /把这个存储地址减 1 ，以便
        DCXH    /使 H 和 L 指到存储器的除
                /数。
        MOVAC   /取部分被除数的 LSBY。
        SUBM    /减去除数的 LSBY。
        MOVCA   /把结果存回 C 寄存器里。
        INXH    /这个地址加 1 。
```

```
        MOVAB   /取部分被除数的 MSBY。
        SBBM    /带借位减存储器的除数。
        MOVBA   /其结果存回 B 寄存器。
        JNC     /如果进位等于 0，不要把
        NOADD   /除数加到前一个减操作
        0       /的结果上。
        DCXH    /除数比部分被除数大，
        MOVAC   /所以，必须把除数加到减操
        ADDM    /作的结果上，
        MOVCA   /恢复部分被除数
        INXH    /的原来的值
        MOVAB   /
        ADCM
        MOVBA
NOADD,  CMC     /进位位取反。
        JMP     /然后，测试除数
        NXTBIT  /的另一位
        0
TEMP,   0       /这些存储单元用来
        0       /存储除数及其
COUNT,          /位数
```

读者可以看到，这个子程序是相当长的。如果我们使用 8080 微处理机的一些功能更强的指令，那么，可以把这个程序编写得较短一些；但是，我们有目的地使该子程序简单，以便它尽可能容易理解。当 8080 调入这个子程序时，被除数必须被存放在寄存器对 D 里，必须把除数存放在寄存器对 B 里。

因为有些数必须被存储在读/写存储器里，这个程序把读/写存储器的一个单元地址装入寄存器对 H。然后把 C 与 B 这两

个寄存器的内容和除数的位数存入读/写存储器里。位数是021(十进制 17)，因为被除数有 16 位，所以，8080 必须把这个子程序的循环移位的循环多执行一次，即执行 17 次，商数的最后一位才能被保存在适当的寄存器对里。然后，8080 把寄存器对 B 清零，因为这个寄存器对是用来保存部分被除数的寄存器对。在 NXTBIT 处，把存储位数的读/写存储器地址（被分配给符号地址 COUNT）装入寄存器对 H。然后，寄存器对 D 里装着的 16 位被除数循环左移一次。由此产生的进位，或者是逻辑 1，或者是逻辑 0。正如前一个例子所说明的那样，那个由六条指令所构成的指令序列不仅把被除数的最高有效位(MSB)循环移入进位，而且它还把进位循环移入寄存器对 D 的最低有效位(LSB)。我们采用这个办法，把商数，逐位循环移入寄存器对 D。

被除数循环移位完毕以后，把存储在读/写存储器的位数减 1。当位数最后被减到 0 时，这个子程序执行完毕。如果 8080 执行 DCRM 指令以后，位数不等于 0，那么，8080 不执行 RZ 指令。被除数的最高有效位（MSB）现在处在进位位置，将它被移入部分被除数的最低有效位(LSB)；部分被除数存储在寄存器对 B 里。请读者注意，这条 DCRM 指令并不影响进位的内容。

正如前一个例子的情况一样，一旦部分被除数形成后，必须从该被除数减去除数。除数被存储在读/写存储器，所以寄存器对 H 所存储的存储单元地址必须减 1 两次，寄存器对 H 才能指示到除数的最低有效字节。8080 执行两条 DCXH 指令，使这个地址减 1。部分被除数的最低有效字节（C 寄存器的内容）被送到 A 寄存器里，然后，8080 执行 SUBM 指令，从 A 寄存器的内容减去寄存器对 H 所寻址的存储单元的内容，

减操作的结果被保存在C寄存器里，寄存器对H所存放的存储单元地址然后加1，使这个地址指示到存放除数的最高有效字节的存储单元。

部分被除数的最高有效字节存放在B寄存器里，现在被传送到A寄存器里，然后8080执行SBBM指令，从A寄存器的内容带借位减去寄存器对H所寻址的存储单元的内容。我们为什么使用SBBM指令而不使用SUBM指令呢？我们使用这条SBBM指令，是因为从部分被除数的最低有效字节（LSBY)减去除数的最低有效字节（LSBY）时，也许已经有借位产生了。这条SBBM指令，不仅从A寄存器的内容减去存储单元的内容，而且从A寄存器的内容减去进位的内容。当然，进位包含任何借位。如果整个16位减操作没有出现借位，那么，部分被除数等于或大于除数。如果由于16位数减操作的结果，产生了一位借位，那么该部分被除数小于除数，然后必须把除数加到16位数减操作的结果上，才能使部分被除数恢复到原来的值。为了达到这个目的，8080执行ADDM和ADCM这两条指令。如果16位数的减操作没有产生借位，那么，8080转移到NOADD。

在NOADD时，进位取反，以便反应出16位减操作的结果是否出现了借位。然后把这位进位位循环移入寄存器对D的最低有效位；与此同时，寄存器对D所存放的被除数的下一最高有效位间循环左移，进入进位。

商，或除法运算的结果被存放在什么地方呢？商存放在寄存器对D里。在许多情况下，被除数不是除数的倍数，因此，被除数的16位数都检测完毕以后，可能有余数存在。这个余数被保留吗？是的。16位数的余数保存在寄存器对B里。

你认为这个16位数除以16位数的除数子程序（如例5-29

所示）可以用来进行 16 位数除以 8 位数的除法运算吗？是的，可以；但是你应该保证：8080 调入这个子程序之前，必须把 B 寄存器（存放除数的最高有效字节）装入000，如果用寄存器对 B 用来存放除数的最高有效字节（MSBY），寄存器对 D 用来存放被除数的最高有效字节（MSBY），把这两个寄存器对分别置成 000（00），那么，这个子程序也可以被用来做 8 位数除以 8 位数的除数运算。

这个子程序（例 5-19）还可以被用来做小数除法运算！做小数除法运算又要假设一个 16 位的数代表一个 8 位数和一个 8 位小数，或者只代表一个 16 位的小数就行了。换句话说，假设在两个 8 位的数之间或者在这个 16 位数的前面有一个小数点（隐含小数点）。实际上，这个小数点叫做二进制小数点，因为我们将用二进制数和二进制小数运算。例如，

原来的	现在的
XXXXXXXX YYYYYYYY	XXXXXXXX·YYYYYYYY
代表 0 和 65，535 之间的各个数。	代表 0 和 255.996 之间的各个数。
	·XXXXXXXX YYYYYYYY
	代表 0 和 0.9998 之间的各个数。

对于整数来说，某一位离二进制小数点的左边越远，这个数就越大。因此，二进制数 1000 比二进制数 0110 大。对于小数来说，离开小数点的右边越远，这个数就越小。这就是说，二进制数 .0001 比二进制数 .1010 小。对于一个 8 位的小数来说，每一位的有效值如下所示：

0.5	0.25	0.125	0.0625	0.03125	0.015625	0.0078125	0.00390625
•	X	X	X	X	X	X	X

为了用一个 8 位的整数去除一个 16 位 的数（8 位 整数和 8 位小数），我们应该把这个 8 位的除数送 入 C 寄存器里，把除数的最高有效字节 000 (00) 送入 B 寄存器。把这个 16 位的被除数装入寄存器对 D（被除数的整数部分 送 入 D 寄 存器，被除数的小数部分送入 E 寄存器）。 应 该把整数和小数的商送回到寄存器对 B （整数送入 B 寄存器里；小数送 入 C 寄 存 器里）。我们用下面两个例子来说明这些概念。

为了进行 20.5 除以 4 的运算， 我们应该把下面 的二进制数装入寄存器对 B 和 D：

B	C	D	E
00000000	00000100	0001010	• 1

这个除法运算的结果等于 5.125； 把该结果送回到寄存器对 D 里：

D	E
00000101	• 00100000

这个结果等于 5＋1/8，即 5.125。 我们 也可以把 20.5 这个数除以 5 ：

B	C	D	E
00000000	00000101	00010100 • 10000000	

我们获得的结果等于 00000100 • 00011001。 这个 结果 是正确的吗？ 是的。 因为这个结果的等效的十进制数是 4＋0.0625＋＋0.03125＋0.0390625，即 4.09765。 二进制分数与整数不一

样，它只能是近似的真分数。由于小数只有 8 位，其结果不可能比 1/256，即 0.00390625 更精确的了。因此，当使用一个 8 位的小数时，4.09765 这个数是 8080 微型计算机能够达到的最精确的正确结果。如果我们使用一个 16 位的小数，8080 微型计算机的精确度能够达到 1/65535，即 1.526×10^{-5}。当然，在许多情况下，8080 微型计算机能得到确切的小数结果；例如，5 除以 10，或者 9 除以 36。

BCD 算术运算

把 ASCII 字符转换成二进制数，用来对二进制数进行算术运算，然后把二进制数又转换成 ASCII 字符，输出给电传打字机或 CRT 显示器，对许多人来说，编制出完成这一系列任务的程序来，可能非常困难，而且要花费时间来编写程序。处理数据的一个比较容易的办法应该是，把 ASCII 字符转换成 BCD 数（二进制编码的十进制数），用 BCD 算术子程序来处理这种数，然后，把 BCD 数的结果又转换成 ASCII 字符。随后，这些 ASCII 字符可以在电传打字机上打印。表 5-1 包含的数是 BCD 数及其等效的二进制数。

二进制 1010 到 1111 没有 BCD 数。这些数应该是八进制数 12～17，或是十六进制字符 A～F。因为 8080 微处理机经常运算的数的字长是 8 位，所以两个四位 BCD 数经常被压缩成一个 8 位字。

你已经看到，8080 微处理机有加法和减法指令；适当地使用这些指令，8080 也可以做乘法和除法运算。但是，所有这些指令、程序和子程序一直都是用二进制数来运算的。8080

表 5-1　　BCD 数的等效二进制数

BCD	二　进　制　数
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

对 BCD 数怎样进行加和减运算的呢？ 我们在前面已经讨论过的加法和减法指令可以用来做这种运算。这是足以令人奇怪的。

要把 BCD 数 32 和 47 相加，8080 可以执行一些很简单的指令。例 5-20 列出了这些指令。

例 5-20　把两个压缩的 BCD 数相加

```
•                1
•                •
MVIA/把压缩的 BCD 数据字
062    /节 32 装入A寄存器。
ADI    /把这个压缩的 BCD
107    /立即数据字节 47 加到累加器。
•      /其结果保留在
•      /A寄存器里。
```

如下所示，这两个压缩的 BCD 数应该在A寄存器进行相

加；

```
  00 11   00 10        32
+01 00   01 11      +47
-----------------------
  01 11   10 01        79
```

当然，79 是正确的答案。但是，假设你需要把 59 和 2 这两个数相加：

```
  01 01   10 01        59
+00 00   00 10      +02
-----------------------
  01 01   10 11        ?
```

这两个数相加，其结果并不是我们所希望得到的 61。相反，BCD 数的十位数字还是 5，个位数字是一个禁用的 BCD 数，没有等效的十进制数。与之相对应，为了解决这个问题，我们将要讨论 8080 的 DAA 指令，即十进制调整累加器指令。这条 DAA 指令的操作码是 047 (27)。这条指令用来调整 A 寄存器的内容，它被执行之后，把一个有效的 BCD 数留在 A 寄存器里。**我们必须给 8080 编一种程序，以便在对 BCD 数进行运算的各条加法指令后 8080 能够执行 DAA 指令。**这就是说，把 59 和 2 相加，可采用例 5-21 这个程序。

执行例 5-21 这个指令序列所得到的正确答案是十进制数 61，作为压缩的两位 BCD 数被保存在 B 寄存器里。如前面所示，8080 执行 ADI 指令之后，它得到的结果是 01 01 1011。但是，DAA 指令把结果调整成 0110001。8080 怎样才能完成这个任务呢？

例 5-21　BCD 数相加的合适的子程序

```
•
•
MVIA   /把压缩的 BCD
```

```
131     /数据字节 59 装入A寄存器里。
ADI     /把它与压缩的
002     /BCD 数 02 相加。
DAA     /对A寄存器的内容进行十进制调整。
MOVBA/把其结果 BCD 数
·       /保存在B寄存器里。
·
```

如果 BCD 数的最低有效数字 （A 寄存器的 4 位最低有效位）比 BCD 数 9 大，或者， 如果辅助进位 （AC） 等于逻辑 1， 那么应该把 BCD 数 6 和A寄存器的内容相加。 如果 BCD 数最高有效数字（A寄存器中的四位最高有效位）大于 9 ，或者进位是逻辑 1， 那么应该把 BCD 数 60 与A寄存器的内容相加。这最后一次加操作，等于把 6 加到 BCD 数最高有效数上。因此，在 BCD 数 59 和 BCD 数 2 相加时，会得到 01011011 这个结果。因为这个最低有效数字比 BCD数 9 大 （1001），所以，8080 执行这条 DAA 指令时， 将 BCD数 6 加到A寄存器的内容上。

```
  01 01  10 11
+ 00 00  01 10
-------------
  01 10  00 01
  进位=0
辅助进位=1
```

请读者注意，辅助进位现在被置成逻辑 1， 但是这个八位进位被置成逻辑 0。 DAA 指令是可以用来测试辅助进位的状态的唯一的指令。 因为被压缩的 BCD 数一位最高有效数字被存放在A寄存器里，它小于 10 10，所以不把 BCD 数 60 加到A寄存器的内容上。

假设把 BCD 数 99 加到 BCD 数 3 上面：

```
  10 01  10 01
+ 00 00  00 11
-------------
  10 01  11 00
```

所得到的结果是 10011100。因为 1100 比 1001 (BCD 数 9)大，所以 DAA 指令把 BCD 数 6 加到A寄存器的内容之上。但是，作为这个加法运算的结果，

```
  10 01  11 00      结果
+ 00 00  01 10      用 DAA 操作实现“加”
-------------
  10 10  00 10      DAA 操作的结果
进位＝1
辅助进位＝1
```

被压缩的 BCD 数的最高有效数字大于 BCD 9 (10 01)。因此，BCD 60 必须加到A寄存器的内容上。

```
  01 01  00 10      第一条 DAA 操作的结果
+ 01 10  00 00      由第二条 DAA 操作“加”
-------------
  00 00  00 10      DAA 的最后结果
进位＝1
辅助进位＝0
```

现在，A 寄存器的结果是 BCD 数 02，进位是逻辑 1。这个加法运算的结果 (99＋3＝102)，是 3 位 BCD 数字；存放在A寄存器的两位 BCD 数字代表该结果的两位最低有效数 字。进位位是逻辑 1，它表示：如果有一个百位上的BCD数字被存储在某处，那么，它应该增 1，或者保存这个进位。

8080 也可以进行 BCD 数的减运算，但是，它是很麻烦的。减法运算不是把两位数字的 BCD 数相减，然后用 DAA 指令来调整其结果，而是必须把被减数加到减数的 10 的补码上。这

正如 23－13＝23＋(－13)一样。这就是说，为了从BCD数 54 减去 BCD 数 26，必须首先确定 BCD 数 26 的对 10 的补码。这个补码可以通过从 100 减去 26 (100－26＝74) 得到。现在，可以把 26 的对 10 的补码与 54 相加，并且用 DAA 指令来调整其结果。

54－26＝54＋(100－26)

```
  01 01  01 00        54
+01 11  01 00         (100－26)＝74
-----------------
  11 00  10 00        结果
  进位＝0
辅助进位＝0
```

这个加法运算的结果仍旧必须用 DAA 指令进行调整。你可以看到，这一调整操作只需要把 BCD 数 60 加到A寄存器的内容之上。

```
  11 00  10 00        结果
+01 10  00 00         BCD 数 60（由于 DAA 指令）
-----------------
  00 10  10 00
  进位＝1
辅助进位＝0
```

26 的对 10 的补码与 54 相加的最终结果是 28，这正是 54－26 所得到的结果。

如果读者认为这个方法用来减 BCD 数好象是一种难弄的方法，那么，你的印象是正确的。现在，有待解决的问题是怎样使用程序指令来得到 10 的补码。请读者注意，为了存储 BCD 数 100，需要一个能存储三位 BCD 数的存储单元。你知道，8080 只能把一个两位 BCD 数保存在某个寄存器或者某个存储单元。因此，从 99 减去减数（两位 BCD 数），然后在减操作

的结果加 1，便形成了 10 的补码。因此，54 减去 26 可以归纳写成下式：

$$54-26=54+[(99-26)+1]$$

为了把 B 寄存器的 BCD 数加到 C 寄存器的 BCD 数上，8080 可以执行例 5-22 所列出的程序。

例 5-22 把 C 的 BCD 内容加到 B 中

```
.
.
MOVAB/把B寄存器的内容送到A寄存器。
ADDC  /把C寄存器的内容加到A。
DAA   /对结果进行十进制调整。
.
.
```

为了从 B 寄存器的 BCD 内容减去 C 寄存器的 BCD 内容，需要执行更复杂的指令序列（例 5-23）。

读者从例 5-23 可以看到，只要把立即数字节（BCD 数 99）装入 A 寄存器，就能形成 9 的补码。然后，从 A 寄存器的内容减去 C 寄存器的内容。再把 9 的补码加 1，便形成 10 的补码。然后，把 B 寄存器的内容加到 A 寄存器的内容上。8080 再执行 DAA 指令，将 A 寄存器的内容进行调整，并且把减操作的结果保存在 A 寄存器。

例 5-23 从 B 减去 C 的 BCD 内容

```
.
.
MVIA/把被压缩的BCD数
231   /99送入A寄存器里。
SUBC /减去C的内容，以形成9的补码。
INRA /为了形成10的补码，将A的内容加1
```

```
ADDB/把被减数加到A的内容上。
DAA  /对A寄存器的内容进行10进制调整。
•    /BCD"减操作"的结果现在
•    /存放在A寄存器里。
```

四位 BCD 数的操作

把寄存器对B的内容（BCD 数）加到寄存器对D的内容（BCD 数）上，需要比较复杂一点的程序，该程序如例 5-24 所示。

例 5-24 把寄存器对B的 BCD 内容加到寄存器对D的 BCD 内容之上

```
•
•
BP LUSD,MOVAE /把一个BCD数的最低有效字节送到A。
      ADDC    /与另一个BCD数的最低有效字节相加。
      DAA     /对该结果进行十进制调整。
      MOVLA   /结果的BCD的最低有效字节保存在L中。
      MOVAD   /把--个数的BCD的最高有效字节送入A。
      ADCB    /与另一个数的BCD的最高有效字节相加。
      DAA     /对结果进行十进制调整。
      MOVHA   /结果的BCD的最高有效字节保存在H中。
        •
        •
```

四位数字的 BCD 加操作的结果应该存储在何处呢？ 我们应该把 BCD 结果的最低有效字节存储在L寄存器里；BCD 结果的最高有效字节存储在H寄存器里。请记住，进位的内容也

可以是逻辑 1，这当然取决于相加的 BCD 数的大小。该进位位必须保存。

假设你需要把多精度的 BCD 数相减；例如， 522—146。这个减法操作应该怎样完成呢？首先，应该必须确定 146 的 9 的补码。然后加 1，于是，得到了这个数的 10 的补码。其后，这个被减数和减数的 10 的补码可以相加，并且可以进行调整。我们把这些操作归纳如下：

```
  99  99
-01  46        减数
  98  53        9 的补码
```

为了形成 10 的补码，我们必须将结果加 1。

```
  98  53
+00  01
  98  54
```

现在，我们可以把 146 (98 54)的 10 的补码加到 522 上。

```
  05  52
+98  54
  03  76
进位=1
辅助进位=0
```

两个两位 BCD 数相加以后，8080 必须执行一条 DAA 指令。当做这种减法运算时，我们可以使用例 5-25 所示的这个程序来产生寄存器对 B 的 BCD 内容的 10 的补码。

8080 执行这些指令以后，可以把寄存器对 B 的 BCD 内容加到寄存器对 D 的 BCD 内容上。因为我们已经把这些加法指令列入例 5-24 这个程序，那么，在 8080 确定寄存器对 B 的 BCD 内容的 10 的补码以后，就可以执行这些加法指令。实际

上，我们把这些指令可以存储在存储器里，如例 5-26 所示。

现在，可以随时调用 COMPB(寄存器对 B 的补码子程序)和 BPLUSD（寄存器对 B 的内容加寄存器对 D 的内容的子程序）这两个子程序。COMPB 子程序首先确定寄存器对 B 的 BCD 内容的 10 的补码。然后，8080 调用 BPLUSD 子程序，对寄存器对 B 的 BCD 内容与寄存器对 D 的 BCD 内容进行加操作。这会对从寄存器对 D 的 BCD 数减去寄存器对 B 的 BCD 数产生净效应。然而，BPLUSD 子程序也可以自己调用自己，把寄存器对 B 的 BCD 内容加到寄存器对 D 的 BCD 内容上。

如果我们把 BCD 内容存储在存储器里，那么，COMPB 和 BPLUSD 这两个子程序会有更强的通用性。如果使用这种方式，或长或短的 BCD 数都可以进行相加或相减。为了使存储在存储器的一个 BCD 数产生10的补码，可以让 8080 执行例 5-27 所列出的这个程序。

例 5-25 确定寄存器对 B 的 BCD 数的 10 的补码

```
/这个程序的这一部分用来计算
/寄存器对 B 的 BCD 数的 10 的补码。
/首先确定 9 的补码，然后，
/将结果加 1。
/以形成 10 的补码。
COMPB,  MVIA    /把 BCD数 99 送入 A 寄存器。
        231     /(八进制数 231＝十六进制数 99)
        SUBC    /减去减数的最低有效字节，
        MOVCA   /然后，把该结果存入 C 寄存器。
        MVIA    /再把 BCD数 99 送入 A 寄存器
        231     /(八进制数 231＝十六进制数 99)。
        SBBB    /带借位减去最高有效字节，
        MOVBA   /然后，把结果保存在 B 里。
```

```
        MOVAC   /现在，这个数加 1。
        ADI
        001
        DAA     /对该结果进行十进制调整。
        MOVCA
        MOVAB   /取这个数的最高有效字节。
        ACI     /加来自最低有效字节的任何进位，
        000
        DAA     /然后，对该结果进行十进制调整。
        MOVBA   /现在，把寄存器对 B 的 10 的补码
        .       /存入寄存器对 B 里。
        .
```

例 5-26 对寄存器对 B 的 BCD 内容进行补码操作，然后，把结果加到寄存器对 D 的 BCD 内容上。

```
COMPB,  MVIA    /把 BCD数 99 送入 A 寄存器里
        231     /(八进制数 231＝十六进制数 99)。
        SUBC    /减去减数的最低有效字节。
        .
        .
        .
        DAA     /对 A 的内容进行十进制调整。
        MOVBA   /把 10 的补码存入 B 与 C
BPLUSD, MOVAE   /把一个数的 BCD 最低有效字节送到 A。
        ADDC    /加另一个数的最低有效字节。
        DAA     /对该结果进行十进制调整。
        .
        .
        .
        DAA     /对这个最高有效字节进行十进制调整。
```

```
        MOVHA   /把这个 BCD 最高有效字节存入H。
        RET     /把 BCD 结果存入寄存器对 H，然后返回。
```

这个子程序要求寄存器对H指示到存储 BCD 数的两位最低有效位数字的存储单元，这个 BCD 数将要进行补码操作。而且这个数的较高的有效字必须按顺序被存储在较高的存储地址；必须把用来存储这个 BCD 数的存储单元的数目装入 C 寄存器。这个数字等于 BCD 数的位数的一半，因为每个存储单元存储两位 BCD 数字。

从根本上说，这个子程序有两个部分。第一部分用来确定 9 的补码；第二部分用来给 9 的补码加 1，以产生 10 的补码。

例 5-27 为存储在存储器的 BCD 数取 10 的补码子程序。

```
/这个子程序用来确定 BCD 数
/的 10 的补码；这个 BCD 数被存储在读/写存储器。
/把这个 BCD 数的最低有效字节的读/写存储器
/地址装入寄存器对H，从而进入这个子程序。
/这个 BCD 数的最低有效字节被存储在最低存储单元
/。C 存储的内容等于被压缩的 BCD
/字的数目 (BCD 数字数目的一半)
COMP 1, PUSHH   /把这个存储地址存入堆栈。
        PUSHB   /把 BCD 字数保存在堆栈里。
        XRAA    /把 A 寄存器和进位置零。
CMPNXT  MVIA    /把 BCD数 99 送入 A 寄存器里，
        231     /(八进制 231＝十六进制 99)
        SBBM    /从这个数减去存储单元的内容。
        MOVMA   /把 9 的补码保存在存储器里。
        INXH    /将这个存贮器地址加 1。
        DCRC    /将 BCD 数减 1。
```

```
        JNZ       /如果字数不等于 0，
        CMPNXT    /则转移，
        0         /以便找到另一个数的 9 的补码。
        POPB      /BCD 字数等于 0，从堆栈弹出该数和
        POPH      /这个存储地址。
        STC       /进位位置为逻辑 1。
ADD 1,  MOVAM     /取一个 9 的补码数值，
        ACI       /把进位和 000 加到这个数上。
        000
        DAA       /对该结果进行十进制调整。
        MOVMA     /把 10 的补码保存在存储器里。
        INXH      /存储地址加 1。
        DCRC      /BCD 数减 1。
        JNZ       /如果不等于 0，
        ADD 1     /必须把进位加到
        0         /另一个被压缩的 BCD 数上。
        RET       /找到了 10 的补码，返回。
```

在 COMP 1 点，把存放在寄存器对 H 的存储器地址和存放在 C 寄存器的存储单元数，都存入堆栈。然后，8080 执行 XRAA 指令，把进位和 A 寄存器清零。接着，把 BCD 数 99（八进制 231，十六进制 99）装入 A 寄存器，于是用 SBBM 指令，从 A 寄存器的内容（BCD 数）减去存储单元的内容（BCD 数）。在第一次通过这个子程序之前，这条 XRAA 指令已被执行，所以 SBBM 指令所产生的结果与执行 SUBM 指令所产生的结果是一致的，因为进位等于 0。但是，当 8080 连续几次执行这个循环时，8080 执行 SBBM 指令，把借位移到较高有效的 BCD 数。减法操作发生以后，把这个数的 9 的补码存入原来存储这个数的同一个存储单元。然后，将这个存储地址加 1，而 BCD

数减 1。只有当C寄存器的内容被减到0时，8080才不执行JNZ-CMPNXT指令。这就是说，把整个BCD数（不论多大）已经进行了9的补码操作。

这个程序的第二部分，正好在符号地址ADD 1之前开始，它用来确定存储在存储器的BCD数的10的补码。首先，存储器地址计数器的内容从堆栈弹出，装入C寄存器。用来存放BCD数的第一个存储单元的地址也从堆栈弹出，然后8080执行STC指令，把进位位置成逻辑1，接着，把第一个BCD数从存储器送到A寄存器。然后，8080执行AC 1000这条指令。为形成10的补码，难道不必把1加到9的补码上吗？回答是肯定的，这个子程序只要把进位位置成逻辑1，并且利用一条ACI指令，把逻辑1加入到BCD数上，就可以完成这个任务。8080第一次通过这个循环时，把进位加到A寄存器的内容上。因为进位被预置成逻辑1状态，所以，把1加到A寄存器的内容上。A寄存器的内容进行10进制调整之后，这个数的10的补码被存回到存储器里。然后，存储地址增1，存储单元计数器减1。若该计数器的值不等于0，这就是说，有更多的BCD数字需要取补，应执行JNZ-ADD 1指令。当C寄存器的内容最后减到0时，则8080从该子程序返回。这个数的10的补码现在被存储在读/写存储器（原来的数就存储在这里）。

一旦减数的10的补码形成以后，如例5-28所示，两个BCD数的加法运算是很简单的。你对这个子程序应该非常熟悉。它是整数加法运算的子程序之一，曾经在本章的整数加法运算一节中编写了这个程序，即例5-4所示的程序。这个子程序与例5-4的程序的唯一区别是：我们给它增添了一条DAA指令；最后的进位位再也不用检查。因为这个子程序与例

5-4 的程序有着基本相同的形式，所以，我们不必去解释这个子程序是怎样完成其功能的。

例 5-23　把存储在存储器的两个 BCD 数相加

```
/利用寄存器对H指示到要参与加法
/运算的一个 BCD 数；寄存器对 D 指示
/到另一个参与加法运算的数，这样来执行
/这个子程序。结果将被存储在存储器里，它是利用寄存器对 H 作为存储
 器地址的。
/寄存器对 C 用来存储参与加法操作
/的 BCD 字数(每个字为两位 BCD 数)。

BCDADD,  XRAA      /把 A 寄存器和进位清零。
ADD1,    LDAXD     /取一个被压缩的 BCD 字。
         ADCM      /把进位和另一个字相加。
         DAA       /对该结果进行十进制调整。
         MOVMA     /把该 BCD 结果保存在存储器里。
         INXH      /把存放在寄存器对 D 和 H
         INXD      /的存储地址加 1 。
         DCRC      /C 寄存器的字数减 1 。
         JNZ       /如果这个数不等于 0 ,
         ADD1      /那么，必须把另外两个字相加。
         0
         RET       /BCD 的答案存入存储器，
                   /然后返回。
```

为简单起见，本书中我们不讨论 BCD 数的任何乘法和除法子程序。读者回顾我们到目前为止已经看到的 BCD 子程序的例子，并不难理解。我们为什么使用 BCD 算术操作，有什么头等重要的理由吗？也许没有。虽然，BCD 数的使用，意味

着我们不必去编写 ASCII 字符转换成二进制，二进制转换成 ASCII 的子程序。但是，BCD 的具体算术子程序可能比等效的二进制子程序更长得多，复杂得多。此外，由四个字(32位)组成二进制数能够表示 0 之 4，294，967，295，之间的任何一个数同一个 32 位字只能代表 0 和 99，999，999 之间的任何一个 BCD 数。8 位，16 位，和 24 位字的存储容量如表 5-2 所示。

表 5-2　　二进制和 BCD 字的存储容量

字　　长	二　进　制	BCD
8 位	0—255	0—99
16位	0—65535	0—9999
24位	0—16,777,215	0—999,999

最后，对 DAA 指令进行几点说明是必要的。首先，使用 DAA 指令，算术加法操作必须在 A 寄存器中发生。读者不能把 BCD 数装入寄存器对 D 和 H，再用一条 DADD 指令来把这些 BCD 数相加；然后，把 L 或 H 寄存器的内容传送到 A 寄存器里，执行一条 DAA 指令来调整这个加法操作的结果，此外，把一个 BCD 数加 1，使用一条 ADIOOI 或一条 ACIOO1 指令。用INRA 指令，不能使 A 寄存器的内容加 1，用 DAA 指令，调整该结果。如果把一个 BCD 数减 1，那么，只要把 BCD 数 99 与它相加，然后调整其结果。最后，**你不能用 DAA 指令来执行二进制-BCD 数为基础的换算操作**。你也不能把两个二进制数相加，然后让 8080 执行一条 DAA 指令来把该结果转换成 BCD 数。**如果你使用D AA 指令，那么，其含义是：BCD**

数正在被进行加操作，即它们的10的补码正在进行“减”操作。

浮点算术操作

在本章，我们已经讨论过的定点算术例行程序和子程序，其局限性之一是：它们几乎不可能用来处理诸如 6.02×10^{23} 或 8.76×10^{-14} 的算式。这些数或者是因为太大，或者因为太小，不容易用三个或四个字的二进制数来表示。正是这个原因，我们要用浮点数和浮点算术程序。浮点数包括两部分：小数和指数。在前面这两个数中，“6.02”和“8.76”是小数；“23”和“−14”是这两个数的指数。在浮点格式中，不仅有指数符号，而且有小数符号。我们在讨论整数的算术例行程序时，曾经假设所有的数都是正数。

下面几个数在符号和数值上都相等吗？

6.02×10^{23}　　$.602\times10^{24}$　　60.2×10^{22}

是的，尽管这些数的指数和小数都不相同，但是它们是相等的。具体使用浮点算术程序指令时，上面这些数可以用8080微型计算机打印出来，如下所示：

6.02E＋23　　.602E＋24　　60.2E＋22

在这个例子中，这个E后面的数字表示带符号的十进制指数。为了表示浮点格式的任何一个数，我们可以使用三或四个8位的存储单元，或三、四个寄存器。用来表示浮点数的格式如下：

S_xXXXXXXX　　$S_f\cdot$FFFFFFF　　FFFFFFFF

S_xS_fXXXXXX　　$\cdot$FFFFFFFF　　FFFFFFFFF

在上面这个例子中，有一个 8 位字，S_xXXXXXXX。它包括：指数符号用 S_x 表示，浮点数 7 位指数用 XXXXXXX 来代表。总之，如果这个符号位是逻辑 1，那么，这个数是负的；如果这个符号位是逻辑 0，那么，这个数是正的。我们可以把这一概念应用于小数和指数的符号。小数的符号位是：S_f·FFFFFFF 这个字的一位。在这个具体的浮点数中，指数有 7 位，二进制小数有 15 位。在第二个例子中，指数和小数的符号位都包含在同一个 8 位字里。所以，指数仅仅是六位的二进制字，小数是一个 16 位的二进制字。

这些浮点数的中间的这个字，或字节，有一位二进制小数点，在各浮点数中都有这种小数点。在前面的例子中，二进制小数点是在小数的最高有效位的左边，这就是说，采用这种格式表示的浮点数以小数的形式表示。这些数的二进制小数点正如十进制的十进制小数点一样。为了满足浮点数格式的要求，6.02×10^{23} 和 8.76×10^{-14} 这两个数必须写成 $.602\times10^{24}$ 和 $.876\times10^{-13}$。如果把这个二进制小数点或十进制小数点向左移一位，相应的二进制数或十进制数的指数必须加 1。如果把二进制小数点，或十进制小数点向右移一位，其相应的二进制数或十进制数必须减 1。

读者从本章整数除法这一节关于小数问题的讨论中可以回忆起：如果离二进制小数点较近的这一位是逻辑 1，那么，这个数等于 0.5 或更大。二进制数离小数点越远，其他位位置的数的大小相应地按 0.5 的倍数依次减少。读者能够确定 24.133 这个数的适当的 8 位整数值和 8 位小数值吗？如果采用第一种浮点格式，其答案是：

00000101　　0.1100000

实际上，这个数就是 $(0.5+0.25)\times2^5$ 或 $.75\times32$，即 24；

用 8 位小数表示， 24 这个数最接近 24.133 这个值。下面这两个小数的值是多少呢？

0.1000000

0.1001000

第一个小数的值是 0.5；第二个小数的值是 .5625。如果必须把一个数与 56.25 相乘，那么，其等效的两个字组成的浮点数应该是什么呢？对于两个字的浮点数来说，我们可以假设是这种格式：S_xXXXXXXX S_fFFFFFFF。对于 56.25 这个数的浮点等效数应该是：

$$56.25=5.625\times10^1=.5625\times10^2$$

00000110　　0.1110000

这个二进制数是$(0.5+0.25+0.125)\times2^6$，或者是 0.875×64，即 56。因为指数和小数的符号都是正的，所以，小数和指数的 D_7 位必定是 0。如果采用与上面相同的浮点字格式(S_xXXXXXXX S_f·FFFFFFF)，那么，下面三个数的等效的十进制数是什么呢？

10000010　　0.101000

10000010　　0.101000

00000010　　1.101000

这三个数的等效的十进制数分别是：$.625\times2^{-2}$，即 0.15625；-0.625×2^{-2}，即 −0.15625；和 -0.625×2^2，即 −2.5。

采用浮点数所带来的问题之一是，人们仍旧在用十进制思维，而 8080 微型计算机用二进制“思维”(不论浮点数还是整数)。因此，浮点数转换程序通常有转换子程序，这样，才能把十进制数送入，并且转换成所需要求的浮点格式。在这个转换程序中，或许还可以调用另一个子程序，把十进制数等效的二

进制数浮动成适当的浮点格式。通常，也可以分别调用这些子程序。这样，为了使一个二进制数适当地浮动，我们不必让8080执行十进制的输入子程序。我们可以把浮动的这个二进制数留在浮点累加器(FPA)里。浮点累加器并不是 8080 微处理器集成电路芯片内的另一个累加器。相反，它只是读/写存储器的几个连续存贮器单元；浮点程序软件的子程序可以在这些存储单元取一个数或者存一个数，这个数将要进行算术操作。一般而言，浮点运算(加、减、乘或除)的某一运算的结果留在浮点累加器里。

通常，在浮点运算程序中有一个子程序，用来把浮点数转换成定点二进制数。这个子程序叫做定点子程序，因为它把这个数的二进制小数点固定。一般而言。8080 执行这个子程序，可以产生正整数值和负整数值。定点运算得到的二进制数，可以留在浮点累加器 (FPA)，或读/写存储器的另一部分的存储单元里。

在大多数浮点程序里，还有一个标准子程序。这个子程序用来完成下述转换功能：

指数	小数		指数	小数
00001011	0.0011011	转换成	00001001	0.1101100

在浮点运算中，被运算的数一定要在 $0.5 \leqq x < 1$ 这个范围之内。这就是说，我们必须把前例中的这个数左循环向移两位，从而，第一位是逻辑 1，它被循环移入 D_6 位。为什么不把逻辑 1 的第一位循环移入 D_7 位呢？请读者记住，小数的第一位通常是符号位；因此，我们不能把小数的第一位(逻辑 1)循环移入符号位。当然，在这一循环移位过程中，不能改变符号位。把第一位(逻辑 1)移入 D_6 位，那么，这个数必定是 0.5，或更大，但是，这个数不可能等于 1，因为不论它们有多少有效位，

二进制小数值只能接近 1。小数循环左移一位，指数一定要减 1。

最普通的浮点运算程序包（它是有用的浮点例行程序和子程序的集合体）可以用来执行四种基本的算术运算：加、减、乘和除。在本章的另一节，我们举了多精度加和减操作的子程序作的例子。与上述例行程序很类似的例行程序和子程序，可以用作浮点加操作或减操作子程序的起始点。

用浮点加运算，下面两个数可以直接相加：

$$\begin{array}{r} 5.02\times10^{13} \\ +4.17\times10^{13} \\ \hline 9.19\times10^{13} \end{array}$$

就数量级而言，这两个数是相等的。这是它们能够相加的唯一理由。下面这两个数也能相加：

$$\begin{array}{r} 5.02\times10^{13} \\ +2.43\times10^{12} \\ \hline 5.263\times10^{13} \end{array}$$

但是，如果你用笔和纸做这个加法运算，你也许会首先把 2.43×10^{12} 这个数变成 $2.43\cdot10^{13}$。用同样的方法，怎样把 5.02×10^{13} 和 3.79×10^{13} 这两个数相加呢？有时，你甚至可能把这两个数相加，因为 3.79×10^{3} 等于 $.000000000379\times10^{13}$，这个数已经超出了 5.02×10^{13} 这个数的有效值的范围。这两个数相加的结果应该是 5.0210^{13}。与 5.02×10^{13} 可以相加的最小的数或许是 1×10^{11}，可能是 5×10^{10} 这个数。比这个数小的任何一个数不会改变该加法操作的结果。不管相加的这两个数的指数的大小，如果这两个数不相等，必须使其中一个指数与另一个指数一致。对于 3.79×10^{3} 这个数来说，你应该使这个数的指数上升，直到它等于 13 时为止。你已经明白，为了提高指数

的值，必须把小数循环右移。为了使指数一致，我们必须在两个数中，找到较小的一个数。一旦找出了较小的数时，就应该使较小的这个数的指数增加，让小数循环右移即可。但是，作为调整指数的结果，3.79×10^{3} 这个数与 5.02×10^{13} 这个数比较，前者会显得无效。这就是说，不能把这两个数调整一致；也就是说，其中一个数比另一个数小得太多。如果我们不能把两个数调整一致，那么，就不能进行加法操作。两个数中较大的一个数，放在浮点数累加器（FPA）里。对于要进行的算术运算而言，我们必须把两个数之一放在浮点数累加器里。因此，如果我们不能把这两个数调整一致，那么，较大的一个数已经放在 FPA 里，所以，在浮点加法操作的子程序中不会发生别的操作。

如果我们可以把这两个数的指数调整一致，那么，就能进行加操作。但是，在执行加运算之前，我们必须把两个数的小数部分向循环右移一位。这样做就可以防止两个小数的高七位有效字节相加所产生的进位进入该小数的最高有效字节的 D_7 位。请读者记住，小数的最高有效字节的 D_7 位是小数的符号位。假设我们要把下面两个数(浮点格式的两个数)相加：

指数	
00000010	0.1011100
00000010	+0.1001000
00000010	1.0100100

尽管被相加的两个数都是正数，但是，现在结果的符号是负的，所以，该结果是错误的。为了正确地把这两个数相加，首先必须把它们循环右移，把指数增加 1 。

指　数	小　数
00000011	0.0101110

```
00000011          +0.0100100
00000011
                   0.1010010
```

这个加法运算的结果是什么呢?该结果等于$(0.5\times0.125+0.015625)\times2^3$或$0.640625\times8$，即5.125。在这两个数循环移位之前，它们的值是0.71875×2^2(即2.875)和0.5625×2^2,即(2.25)。

当进行加法操作时，结果的符号还是绝对值最大的那个数的符号。例如，5+(−3)=2。这个最大的数是5，所以结果的符号仍证明是正的。对于3+(−7)而言，7是最大的数,所以，该结果的符号与这个−7的符号相同，因而是负的。当然，3+(−3)是一种特殊情况，其结果的符号是正的。在一些浮点运算程序包中,这种加法运算的结果会是负的；也就是说，有正0和负0之分!

既然我们已经讨论了浮点加法运算的概念，那么，理解浮点减法运算的过程也是容易的。为了把两个浮点数相减，可以求出减数的二进制补码，然后，调用浮点加法运算的子程序。做加法操作时，“减法”操作的结果保持不变。这个结果可以留在浮点累加器(FPA),也可以不留在FPA。之所以可以这样做，是因为：

$$25-13=25+(-13)$$

实际上，浮点乘法和浮点除法比浮点加法容易。这句话是正确的。因为乘数和被乘数的小数不需要做到一致；在做乘法运算之前，也不需要把这些数循环右移一位。乘法运算、只把这两个数的指数相加，而小数部分，如同整数乘法一样，进行相乘。浮点除法,将指数相减,小数相除。你已经看到了一些整数乘法和除法运算的子程序。这些子程序可以用来对那些正在进行乘法运算或除法运算的浮点数的小数部分进行运算，与这

两种运算有关的唯一比较难的工作是，记录其结果的小数的符号。

当进行乘法运算时，小数的符号组合可以如下所示：

乘数	被乘数	结果
+	+	+
+	−	−
−	+	−
−	−	+

在我们已经讨论过的浮点格式中，负号用逻辑 1 表示，所以，我们可以得到下面这个真值表：

乘数	被乘数	结果
0	0	0
0	1	1
1	0	1
1	1	0

从这个真值表，你会看到，只要把乘数的符号和被乘数的符号进行“异”操作，就可以确定结果的符号。你应该记得，两个 16 位的数相乘时，结果是 32 位。所以，对于浮点乘法运算而言，在进行乘法运算之后，还必须保留一些存储单元，供浮点累加器(FPA)用，这些存储单元的数量应该倍增。对于由三个字构成的小数来说，必须保留六个连续的存储单元，来存储 FPA 的小数。

浮点除法只是稍微复杂一点。小数除法运算可以用一个整数除法子程序来完成。这些整数除法子程序已经在本章前面一节讨论过了。除法运算结果的指数应该是从被除数的指数减去除数的指数所得到的结果。如果除数和被除数的符号相同，那么，除法运算的结果是正值如果符号不同，那么，除法运算的

结果的符号是负的。

制作通用浮点运算的程序包所需要掌握的许多要点，我们还没有讨论；例如，可能产生的错误的情况，及其如何防止的办法。假设乘法运算或加法运算的结果所需要的存储空间比浮点累加器 FPA 的大，那么它能保存该果吗？能用零做除数吗？这些问题以及关于浮点运算程序包的有趣的特征，我们在本章末推荐了几本参考书，供读者参考。

特 殊 的 功 能

有些浮点运算程序包不仅可以用来完成四种基本功能，它们还能用来求对数、角的正弦和余弦。关于用来求这些运算的结果所用的公式，我们将在本章最后一节予以归纳。

正弦

对于 x 的各个真值；(以弧度表示的)可以用一个级数来求出 sin(x)：

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} \cdots$$

用另一种方法可以写成：

$$\sin x = x - 0.166666(x^3) + 0.008333(x^5) - 0.000198(x^7) + \\ + 0.0000028(x^9)$$

余弦

对于 x 的各个实数值，以弧度表示的角的余弦也可以用级数来计算：

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} \cdots$$

这个方程式也可以简化成：

$$\cos x = 1 - 0.50000(x^2) + 0.041667(x^4) - 0.001389(x^6) + + 0.000025(x^8)$$

正切

角的正切(用弧度表示)可以用计算角的正弦和余弦的方程式来计算，我们可以利用下面这个恒等式：

$$\tan x = \frac{\sin x}{\cos x}$$

对数

自然对数也可以用级数展开式来进行计算：

$$\log_e(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4}$$

这个等式用来求值的范围是：$-1 < x \leqq 1$，为了计算底数是 10 的对数值，可以使用前面的式子，然后把所得到的结果乘以 2.30258，

$$\log_e x = \frac{\log_{10} x}{\log_{10} x} = (\log_e 16)(\log_{10} x)$$

$$= 2.30258 \log_{10} x$$

到现在为止，我们已经见到了许多可以用来处理算术运算的算术程序和子程序。这些程序和子程序可以用于会计，勘测、解决科学问题，航行和商业等方面的应用，是使用整数还是浮点算术程序，应该视具体应用而定。

参 考 文 献

1. Wadsworth,N. *Machine Language Programming for the* 8008 *and Similar Microcomputers.* Scelbi Computer Consulting, Inc.,Milford,CT,1975.
2. *PDP-8 Floaiing Point System Programmer'sReference Manual*,No. DEC-08-YQYB-D. Digital Equipment Corporation,Maynard,MA,1969.
3. Rony,P. R.,Titus,J. A.,Titus,C. A.,and Larsen,D. C."Microcomputer Interfacing:Integer Addition and Subtraction." *American Laboratory*,VoI. 10,No. 2,1978,p. 153.

第六章　数制的转换

任何一种计算机都必须有一输入数字信息的方法。这种信息可以是数据或指令。它们来自存储器，电传打字机，卡片阅读机，数字录音机或软磁盘。这些信息可以是ASCII码，EBCDIC（扩展的二进制编码的十进制交换码），二进制或葛莱码。为使8080便于处理这些信息，要求这些信息一定是由一串逻辑1和逻辑0构成。但是，当把指令或数据送入微型计算机时，程序设计员要记住一串串的逻辑1和逻辑0，是十分困难的。正是这个原因，我们用八进制和十六进制数制来代替这些数据串。

三位ASCII八进制数一二进制数的转换

我们要讨论的数制转换的第一个问题，是把三位八进制数转换成8位二进制数。八进制数可以用三个ASCII字符(ASCII为基础）来表示。操作员将电传打字机或CRT显示器上编号为0～7的这些键按下；然后，微型计算机将接收这样产生的ASCII字符；利用微型计算机程序，把这些ASCII字符压缩成一个8位的字；再把这些二进制信息存入一个寄存器或一个存储单元。例如，按顺序按下数字键3、2和3时，二进制数11010011必定被保存在存储器里，因为$323_8=11010011_2$。如果依次把0～7的各只键按下，那么，电传打字机所产生的

代码如表 6-1 表示。

幸好，不论用八进制还是十六进制，其代码的最低有效数字与被按下的这只键所代表的数是一致的。请读者注意，任何一个键被按下时，电传打字机将发送表 6-1 所列出的相应的二进制数。如果按顺序按下 1，2，5 这三个键，电传打字机将要发送 261（B 1）、262（B 2）和 265（B 5）这三个代码。微型计算机程序必须屏蔽高 5 位有效位，不传送它们，然后，分别把其余各位信息压缩成两位较高的有效位，三位中间位和三位较低有效位。最后把这个 8 位字节放入寄存器或存储单元。

表 6-1　ASCII 字符 0～7 的八进制、十六进制和二进制值

键	字符值		
	八进制	十六进制	二进制
0	260	B 0	10110000
1	261	B 1	10110001
2	262	B 2	10110010
3	263	B 3	10110011
4	264	B 4	10110100
5	265	B 5	10110101
6	266	B 6	10110110
7	267	B 7	10110111

如果依次按下 1，2 和 5 这三只键，那么，必定会将二进制数 01010101 保存起来。8080 微处理器接收的第一个字符是 ASCII 的“1”。它把高六位有效位屏蔽，然后，把其余两位循环移入 A 寄存器的最高有效位，如例 6-1 所示。

例 6-1　把一个数输入和保存在 A 寄存器的两位高有效位的例行程序

CALL/从电传打字机取一个字符。

```
TTYIN
0
ANI  /去掉6位高有效位，
003  /保存两位低有效位。
RRC  /把这个数循环右移，进入这个
RRC  /字的最高有效位。
.
.
.
```

8080 执行CALL 指令，调用 TTYIN 子程序，该子程序从电传打字机或 CRT 输入一个字符。然后，8080 执行 ANI 003 指令，把A寄存器的六位高有效位置 0。然后，把A寄存器的内容循环右移两次。这样，使 D_0 位移入 D_6 位，D_1 进入 D_7 位。当然，8080 执行 6 次 RLC 指令，也能把这些位移入 D_6 位和 D_7 位。但是，这意味着这个程序的长度会增加四条指令。既然，ACSII 字符“1”已经被接收和处理，接着将要接收 ASCII 字符“2”，所以，把下一个 ASCII 字符输入到 8080 之前，A寄存器所存储的内容 01000000 这个数必须保存起来。为了接收和处理三位八进制数，可以使用例 6-2 所列出的这个程序。这个程序要求输入键盘上的键 0～7 的三个 ASCII 字符。这个程序不能检验无效键如“A”，“？”或“9”。

如果不用这条 ANI 003 指令，那么，可以用另外哪一条两个字节的指令呢？双字节指令 SUI 060 可以使用。

例 6-2 ASCII 码八进制-二进制转换子程序

```
OCTIN,  CALL  / 从电传打字机
        TTYIN        / 取一位最高有效数字。
        0
```

```
ANI         / 从这个 ASCII 值(1010 XXX)
003         / 中去掉 6 位高有效位。
RRC         / 把这个数循环右移，
RRC         / 进入这个字的最高有效位。
MOVCA       / 把这个数保存在 C 寄存器。
CALL        / 从电传打字机取
TTYIN       / 下一个字符。
0
ANI         / 去掉这个 ASCII 值(10110 XXX)中
007         / 5 位高有效位。
RLC         / 把这个数循环左移，
RLC         / 进入 A 寄存器的中间三位
RLC
ADDC        / 把第一个数字加到它上面。
MOVCA       / 把这个组合的字保存在 C 寄存器里。
CALL        / 现在，取这个字的
TTYIN       / 最低有效数字。
0
ANI         / 去掉这个 ASCII 值(10110 XXX)中
007         / 高有效位。
ADDC        / 把这些字组合起来，
RET         / 现在，这个二进制值存储在 A。
```

请读者记住，TTYIN 子程序把从电传打字机接收到的各个字符的校验位（D_7位），（请参考第三章）都屏蔽起来。因此，如果 TTYIN 这个子程序所接收并处理的数据是 260（B 0），那么，当程序执行控制回到调入程序时，060（30）被存储在A寄存器里。如果我们使用例 6-2 所列出的这个子程序，那么，8080 从该子程序返回时，A寄存器的内容是什么呢？如果电传打字机上的 1，9，和 2 这三个键按照给定的顺

序按下时，A寄存器的内容又是什么呢？A寄存器的内容应该是112（4A）。为什么？请记住，当输入一个数时，这个数与立即数字节003或（03）或007（07）进行“与”操作。因为“9”这只键是被按下的第二只键，所以，这只键所代表的ASCII值与007（07）进行与操作。因为9这只键的ASCII值是271，

```
00111001      A的内容
00000111      ANI数据字节
00000001      ANI被执行后
              A寄存器的内容
```

所以，A寄存器的内容应该是001（01）。因此，在中间这三位，不是二进制数1001（ASCII字符9），而是001。当然，如果我们已经把ASCII的八进制转换为二进制的程序写好了，那么，8080不会去处理ASCII字符8和9，因为这两个数不是有效的8进制数。实际上，8080只处理0到7这些ASCII字符，而对其他ASCII字符不予处理。因此，例6-3所列出的这个转换子程序是ASCII字符八进制转换成二进制的改进程序。

例 6-3 ASCII八进制-二进制转换子程序

```
OCTIN,  LXID    /把000 003（0003）
        003     /装入寄存器对D。E
        000     /存储003（03），D存储000（00）。
OCTIN 1, CALL   /从电传打字机或CRT取一个
        TTYIN   /字符（它将被打印在电传打字机
        0       /或显示在CRT上）。然后存入A，返回。
        CPI     /校验位（D7）=0
        060     /这个值小于ASCII值0吗？
        JC      /是的，则忽略这个字符。
        OCTIN 1
```

```
0
CPI       /这个字符等于或大于
070       /ASCII 字符 8 吗?
JNC       /是的，那么进位被消除，所以
OCTIN 1   /如果这个数等于或大于 ASC II
0         /码 8 。则转移。
ANI       /对，它是 ASCII 码 0～7，
007       /把各位 (D2，D1 和 D0 除外)
          /置 0 。
MOVBA     /把这个数暂时保存在 B 寄存
          /器里。
MOVAD     /取前面的数字，然后把
RLC       /它们循环左移三次。
RLC       /这样，会使它们的值增加，
RLC       /并且为刚刚输入的这个数准
          /备存储单元。
ADDB      /加刚输入的这个数。
MOVDA     /把这个新的二进制数保存在 D。
DCRE      /数字计数器减 1。
JNZ       /如果这个数不等于 0，
OCTIN 1   /取另一个字符。
0
RET       /把这个二进制数存入D，而返回。
```

这个子程序的第一条指令用 000 003 (0003) 给寄存器对 D 预置初始值。003 (03) 被装入 E 寄存器里。E 寄存器将用来作为数字计数器；即可以被压缩成一个 8 位字的 ASCII 八进制数字的位数。把 000 (00) 装入 D 寄存器，因为它将供暂时存储用。在 OCTIN 1 的开始，8080 调用 TTYIN 这个

子程序，从电传打字机接收一个字符。当8080从TTYIN子程序返回时，A寄存器的内容正是刚从电传打字机接收到的这个ASCII字符，立即与060（30）这个数进行比较。060(30)是0键所表示的7位ASCII值。如果8080所接收的这个ASCII字符比ASCII键0所表示的值小，那么,8080执行JC-OCTIN1指令。该子程序的这段指令使8080忽视其值小于060的所有的ASCII字符。

这个子程序的第二条指令CPI把A寄存器的内容与070（38）这个数值进行比较。这条指令可以用来确定A寄存器的内容是不是067（2F），或者更小。因为ASCII字符0～7所表示的数值是在060和067（30和37）之间，所以，如果接收0～7的某个ASCII字符，在第二条指令执行后，标识位将是真。如果键盘上8这只键被按下，或产生的键代码的值大于067的其他任何一只键被按下，那么，8080将执行JNC指令，然后，调入TTYIN子程序。8080执行这四条指令，把不代表ASCII字符0～7的各个值都忽略。这是对使用连续比较，滤掉一组值，而不是一个值的良好的说明。

只有把0～7这些键的某一个按下，8080才执行ANI指令。这条ANI 007指令只把数据位D_0，D_1和D_2存入A寄存器。然后把这个二进制数存入B寄存器里。再把前面被压缩的1、2位数字值从D寄存器（暂时存储寄存器）送到A寄存器。然后执行三次RLC指令，从而，在A寄存器的D_0，D_1和D_2的位置上留出了“空间”，供存储那些刚刚输入的ASCII码八进制数字的二进制代码使用。8080执行这三条RLC指令后，把B寄存器的内容加到A寄存器的内容上，因为只有A寄存器的D_0，D_1和D_2不是0（由于使用了ANI指令），所以，我们用ADDB指令，把前面的二进制数值与刚刚按下的这只键所

表示的二进制数值结合。我们也可以使用 ORAB 指令。8080执行 ADDB 指令之后，把这个新的八位字存入D 寄存器里；然后，E 寄存器的数字位数 减 1。8080 执 行 DCRE 指 令 之 后，如果 E 寄存器的内容不等于 0，则执行 JNZ-OCTIN 1 指令，这时，8080 等待电传打字机输入另一个字符。只有 在 按 了电传打字机的键盘上的三个有效键（0～7）之后，这就是说 E 寄存器的内容已经被减为 0，8080 才执行 RET 返回指 令。8080 执行这条 RET 指令时，把被输入的三个八进制数字 的 等效二进制数存储在 D 寄存器里。

如果从电传打字机的键盘上输入 477 这个数，那么，D寄存器的内容应该是什么呢？因为寄存器只能存储一个 8 位字长的数，所以，D寄存器的内容应该是 077。这种情况如下所示：

477		077
100111111	变为	00111111

因此，如果输入 777 这个数，则D寄存器的内容应该是 11111111_2，即 377。

八位二进制数-ASCII 八进制数转换

二进制数转换成 ASCII 字符的八进制 数 的转 换子程序与 ASCII 字符的八进制数转换成二进制数的转换子程序不同：后一个转换子程序可以使用一条通用的屏 蔽指令（ANI 007）和三条 RLC 指令；前一个转换子程序却不能使用这种技术。若要弄清为什么，请看八进制数 156：

01 101 110

为了把一个二进制字转换成 ASCII 字符的八进制数字，必须把 ASCII 字符 0～7(260～267) 打印在电传打字机上，或显示在 CRT 上。为此，我们必须把被打印的数(000_2到111_2)，传送到A寄存器的低有效位。然后，把 260 这个数加到A寄存器的内容上，并打印合成的 ASCII 数字字符。为了把 01101110 这个二进制数据的位数放在适当的位置上，应该把两位最高有效位的 01 循环右移六次，或循环左移两次；必须把中间这三位 101 循环右移三次，最低有效位 110 根本不用移动。寄存器对H所寻址的存储单元的二进制内容适当地被转换成三个 ASCII 字符，然后用电传打字机进行打印，或显示在 CRT 上，这个子程序如例 6-4 所示。

这个程序例的第一条指令把寄存器对H所寻址的那个存储单元的内容装入 8080 微处理器芯片内的A寄存器里。ANI 指令用来屏蔽A寄存器除D_6和D_7之外的各位。把这两位循环左移两次，这样，使这两位进入了A寄存器的D_0和D_1的位置上。8080 然后调用 BCDOUT 这个子程序，把 260 这个数加到A寄存器的内容上，然后把它在电传打字机上打印出来。260 被加到A寄存器的内容之前，A寄存器所存储的内容只能是八进制数 000,001,002 或 003，因为 8080 执行 ANI 300 这条指令，只把两位留在A寄存器。8080 把 260 加到A寄存器的内容之后，其结果可能是 260，261，262 或 263。

例 6-4 二进制-ASCII 八进制转换子程序

```
/这个子程序用来把存储在寄存器对H所寻址
/的存储单元的二进制数
/转换成以
/ASCII码的三位八进制数。
BINOCT, MOVAM / 从存储器取一个二进制数。
```

```
        ANI     / 必须首先打印
        300     / 这个最高有效数字。
        RLC     / 把这两位最高有效位
        RLC     / 循环移入最低有效位。
        CALL    / 现在，调用 BCDOUT 子程序。
        BCDOUT  / 它把 260 加到A寄存器
        0       / 的内容上，并打印其结果。
        MOVAM   / 现在,必须打印中间这一位
                / 八进制数字。
        ANI     / 屏蔽各位，(中间这三位二
        070       进制数除外)。
        RRC     / 然后把这三位循环移位三次，
        RRC     / 进入三位最低有效
        RRC     / 位的位置。
        CALL    / 把 260 这个数加到
        BCDOUT  / A寄存器的内容上，然后打
        0       / 印其结果。
        MOVAM   / 现在，打印右边这位八进制
                / 数。
        ANI     / 屏蔽各位，(三位
        007     / 最低有效位除外)。
        CALL    / 然后，把 260 加到这个数上
        BCDOUT  / 并打印这个字符。
        0
        RET     / 从这个子程序返回。
BCDOUT, ADI     / 加 260 到A的内容上。
        260     / (260=十六进制数 BO)。
        CALL    / 把这个 ASCII 字符在
        TTYOUT  / 电传打字机上打印或在 CRT 上
        0       / 显示。
```

```
        RET       / 然后从 BCDOUT 返回。
TTYOUT, MOVBA     / 把这个字符保存在 B 寄存器
                  / 里。
TTYO,   IN        /输入"UART"的状态字。
        001
        ANI       / 只保存发送器的标识位。
        004       / 如果 A＝004，发送器（打
                  / 印机）准备好。
        JZ        / 如果 A＝000，发送器（打印
                  / 机）不空，
        TTYO      / 所以，继续等待，等发送器(打
        0         / 印机）完成操作后，
        MOVAB     / 才能打印 A 寄存器的内容。
        OUT       / 这个字符从 B 寄存器送到
        000       / A 后，把它输出给"UART"。
        RET       / 把字符留在 A，然后返回。
```

加法操作之后，8080 调用 TTYOUT 子程序，该子程序把 A 寄存器存储的 ASCII 字符打印在电传打字机上，或显示在 CRT 上。8080 使用这个 TTYOUT 子程序打印这个字符之后，程序执行控制然后返回到 BCDOUT子程序的 RET 返回指令。然后 RET 返回指令把程序执行控制送回到 BINOCT 这个子程序的第二条 MOVAM 指令。只要把 BCDOUT 这个子程序的 CALL～TTYOUT 指令改换成 JMP～TTYOUT 指令，就可以把 BCDOUT 子程序简化。BCDOUT 子程序末尾的 RET 指令可以省略掉，如例 6-5 所示。

在 8080 执行例 6-4 所列出的程序的过程中，它调入 BCDOUT 子程序时，把一个返回地址保存在堆栈里。当 8080

调入 TTYOUT 这个子程序后，把第二个返回地址保存在堆栈里。然后，TTYOUT 这个子程序末尾的那条 RET 返回指令使 8080 返回到 BINOCT 子程序；这个子程序末尾的那条RET 指令使 8080 返回到 BINOCT 子程序。只要把 CALL-TTY-OUT 指令改变成 JMP-TTYOUT，当 8080 调用 BCDOUT 这个子程序后，就把一个返回地址保留在堆栈里。然后，8080 执行 JMP-TTYOUT 这段指令，把 A 寄存器的内容打印在电传打字机上。接着，8080 执行 TTYOUT 子程序末尾的 RET 指令，使程序执行控制返回到 BINOCT，而不是返回到 BC-DOUT。请读者注意，BCDOUT 和 TTYOUT 这两个子程序仍然是通用子程序，并且它们除了由 BINDUT 子程序调用之外，还可以为其他任何程序或子程序所调用。由于程序中是转移到 TTYOUT 子程序，而不是调用它，所以，在 BCDOUT 的末尾不需要使用一条 RET 指令，从而节省了一个存储单元；并且可以节省堆栈所用的两个读/写存储器单元，只有一个返回地址被保存在堆栈里，而不是两个返回地址。

例 6-5 修改了的更简单的 BCDOUT 子程序

```
BCDOUT,  ADI/把 260 (B 0) 加到 A 寄存
           260/器的内容上，以产生一
              /个 ASCII 字符。
         JMP  /然后转移到电传打字机或 CRT
              TTYOUT        /的打印子程序，打印
              0             /A 寄存器的 ASCII 字符。
```

如果你曾经用下述方式结束一个子程序：

```
CALL
XXX
YYY
RET
```

那么，可以写成下面的形式而使之简化：

```
JMP
XXX
YYY
```

8080 在电传打字机上打印第一个字符以后，程序执行控制返回到 BINOCT 这个子程序。把寄存器对 H 寻址的同一个存储单元的内容装入 A 寄存器。但是，这一次，把 A 寄存器的内容与 070 进行与操作。这就是说，A 寄存器只有中间这三位（D_5、D_4 和 D_3）数据不等于零。然后，把这三位循环右移三次，进入 D_2、D_1 和 D_0 的位置，然后，8080 调用 BCDOUT 这个子程序：打印存储单元的内容的中间三位（D_5、D_4 和 D_3）的 ASCII 等效值。一个 8 位字的其余三位如何处理的问题，是不难明白的。我们可以把这个子程序叫强制方法，因为这个子程序并没有使用任何方便的循环，使这个程序更为简洁。

例 6-6 所列出的这个程序，使用一个循环来把二进制数转换成 ASCII 字符八进制数。这个子程序不仅比例 6-4 所列出的子程序长，而且更复杂。正是这个原因，我们将不对它进行详细说明。

例 6-6 二进制-ASCII 八进制的转换子程序（带有一个循环）

```
/我们在这个子程序中，用了一个循环，
/把存储在寄存器对 H
/所寻址的存储单元的二进制数，转换成一个以
/ASCII 字符为基础的三位八进制数。
BINOCT, MOV AM/从存储器取一个二进制字
        ANI     /屏蔽各位（最高有效位除外）。
        300
```

```
        MVIC    /把 002 装入移位位数计数器。
        002     /(每个数移动三次)。
        CALL    /这样，会使A寄存器的内容
        BNOCT   /循环右移六次。
        0       /然后，加 260(B 0)，并打印其
                /结果
        MOVAM   /再取一个二进制字。
        ANI     /屏蔽各位（中间三位除外)。
        070
        MVIC    /把移位位数计数器置 1，
        001     /所以，只进行三次循环移位。
        CALL    /8080 执行循环移位指令后，
        BNOCT   /加 260，并打印其结果。
        0
        MOVAM   /再取一个二进制数。
        ANI     /不需要进行移位，
        007     /直接转向加操作指令（ADI)。
        JMP
        BNOCT 1
        0
BNOCT,  RRC     /把A寄存器的内容
        RRC     /循环右移三次。
        RRC
        DCRC    /计数数字减 1。
        JNZ     /如果不等于 0，把它再循环移位
        BNOCT   /三次。
        0
BNOCT1, ADI     现在，把 260(BO) 加到A寄
        260     /存器的内容上。
        JMP     /然后，转移到电传打字机子
```

TTYOUT/程序。该子程序在前面例子中
0 /已经被用过了。

例 6-7 二进制字转换成 ASCII 八进制字：两种方法的比较

老方法—不用循环

```
BINOCT, MOVAM
        ANI
        300
        RLC
        RLC
        CALL
        BCDOUT
        0
        MOVAM
        ANI
        070
        RRC
        RRC
        RRC
        CALL
        BCDOUT
        0
        MOVAM
        ANI
        007
BCDOUT, ADI
        260
        JMP
        TTYOUT
```

新方法—使用循环

```
BINOCT, MOVAM
        ANI
        300
        MVIC
        002
        CALL
        BNOCT
        0
        MOVAM
        ANI
        070
        MVIC
        001
        CALL
        BNOCT
        0
        MOVAM
        ANI
        007
        JMP
        BNOCT 1
        0

        BNOCT,  RRC
```

```
0               RRC
                RRC
                DCRC
                JNZ
                BNOCT
                0
          BNOCT 1, ADI
                260
                JMP
                TTYOUT
                0
```

我们在例 6-7，采用了逐步对照的方法将这两个子程序进行了比较。读者可以看到，老方法，即流水线法所需要的存储单元少几个，所需要的寄存器少一个。很明显，在一个程序中，使用一个循环，也许并不总是编写程序或子程序的最好方法。用来把二进制转换成以 ASCII 码的八进制的最简短的转换子程序之一，如例 6-8 所示。

例 6-8 二进制数-ASCII 八进制数的简洁的转换子程序

```
/这个子程序把存储在寄存器对 H
/寻址的存储单元的二进制数转换成
/ASCII 的三位八进制数。
BINOCT, MOVAM  /从存储器取一个二进制数。
        ANI    /首先必须打印
        300    /最高有效数字。
        RLC    /把两位最高有效位循环
        RLC    /移入最低有效位位置。
        CALL   /现在，8080 调用 BCDOUT 子程
        BCDOUT /序，它把 260 加到A寄存器
```

```
          0          /的内容上，然后打印之。
          MOVAM      /打印中间这位数字。
          ANI        /屏蔽各位（中间这三位数据
          070        /除外）。
          RRC        /然后，把这三位循环移位三次，
          RRC        /移入三位最低有效位
          RRC        /的位置。
          CALL       /把 260 加到A寄存器
          BCDOUT     /的内容上，并打印其结果。
          0
          MOVAM      /现在，打印右边这位数字。
          ANI        /屏蔽各位（三位最低有效位
          007        /除外）。
BCDOUT,ADI           /把 260 加到A寄存器的内
          260        /容上（260＝十六进制 B 0）。
          JMP        /把这个 ASCII 字符
          TTYOUT     /在电传打字机上打印，
          0          /或在 CRT 上显示。
```

两位 ASCII 的十六进制数-二进制数转换

有些程序设计员在给 8080 微型计算机编程序时比较喜欢使用十六进制数制，而不喜欢使用八进制数制。因此，我们需要介绍以 ASCII 码为基础的十六进制转换成二进制和二进制转换成十六进制的转换子程序。为了方便那些尚未熟悉十六进制数制的读者，我们给出了表 6-2。

9 这个数之后，我们用字母A来表示 1010，即用A代表 10

表 6-2　　十六进制数制

十六进制字符	二进制数值	ASCII 值
0	0000	260　B 0
1	0001	261　B 1
2	0010	262　B 2
3	0011	263　B 3
4	0100	264　B 4
5	0101	265　B 5
6	0110	266　B 6
7	0111	267　B 7
8	1000	270　B 8
9	1001	271　B 9
A	1010	301　C 1
B	1011	302　C 2
C	1100	303　C 3
D	1101	304　C 4
E	1110	305　C 5
F	1111	306　C 6

这个数。请注意，9(271，B9) 这个字符和A(301，C1) 这个字符的 ASCII值之间的非连续性。对于表 6-2 所给出的ASCII值，我们已经假设 8 位数据值的最高有效位总是逻辑 1。这位最高有效位是ASCII 值的校验位。这一位既可以是逻辑 1，也可以是逻辑 0，这要视所用的电传打字机或 CRT 而定。在我们的程序例中，我们已经假设这一位是逻辑 1。

在前面一个例子（例 6-2），把 ASCII 字符 0～7 转换成二进制数，是非常容易的事情。实际上，这个子程序只要求使用一条“与”屏蔽指令和几条循环指令。把以 ASCII 码为基础的十六进制字符转换成二进制却需要比较复杂的转换子程序，因

为这些字母键所表示的 ASCII 值的四位最低有效位与所要求的二进制数不一致。例如，我们必须把 ASCII 字符 A（1100 0001 转换成 00001010。使用十六进制所带来的优点之一是：每个 8 位的二进制数只需要两位十六进制数就能代表。读者知道，每一个 8 位的二进制数需要三位八进制数来表示。一般来说，这一点是在给 8080 编程序时，使用十六进制而不使用八进制的最强力的证据。二进制，八进制和十六进制数的一些典型的数如表 6-3 所示。

表 6-3　二进制、八进制和十六进制数的一些典型数

二　进　制	八　进	十　六　进
00000000	000	00
01000000	100	40
10000000	200	80
11001010	312	CA
11111111	377	FF

把从电传打字机或 CRT 显示器的 ASCII 字符键盘所得到的两个十六进制数转换成 8 位二进制数的子程序如例 6-9 所示。8080 随时都可以调用 HEXBIN 子程序。8080 调入该子程序时，可以从电传打字机或 CRT 键盘送入两个十六进制数。8080 从该子程序返回时，将把等效的二进制数存储在 A 寄存器里。

例 6-9　ASCII 十六进制数-二进制数转换子程序

```
/这个子程序把两个 ASCII 十六进制字符
/转换成一个八位二进制字。
HEXBIN, CALL      /从电传打字机的键盘
```

```
        HEXIN   /取一个十六进制字符（ASCII）
        0
        RLC     /把这个等效二进制数
        RLC     /循环左移四次，进入高四
        RLC     /位有效位的位置。
        RLC
        MOVCA   /把这个值暂时保存在C
        CALL    /从键盘取第二个即最后
        HEXIN   /一个 ASCII 十六进制字符。
        0
        ADDC    /把这两个十六进制字加到一
        RET     /起，再把结果存入A，返回。
HEXIN,  CALL    /从键盘取一个 ASCII 字符。
        TTYIN
        0
        CPI     /这个字符小于 060(0)吗？
        060
        JC      /是的。则置之不理。
        HEXIN
        0
        CP I    /它比 0～9 这些数小吗？
        072
        JC      /是的，输入一个数（0～9）后，
        NOLET   /所以，转移到 NOLET。
        0
        CP I    /它不是 0～9 的某个数，是 A～F
        101     /的一个字母？小于 ASCIIA 吗？
        JC      /是的，则置之不理。
        HEXIN
        0
```

```
        CPI     /它比 ASCIIG少吗?
        107
        JNC     /不，则置之不理。
        HEXIH
        0
        ADI     /把 011(09) 加到A～F，把它
        011     /们转换成等效的二进制数。
NOLET,  ANI     /现在只保留四位最低有效位
        017
        RET
```

8080 立即调入 HEXBIN 这个子程序。在该子程序的开头，8080 调用子程序 TTYIN，所以，8080 等待 CRT 显示器或电传打字机的键被按下。如前面所示的那样，TTYIN 子程序用来输入 ASCII 字符，然后，8080 执行一条 ANI 177 指令，将这个ASCII 的校验位(D_7)屏蔽。当程序执行控制返回到HEXIN 这个子程序时，如果按下的键是一个有效的十六进制数，则A 寄存器存储的内容可以是表 6-4 所列出的十六进制数值的任何一个。

8080 从 TTYIN 子程序返回后，CPI 060 这条指令检查其 ASCII 值小于或不等于 060 (30)的各个 ASCII 字符；060(03) 等于 ASCII 值 0 。任何十六进制数的值都不会比这个数小。如果某个键被按下，所产生的值小于 060 (30)，那么，8080 就执行 JC-NEXIN 指令。用这种方式，8080 忽略 ASCII 值小于 060 (30)的无效的十六进制数。8080 执行CPI 072 指令，判明某个键(0～9)的 ASCII 值是否存储在A寄存器里。如果A 寄存器的内容比 072 小，那么 8080 执行 CPI 072 指令，把进位标识位置逻辑 1 。因此，如果某个数字键 (0～9) 被按下，那么，8080 就执行 JC-NOLET (Not a LETter，非字母)

指令。8080 到达 NOLET 处时，它执行一条 ANI 指令，该指令再一次把A寄存器存储的 ASCII 值屏蔽。其结果是：D_7～D_4 位却都置 0，所以，这些数(0～9)的ASCII 值(260～271，B 0～B 9)被转换成 0000～1001 。这些数是十六进制数的对应的二进制数。8080 执行 NOLET 处的 ANI 指令后，它返回到 HEXBIN。

表 6-4　　接收一个 ASCII 字符后，A 的内容

键	A寄存器的内容
0	00110000
1	00110001
2	00110010
3	00110011
4	00110100
5	00110101
6	00110110
7	00110111
8	00111000
9	00111001
A	01000001
B	01000010
C	01000011
D	01000100
E	01000101
F	01000110

8080 执行 TTYIN，已经把校验位置 0

假设不输入某个数字(0～9)，而是按下一个字母键。如果这种情况出现，那么，8080 不执行 JC-NOLET 指令，因为这些字母的 ASCII 值都比 072 大。8080 执行 CPI 101 这条指令。因为 ASCII 字母的最小值是 101 (41)，所以，8080 必须查明，

是否按下了产生的值是大于 071 而小于 101 的某只键。如果某只键已经按下了，那么，8080 转移去执行 JC—HEXIN 指令，忽略这只键的代码。如果这个 ASCII 值是等于 101 或者更大，那么，8080 执行 CPI 107 这条指令。如果字母键(A～F)的某一只被按下，由于执行了这条指令，则进位被清零。如果产生的 ASCII 值大于 106 的任何一只键按下，那么，8080 将执行 JNC-HEXIN 指令，从而忽略这个值。我们使用四条 CPI 指令的目的是，为了把 ASCII 键的有效的十六进制值"套以括符"。也就是说，只有当 0～9 的某个数字键被按下，或者 A～F 的某个字符键被按下，8080 执行 HEXIN 这个子程序的末尾的那些指令。但是，请读者注意，在 NOLET 处，在四位高有效位被置 0 之前，应该把 011 (09)这个数加到某个字母(A～F)的 ASCII 值上。如果某个数字键(0～9) 被按下，则 8080 只屏蔽四位高有效位。为什么要把 011 (09)加到以 ASCII 为基础的有效的十六进制字母的对应值上呢？A 寄存器的 ASCII 值的四位最低有效位是 0001 。但是，我们必须把十六进制数 A 转换成二进制数 1010 。所以，为了把它们转换成适当的二进制数，必须把 1001 加到十六进制的字母所表示的 ASCII 值上。

表 6-5　ASCII 十六进制字母—二进制数的转换

字　母	ASCII 值	执行 ADI011 后	执行 ANI017 后
A	01000001	01001010	00001010
B	01000010	01001011	00001011
C	01000011	01001100	00001100
D	01000100	01001101	00001101
E	01000101	01001110	00001110
F	01000110	01001111	00001111

当 8080 从 HEXIN 这个子程序返回到 HEXBIN 子程序时，A 寄存器存储的内容是所键入的以 ASCII 的十六进制字符的四位二进制数。如果送入的十六进制字符是无效的，那么，在 0～9 或 A～F 的某个字符被输入以前，8080 不执行 HEXIN 子程序。因为两个十六进制字符组成一个 8 位数，所以必须把头四位二进制信息循环左移四位。请读者记住，被送入的第一个字符代表两个字符构成的十六进制数的一个高位有效字符。8080 把头四位二进制数循环左移四位后，把它保存在 C 寄存器里。这时，8080 第二次调入 HEXIN 子程序，从而能够输入第二个字符即最低有效位字符。当 8080 第二次从 HEXIN 子程序返回时，A 寄存器存储的内容是这个十六进制数的第二个字符的相应的二进制数。这两个值，一个被存储在 C 寄存器里，一个被存储在 A 寄存器里；当 8080 执行 ADDC 指令时，把它们合并起来。8080 把 ASCII 值的两个十六进制字符的二进制值存入 A 寄存器里，然后返回。

八位二进制数-ASCII 的十六进制数的转换

既然我们可以把电传打字机或 CRT 显示器输入的十六进制数字符转换成适当的 8 位二进制数，那么，我们当然可以编写二进制转换成十六进制的转换子程序。8080 在调用例 6-10 所示的例行程序之前，必须把需要转换成两个 ASCII 的十六进制字符的二进制数值存储在 A 寄存器里。转换操作发生时，把 ASCII 的两个十六进制数字符打印在电传打字机上或显示在 CRT 上。

8080 调入 BINHEX 子程序时，把A寄存器的内容，即二进制数保存在C寄存器里；它是将要被转换成 ASCII 的十六进制数的二进制数。然后，8080 执行ANI 360 这条指令，屏蔽四位最低有效位(LSB)。如果A寄存器的内容是：

11111110

那么，在 ANI 指令执行之后，A寄存器的内容将是：

11110000

再把四位高有效位循环右移，进入A寄存器的四位低有效位的位置。现在，A寄存器的内容可能处在 00000000 和 00001111 (00—0F)之间。子程序执行到这里时，必须把A寄存器的内容用电传打字机打印，或显示在 CRT 显示器上。因此，8080 调用 PHEX (打印十六进制字符)子程序。程序执行控制到达 PHEX 这个子程序时，把A寄存器的内容与 012 (0 A) 进行比较。如果A寄存器的内容是 00000000-00001001 (00～09)，那么，通过这一比较，使进位标识位置位，表示A寄存器的内容等于 0～9，因此，必须把 ASCII 数字字符 0～9 予以打印。作为按这一比较的结果，如果A寄存器的内容小于 00001010，那么，8080 执行 JC-NMBOK 这段指令。

例 6-10 二进制～ASCII 十六进制的转换子程序

```
/这个子程序用来把 A 寄存器的内容转换成
/两个 ASCII 的十六进制字符，
/并把它们用电传打字机打印，或者显示
/在 CRT 显示器上。
BINHEX, MOVAC/把这个二进制数保存在 C 寄存
        ANI     /器里。将这个数与 360 (F 0)相加，
        360     /从而把四位 MSB 保存在 A 寄存
        RRC     /器里。现在，把这四位最高有
        RRC     /效位循环右移四次，进入
```

```
        RRC     /四位最低有效位。
        RRC
        CALL    /然后打印这个十六进制字符，
        PHEX    /它与A寄存器的内容等效。
        0
        MOVAC/取原来的二进制数，装入
        ANI     /A寄存器里。现在，只保存
        017     /四位最低有效位。
PHEX,   CPI     /应该打印一个数或字母吗？
        012     /是 0000～1001 (0—9)某个数吗？
        JC      /是的，它应该是一个数。
        NMBOK
        0
        ADI     /不，它是一个字母，所以
        007     /加 007 (07)
NMBOK,  ADI     /现在，加 260，以便把
        260     /结果转换成一个 ASCII 字符。
        JMP     /然后，转移到打印机的子程序
        TTYOUT
        0
```

8080 的程序执行控制到达 NMBOK 时，把 260 (B 0)加到A寄存器的内容上，以便把A寄存器的二进制数转换成 ASCII 数字字符。当 8080 转移到 TTYOUT 子程序时，它将该字符打印。

如果A寄存器的内容是 00001010—00001111，那么 8080 执行 CPI 012(0 A)这条指令，把进位标识清零。所以，8080 不执行 JC-NMBOK 这段指令。可是，它把 007 (07) 加到A寄存器的内容上，其后，再加 260 (B 0)。8080 转移到 TTYOUT

子程序时，再打印这个 ASCII 字符。对于 1010～1111 这些数来说，必须把和数 267（B 7）加到A寄存器的内容上。如果A寄存器的内容是 00001010，那么

$$\begin{array}{r} 00001010 \\ +10110111 \\ \hline 11000001 \end{array}$$

这就是说，必须把 00001010 转换成 11000001，它是字母A的 ASCII 值。这两条连续的加操作指令，用来把和数 267（B 7）加到A寄存器的内容上，然后把 1010-1111 的这些二进制数转换成相应的 ASCII 的十六进制的字符（A到F）；这些字符的值是 301～306（C1 到C6）。

8080 执行 TTYOUT 子程序，打印这个字符以后，执行 TTYOUT 子程序末尾的 RET 指令。这条指令使 8080 返回到 BINHEX 子程序。MOVAC 指令把C寄存器的 8 位内容传送到A寄存器。然后 8080 执行 ANI 017 这条指令，把四位最高有效位屏蔽，即置 0。下一条要执行的指令是 CPI 012。即使 8080 没有调入这个 PHEX 子程序，这时，8080 也要执行这个子程序。CPI 012 是 PHEX 子程序的第一条指令，因此，下条要执行的指令是它。8080 这样编程序：让 BINHEX 子程序能够“进入”PHEX 子程序。有些程序设计员可能认为这种方法可以节省存储单元；其他一些程序设计员可能认为这是把这个子程序编写在开始的位置上的唯一方法。这个子程序与 TTYIN 子程序（例 3-20）类似，TTYIN 可以供输入和输出（回送）或只输出（TTYOUT）使用。读者可以观察得到，BINHEX 这个子程序可以被写成如例 6-11 那样的子程序。

读者可以看到，有许多 CALL 指令，后面紧跟着 RET 指令。只要避免这种情况，就可以使该子程序编写得更精炼。由

于该子程序编写得更精炼，当第一次检查这个子程序时，可能比较难于理解，这就是唯一的问题所在。

我们如果进一步把 ANI 指令删去，那么，可以使 BINHEX 子程序更精炼，如例 6-12 所示。

既然我们已经讨论了二进制，八进制和十六进制这三个数制的转换子程序，因此，我们将要讨论十进制和二进制这两个数制的转换子程序。

三位 ASCII 的十进制数-二进制数转换

如果将要执行算术运算，那么，我们可能要把许多十进制数装入 8080 微计算机里。读者已经看到，8080 所做的大多数算术操作都是用二进制数进行的。因此，我们必须找到一些方法，把十进制数转换成二进制数；8080 微型计算机计算出二进制结果后，把它转换成十进制记数法。一旦这个结果成了十进制数以后，就可以打印在电传打字机上，或者显示在 CRT 上。读者已经知道，十进制数可以表示为整数乘以 10 的幂。例如：

$$237=(2\times10^2)+(3\times10^1)+(7\times10^0)$$

即

$$237=(2\times100)+(3\times10)+(7\times1)$$

因为我们一次只能给 8080 微型计算机送入一位数，所以，在把 ASCII 的十进制转换成二进制的通用转换子程序中，必须有一个乘以 10 的乘法例行子程序或子程序。在这种乘法例行子程序中，当送入 93 时，必须首先把 9 与 10 相乘，然后再与

3 相加。ASCII 的十进制转换成二进制的通用转换子程序如例 6-13 所示。这个子程序所存在的问题与三位 ASCII 的八进制数转换成二进制数的子程序所存在的问题相同。这就是说，数可以被送入，但并没有适当地被转换相应的数。用例 6-3 的程序，可以把送 777 送入，但是，这个数将被转换成 1111111。用例 6-13 的子程序，只能送入 0～255 之间的一个数，并能加以适当地转换。如果要送入 256～999 之间的任何一个数，则不能转换成适当的二进制数。

例 6-11 较长的二进制数转换成 ASCII 的十六进制数的转换子程序

```
/这个子程序用来把A寄存器的内容
/转换成两位 ASCII 的十六
/进制字符，并且把它打印在电传打字机上
/或显示在 CRT 上。
BINHEX,MOVCA/把这个二进制数保存在C寄
        ANI     /存器里。这个数与 360 (F 0)
        360     /相与，四位最高有效位保
        RRC     /存在A寄存器里。现在，把
        RRC     /这四位最高有效位循环右移四次。
        RRC     /进入四位最低有效位。
        RRC
        CALL    /然后,打印这个十六进制字
        PHEX    /符，该字符与A寄存器的内
        0       /容等效。
        MOVAC/把原来的这个二进制数,取
        ANI     /到A寄存器。现在，只保存四位最低有效位。
        017
        CALL
        PHEX    /现在，打印这个十六进制字
```

```
        0       /的一位最低有效数字。
        RET     /然后，返回到该调入程序。
PHEX,   CPI     /一个数或一个字母应该被打
        012     /印吗？是 0000～1001(0～9)吗？
        JC      /是的。它是一个数。
        NMBOK
        0
        ADI     /不是。它是一个字母，
        007     /所以加 007(07)。
NMBOK,  ADI     /现在，为了把它转换为一个
        260     /ASCII 字符，加 260。
        CALL    /然后，调入 TTYOUT 子程序，
        TTYOUT  /该子程序把A寄存器的内容
        0       /打印在电传打字机上，或显
        RET     /示在 CRT 上。然后返回到BINHEX 子程序。
```

例 6-12 二进制数—ASCII 十六进制数的很精练的转 换程序

```
/这个子程序把A寄存器的内容转换成
/两个 ASCII 十六进制字符；把这两个字符
/打印在电传打字机上或者显示在 CRT 上。
BINHEX,MOVCA/把这个二进制数保存在C。
        RRC     /现在，把四位最高有效位
        RRC     /循环移入四位最低
        RRC     /有效位。
        RRC
        CALL    /然后打印这个十六进制字符：
        PHEX    /该字符等效于
        0       /A寄存器的内容。
        MOVAC/把原来的二进制数取到A。
```

```
PHEX,   ANI     /现在,只保存四位
        017     /最低有效位。
        CPI     /打印一个数或一个字母
        012     /吗?是一个 0000～1001(0～9)的
        JC      /数吗? 是的，它是一个数。
        NMBOK
        0
        ADI     /不是。它是一个字母,所以
        007     /加 007(07)。
NMBOK, ADI      /现在加 260，把它
        260     /转换成一个 ASCII 字符。
        JMP     /然后，转移到打印子程序。
       TTYOUT
        0
```

在 DECBIN 这个子程序的开始，把 8080 的两个寄存器预置初值。把 003(03) 装入D寄存器； 这个数是 DECBIN 子程序可以输入的 ASCII 十进制数的位数。C 寄存器用作暂时存储器，所以，应该把它置 000(00)。如果使用一个寄存器对， 而不使用C和D这两个寄存器，那么可以节省一个存储单元。8080 的程序执行控制到达 DECIT 时， 它调入 TTYIN 子程序。从 DECIT 往下到 ANI 指令， 这些指令使 8080 只输入有效的 ASCII 字符 (0～9)。在本章前面许多软件程序例子中， 也有这种分类功能。ANI 这条指令只用来把每一位有效数字的 BCD (二进制编码的十进制) 代码“滤” 出。现在，可以把后面哪个数存入A寄存器，应视所按下的键而定 (请参考表 6-6)。

例 6-13 ASCII 十进制数-二进制数的转换子程序

```
/这个子程序把三位 ASCII
/十进制数转换成等效的二进制数。
```

表 6-6　　ASCII 的十进制字符

字　　符	等 效 的 二 进 制
ASCII 1	00000001
ASCII 2	00000010
ASCII 3	00000011
ASCII 4	00000100
ASCII 5	00000101
ASCII 6	00000110
ASCII 7	00000111
ASCII 8	00001000
ASCII 9	00001001

```
DECBIN,MVID   /这个数是可以被送入的十进制
        003   /数字(000—255)的位数
        MVIC  /把C寄存器清零，因为要用它
        000   /作为暂时存储的寄存器。
DECIT,  CALL  /从键盘取一个字符。
        TTYIN
        0
        CPI   /这个键代码比 ASCII 值
        060   /0 小吗?
        JC    /是的。忽略该代码，
        DECIT /等待按下另一只键。
        0
        CPI   /它比 0～9 的各个数小吗?
        072
        JNC   /不。忽略它,并且等待
        DECIT /另一只键按下。
        0
        ANI   /现在，屏蔽掉四位最高有效位，
```

```
017     /留下四位最低有效位。
MOVBA /把这位十进制数保存在B里。
MOVAC/取C寄存器的暂存数。
RLC     /把这个数乘以 2，
RLC     /再乘以 2（总数＝×4）。
ADDC    加原来的数(总数＝×5)。
RLC     /乘以 2（总数＝×10），
ADDB    /加原来这个字符代码。
MOVCA/把这个新的二进制数保存在 C。
DCRD    /数位计数器减 1。
JNZ     /3 位数字都输入了吗？
DECIT
0
RET     /送入了三位数，
        /把二进制数存入C后，返回。
```

无论A寄存器的内容是什么，在 8080 执行 ANI 指令后，把它保存在B寄存器里。然后，把C寄存器的内容送到A寄存器，再执行若干条循环指令和加操作指令。这些指令的作用是什么呢？

假设必须把 93 这个数送入 8080 的一个具体程序。如果输入 9 这个数，读者希望把它用下面的格式存储在某个寄存器里：

00001001

但是，当把3这个数送入时，必须把 9 乘以 10，其结果等于 90，即其等效的二进制数是：

01011010

执行乘以 10 的乘法运算后，可以把 3 这个数的二进制数值加到 90 这个数的二进制数上，组成所希望的结果，即 $93(0101101_2)$。

乘以 10 的乘法运算，应该怎样进行呢？从我们的有关乘法运算的讨论中，读者应该记得，某个数乘以 2，同把这个数循环左移一次一样。为了把某个数除以 2，只要把这个数循环右移一次就行了。

如果A寄存器存储的内容是 00000001，那么，把它循环右移三次，正如把这个数乘以 2^3，即乘以 8 一样。如果把A寄存器的内容循环左移 5 次，则等于A寄存器的内容乘以 2^5，即乘以 32。幸好，使用循环移位指令和其他一些简单指令，可以很容易地实现A寄存器的内容乘以 9,31，或者甚至 62 的操作。

在 DECBIN 这个子程序中，必须把一个数乘以 10。读者已经知道

$$A \times 10 = (A \times 5) \times 2$$

所以，为了把C寄存器的内容乘以 10，8080 可以执行例 6-14 所列出的这些指令。

例 6-14 C寄存器的内容乘以 10

```
.
.
MOVAC/把这个数从C送到A。
RLC    /把这个数乘以2。
RLC    /再把它乘以2(总数＝×4)。
ADDC   /加原来的这个数(总数＝×5)。
RLC    /把该结果乘以2(总数＝×10)。
.
.
```

这个指令序列与子程序 DECBIN（例 6-13）所用的指令序列是一样的。C 寄存器的内容被乘以 10(原来送入的值)之后，把B寄存器的内容(最新输入的值)加到它上面。然后，把这个加操作的结果存回到寄存器C里，再使D寄存器的数字数位减

1。开始，D寄存器所存储的内容是003，因为3位十进制数字要被送入，并要转换成8位的二进制数。如果8080执行DCRD指令的结果不等于0，那么，它就执行JNZ-DECIT这段指令。这就是说三位ASCII十进制数尚未全部送入，因而计算机等待另一个有效的ASCII字符。如果三个ASCII的有效十进制键已经被按下了，那么，当8080从这个子程序返回时，由电传打字机或CRT的键盘送入的十进制数的等效的二进制数，存储在C寄存器里。

为了把93这个数送入，必须按下哪些键呢？因为8080微型计算机需要三次执行DECBIN这一子程序中的循环，所以，为了送入93这个数，我们应该按0，9和3这三只键如果只按9和3这两只键，那么，8080永远不会从DECBIN子程序返回，它会继续等待第三只键被按下。

三位ASCII八进制数转换成二进制的转换子程序如例6-3所示；可是这个程序不能用来输入比377大的数，并把它们转换成适当的二进制，这就是它所存在的难题之一。DECBIN子程序(例6-13)也存在着同样的问题。把256这个数送入以后，C寄存器的内容将是什么呢？C寄存器的内容将是000(00)。请读者记住，八位的二进制数只能表示十进制数0～255之间的一个数。所以，256会太大，不能只存储在C寄存器里。

我们经常需要处理比十进制数255大的数。这就是说ASCII的双倍或三倍精度的十进制转换为二进制的转换子程序，可以提供16位或24位的结果(0～65535或$0\sim1.67\times10^{7}$)，这正是人们所迫切希望的。读者可以回忆得起来例6-13的那段乘以10的指令如下：

MOVBA/把这个十进制数保存在B寄存器。

MOVAC/取C寄存器的暂时存储的数。

```
RLC     /把这个数乘以 2。
RLC     /再乘以 2(总数＝×4)。
ADDC    /加原来的数(总数＝×5)。
RLC     /乘以 2(总数＝×10)。
ADDB    /加原来的字符代码。
MOVCA/把这个新的二进制数保存在 C 寄存器。
```

但是，双精度循环移位方式与那些可以用来把十进制数转换成 16 进制数的乘法和除法子程序所采用的方法是类似的。例 6-15 所列出的这些指令正 是做双精度乘以 10 运算所需要的指令。根据该转换子程序的这段指令，读者能确定该转换所得到的二进制结果存储在什么地方吗?该结果的最高有效字节(MSBY)被存储在 D 寄存器里， 最低有效字节(LSBY)将被存储在 E 寄存器里。

我们不采用诸如例 6-15 所示出的那些指令， 而采用 8080 的一些 16 位数据值操作指令(例 6-16)，同样也能完 成 ASCH 的十进制数转换成 16 位的二进制数的任务。

例 6-15 双精度的 ASCII 十进制-二进制的转 换子 程序的一部分

```
/这是双精度的，十进制-二进制的转换
/子程序的一部分。
/它可以用来把 0～65535 之间的任何一个
/数转换成其等效的二进制数。
/仅就这个子程序的双精度的循环移位指令
/部分列出如下：
.
.
MOVBD/把最高有效字节暂时保存在 B 寄存器里。
MOVCE/把最低有效字节暂时保存在 C 寄存器里。
```

```
CALL   /将存储在寄存器对 D
ROT1   /的这个数乘以 2。
0
CALL   /现在把寄存器对 D 的
ROT 1  /内容再乘以 2(总数＝×4)。
0
MOVAC/现在，做双精度加法操作，
ADDE   /其后，
MOVEA/可以认为这个数已乘以 5。
MOVAB/取暂时结果的最高有效字节。
ADDC   /将这个数乘 4 后的内容的最高有效字节与 A 相加。
MOVDA/保存这个最高有效字节(MSBY)。
CALL   /把这个结果再乘以 2。
ROT 1  /现在，这个结果是原数的 10 倍。
0
•
•
ROT 1, MOVAE/取暂时结果的最低有效字节。
       RLC    /把它循环右移。
       MOVEA/接着，把这个最低有效字节存
       MOVAD/回 E。然后，取暂时结果的
       RLC    /最高有效字节。把进位标识
       MOVDA/位循环移位。再把该最高有效字节
       RET    /存回 D，然后返回。
```

双精度转换子程序的流程图如图 6-1 所示。

读者从例 6-16 可以看出，16 位数据操作指令的使用，可以使该子程序简单得多。乘以 2，与把这个数本身自加一次，其结果相等，这个子程序的一段指令就是根据这一事实而产生的。例如：

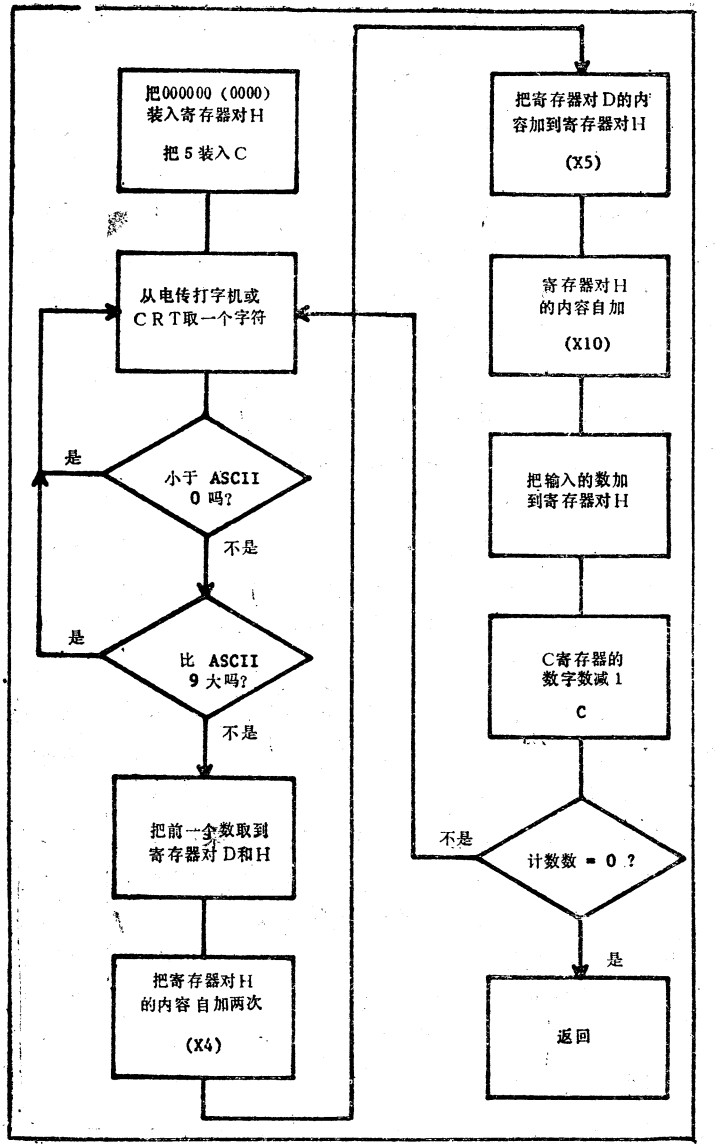


图 6-1　ASCII，16位十进制-二进制转换子程流程图

$$2\times5=5+5=10$$

$$2\times13=13+13=26$$

在例 6-16，寄存器对H所装入的内容是 000000(0000)。这个寄存器对用来存储暂时结果，和转换的最终结果。C 寄存器被作为位数计数器使用。因为一个 16 位的数可以代表 65,535，所以，可以送入一个 5 位的 ASCII 的十进制数并加以转换——只要这个五位的十进制数不大于 65,535。因此，送入 65,207 是有效的；送入 99,956 则是无效的。

寄存器对H和C寄存器被预置以后，为了保证该子程序只处理十进制数，8080 经常执行调用指令和比较指令。8080 接收了一个字符之后，从A寄存器的内容减去一个立即数据字节(060)。进行该操作的指令的功能与前面的例子中的 ANI 017 指令的功能相同。因为 TTYIN 子程序去掉 ASCII 码的校验位，所以，不管外部设备的校验位如何，SUI 060 这条指令可以用来处理任何电传打字机或 CRT 的操作。8080 执行减法操作之后，它把寄存器对H的内容压入堆栈，然后，再从堆栈弹出，送入D寄存器。寄存器对D和H现在存储的内容都是同一个 16 位数。此处不能用一条 XCHG 指令，取代这两条指令。8080 执行 DADH 指令，将寄存器对H的内容乘以 2。这条指令再执行一次，等于使寄存器对H的内容乘以 4。DADD 指令用来把寄存器对H的原来的内容（现在该内容被存储在寄存器对D里）加到寄存器对H的内容上，这样，现在寄存器对H的内容可以认为是乘了 5。8080 之所以执行 PUSHH，POPD 指令序列，是为了保存原来的这个数，以实现乘以 5 的操作。原来的这个 16 位数乘以 10 的操作由第三条 DADH 指令来完成。

从电传打字机或 CRT 显示器输入到A寄存器的字符，然后被存储在E寄存器里；D寄存器被置成 000。由于把D寄存

器置成 000，8080 只要执行一条 DADD 指令，就可以很方便地将一个新的十进制数与原来的数的 10 倍之积相加。乘以 10 的结果和加数现在被存储在寄存器对 H 里。然后，将 C 寄存器所存储的数的位数减 1。如果 C 寄存器的内容减 1，其结果不等于 0，那么，为了允许输入另一个十进制字符，8080 执行 JNZ-DECIN 1这段指令。如果 C 寄存器的内容减为 0 时，把与原来存储在寄存器对 H 的那个十进制数等效的 16 位的二进制数被存储在寄存器对 H 以后，8080 从该子程序返回。请读者注意，该子程序仍然不允许 76,128 和68,921 这两个数输入。因为它们太大，不能用一个 16 位的二进制数来代表。该子程序中，也有五个键操作动作，所以，57，必须以 00057 的形式送入。

例 6-16 ASCII 十进制数转换成 16 位的二进制数的转换子程序

```
/用这个 ASCII 的十进制——二进制
/的转换子程序，产生一个双精度的结
/果(16 位)。它包括 8080 的一些
/高级的指令，使软件的执行速度更快，
/程序的长度更短。
DECBIN, LXIH    /把寄存器对 H 置000 000 (0000)。
        000     /寄存器对 H 将用来存储十
        000     /进制数的等效的二进制数。
        MVIC    /用 C 寄存器作位数计数器。
        005
DECIN 1, CALL   /从电传打字机或 CRT 取一个
        TTYIN   /字符。打印这个字符，然后把
        0       /它存入 A 而返回 (D 7 已经等于 0)。
        CPI     /这个字符小于 ASCII 值 0 吗?
```

```
260
JC          /是的。则置之不理，
DECIN 1     /然后，取另一个字符。
0
CPI         /这个字符等于或大于 260。
272         /它小于并不等于 272 吗？
JNC         /不。它等于或大于 272 (BA)，
DECIN 1     /所以，忽略它。
0
SUI         /这个字符是一个有效的 ASCII 数。
060         /去掉 D 7，D 6，D 5 和 D 4 这四位。
PUSHH       /把寄存器对 H 中的数保存在
POPD        /堆栈。把这个数弹出堆栈，送
            /入寄存器对 D。
DADH        /把寄存器对 H 的内容自加，结
            /果存入 H(×2)。
DADH        /把寄存器对 H 的内容自加，结
            /果存入 H(×4)。
DADD        /把 D 和 E 的内容与 H 和 L 的内
            /容相加，结果存入 H 和 L (×5)。
DADH        /把 H 和 L 的内容自加，结果存
            /入 H 和 L (×10)。
MOVEA
MVID
000
DADD        /把输入项加到和数上。
DCRC        /位数计数器减 1。
JNZ
DECIN 1     /五位还没有取完，所以取另一个字符。
0
```

```
RET     /送入了五位，把其等效的二进
        /制数存入寄存器对 H，然后退
        /出该子程序
```

八位二进制数-ASCII十进制数转换

一旦十进制数已经被转换成二进制数，并进行适当的算术操作之后，人们希望用 ASCII 的十进制数把这一结果打印出来。如果这一结果用二进制、八进制或十六进制数来打印的话，那么，要把答案与原来所送入的那些十进制数联系起来，则是困难的。因此，我们要讨论把二进制数转换成十进制数的子程序。

如果读者把十进制数作为整数乘以 10 的幂来看待，那么，把二进制数转换成十进制数的过程，将可以大大地简化。为了把一个八位的二进制数转换成一个十进制数，可以通过连续的减法操作来实现。例如，为了确定 237 这个数包含几个 100，我们可以以 237 减去 100，一直到被减数是负值时为止。每当从被转换的这个数减去 100，并且其结果还不是负的时，那么，100 的个数就加 1。

```
  237
 -100
 ----
  137        百位数＝1
 -100
 ----
   37        百位数＝2
 -100
 ----
  -63
```

```
  +100
  ----
    37
```

为了决定这个数包含的 10 位数的数目，可以从这个数最后减 100 后的余数连续减去 10。就好象减 100 这个数一样，每当从这个余数减去 10，而且，该结果不是负值时，10 的个数增 1：

```
    37
  - 10
  ----
    27          十位数＝1
  - 10
  ----
    17          十位数＝2
  - 10
  ----
     7          十位数＝3
  - 10
  ----
  -  3
  + 10
  ----
     7
```

为了确定个位的数目，可以从十位减法的余数连续减去 1。但是，这个余数已表示出要转换的数中的个位的数目。为了对一个二进制数实行这些连减，我们必须采用 100（八进制 144，十六进制 64）的等效的二进制数，和 10（八进制 012，十六进制 0A）的等效的二进制数。这些数用于例 6-17 所示的八位二进制数转换成十六进制数的转换子程序。

例 6-17 八位二进制数转换成十进制数的转换子程序

```
/这个子程序用于二进制-十进制
/的转换。要调入这个子程序时，必须把要
/被转换的二进制数存入A寄存器里。
```

```
/8080 把等效的十进制数存入 E 寄存器（百位数）、
/D 寄存器（十位数）和 C 寄存器（个位数），然后返回。
BINDEC, LXID     /将 D 寄存器、
        000      /E 寄存器的内容，
        000      /置为 000。
        MVIC     /将 C 寄存器置于 000(00)
        000
SUB 100, SUI     /从 A 寄存器减去十进制数 100。
        144      /(八进制 144，十进制 100)。
        JC       /出现一个借位。
        ADD 100  /则把 100 加到 A 寄存器的内容上。
        0
        INRE     /百位计数器加 1。
        JMP      /再试减。
        SUB 100
        0
ADD 100 ADI      /试减太大，
        144      /所以，加 100（十进制）到 A 寄存器。
SUB 10, SUI      /现在，从 A 减去 10（十进制）。
        012
        JC       /借位已经出现，最好把十进制数
        UNITS    /10加到 A 的内容上。
        0
        INRD     /十位计数器加 1。
        JMP      /再试减
        SUB 10
        0
UNITS,  ADI/把十进制数 10 加到 A 的内容上。
        012
        MOVCA    /把个位的数目存入 C 寄存器里。
```

```
RET        /把答案存入E，D和C后，返
           /回。
```

当转换过程完成时，百位数的数目被存储在E寄存器里，十位数的数目被存储在D寄存器里，个位的数目被存储在C寄存器里。读者或许已经看到，调用这个子程序时，必须把要被转换的二进制数存储在A寄存器。

例6-17所示的这个子程序是比较简单的。在这个子程序的开头，C，D，E这三个寄存器都置于000，因为要用它们来存储转换的结果。然后，8080从A寄存器的内容减去立即数字节100（十进制）。如果A寄存器存储的内容是237_{10}，那么，现在，它存储的内容是137_{10}。没有借位出现，因此，8080不执行JC-ADD 100这段指令。8080而是使E寄存器的内容加1，所以E寄存器现在存储的内容是001。然后8080返回到SUB 100，再一次从A寄存器的内容减去十进制数100，因此，现在A寄存器的内容是十进制数37。没有出现借位，所以，现在E寄存器的内容增量到002；8080再返回到SUB 100。8080从十进制数37减去十进制数100时，因为从较小的数减去较大的数，所以会出现借位。因此，8080执行JC-ADD 100这段指令，在ADD 100这条指令上，8080把十进制数100加回到A寄存器的内容上。现在，A寄存器的内容是十进制数37。请读者注意，当8080从该子程序返回时，E寄存器可能存储的最大的数是002，因为8位的寄存器能够存储的最大的数是十进制数255。

最后，出现借位时，8080则执行JC-ADD 100这段指令；而且，把十进制数100（八进制144，十六进制64）加到A寄存器的内容上，以便产生最后一次"有效"的减操作的余数。然后，8080执行SUB 10这个循环，从A寄存器的内容减去十

进制数 10。每当从A寄存器的内容减去十进制数 10，而不产生借位状态时，则D寄存器的内容加 1 。最后，当出现一位借位时，8080 执行 JC-UNITS 这段指令；在 UNITS 指令上，把十进制数 10 加到A寄存器的内容上。因为这一加操作对于保留最后一次减去十进制 10 的余数产生影响，所以该结果代表个位数字的数目。然后，8080 把个位数字的数目存入 C 寄存器里。

既然八位二进制数-十进制数的转换操作已经发生了，那么，其转换的结果应该被存储在何处呢？用二进制数表示的十进制数的百位数的数目应该被存储在E寄存器里，表示十位数的数目应该被存储在D寄存器里，个位数的数目应该被存储在C寄存器里。由于这些数被存储在这些寄存器里，因此，下一个问题是编写一个子程序，将用来把该二进制数的等效的十进制数在电传打字机上打印，或者在 CRT 上显示。例 6-18 所列出的子程序可以用来达到上述目的。

我们应该首先打印哪个字符，是百位的数字还是个位的数字呢？应首先打印百位的数字，因为电传打字机是从左向右进行打印的。因为我们只能把 ASCII 字符在电传打字机上打印，或者在 CRT 上显示，所以，我们使用 BCDOUT 这个子程序来产生和打印适当的 ASCII 数字字符。

例 6-18 打印八位二进制—十进制转换的十进制结果的子程序

```
/这个子程序把存储在E、D、C这
/三个寄存器的 ASCII 字符
/在电传打字机上打印。TTYOUT 子程序的
/末尾的 RET 指令使程序执行控制返回
/到调入 DECPNT 子程序的程序点。
```

```
DECPNT,  MOVAE   /取百位的数，把它转换成
         CALL    /ASCII 字符，然后
         BCDOUT  /打印该 ASCII 字符。
         0
         MOVAD   /现在取十位的数，把它
         CALL    /转换成 ASCII 字符，然后打印
         BCDOUT  /百位的字符的右边的这个 ASCII
         0       /字符。
         MOVAC   /然后，取个位的数。
BCDOUT,  ADI     /为了把这个数转换成一个 ASCII
         260     /数，把 260 (B0)加到 A 的内容上。
         JMP     /然后在电传打字机上打
         TTYOUT  /印出 A 寄存器的内容。
         0
```

我们用来为C、D和E这三个寄存器的预置值的这个BINDEC子程序可以简化。开始把这些寄存器都装入000，然后根据BINDEC这个子程序所进行的减操作的结果，使它们增量。当八位二进制数全部被转换以后，8080可以调入DECPNT这个子程序，把该结果打印在电传打字机上或者显示在CRT上。8080在DECPNT处，调入BCDOUT子程序。这个子程序用来把260(B0)加到A寄存器的内容上后，再把这个ASCII数打印出来。如果C、D和E这三个寄存器的内容预置为260(B0)，则比较容易。这就是说，DECPNT这个子程序可以调入TTYOUT子程序，或者直接转移到TTYOUT子程序；因而可以取消BCDOUT子程序。例6-19所示的子程序反映了这些变化。

由于最初把260(B0)装入C、D和E这三个寄存器里，所以并不需要调用或者转移到BCDOUT子程序。8080可以调用

或者返回到 TTYOUT 子程序，因为C、D和E 这三个寄存器的内容已经成了 ASCII 数字字符的形式。当然，如果其它某一个例行程序或者某一个子程序需要使用 BCDOUT 子程序，那么，把 260 (B 0)装入这三个寄存器作为预置值，并没有什么优点。

十六位二进制数-ASCII 十进制数转换

BINDEC 这个子程序(例 6-17)只能用来把八位二进制数转换成十进制数，这是该子程序的局限性之一。因此，我们将要编写一个转换子程序，用来把一个 16 位 二进 制 数 转 换 成 ASCII 十进制数。这个子程序可以转换最大的数是 65,535，这就是说，转换所得到的结果的数位可以高达 5 位。这就告诉我们，将要使用五个存储单元来存储 5 位数字的结果。

例 6-19 简化的 BINDEC 和 DECPNT 子程序

```
/这是一个用来把二进制数转换成以 ASCII
/十进制数的较简单的转换子程序。
/我们必须把要转换的二进制数存入 A 寄存器
/后，才能调用这个子程序。用电传打字机以
/三位数的形式打印或用 CRT 显示器显示这个
/ASCII 十进制数的等效二进制数。
BINDEC,   LXID     /置 D=E=260 (ASCII 值 0 )
          260
          260
          MVIC     /置 C=260 (ASCII 值 0 )
          260
          •
```

```
          .
          ADI     /把十进制数 10
          012     /加到 A 寄存器的内容上。
          ADDC    /把 260 加到这个结果上。
          MOVCA   /把该结果保存在 C 寄存器里。
DECPNT,   MOVAE   /取最高有效数字，
          CALL    /然后打印它(它已经是
          TTYOUT  /ASCII 有效字符)。
          0
          MOVAD   /取十位数的数字，
          CALL    /然后打印它。
          TTYOUT
          0
          MOVAC   /最后，取个位的数，
          JMP     /然后打印它。TTYOUT 的末尾的
          TTYOUT  /RET 指令将使 8080 返回到
          0       /调用这个子程序的主程序。
```

为了使这个子程序尽可能简短，我们要使用几条 8080 16 位数据操作指令，特别是 DAD 类指令。采用前一个子程序，我们必须从正在被转换的二进制数减去十进制数 100 和 10 。那么，怎样用 DAD 指令来处理减法操作呢？读者应该记得，减法运算可以通过给被减数加上负数的方法来实现；例如：

$$\begin{array}{rr} 32 & 127 \\ +(-28) & +(-31) \\ \hline 4 & 96 \end{array}$$

因此，把一个 16 位的二进制数转换成 ASCII 十进制数，不是从这个 16 位的二进制数减去 10,000，而是执行一条 DAD 指令，把－10000 这个负数加到这个二进制数上。10,000 的

等效的负数是它的 2 的补码。当转换被完成时，把 5 位数字的结果存储在存储器里；当 8080 要调入这个子程序时，必须把要转换的这个 16 位的数存入寄存器对D里。

当 8080 进入 DPBDEC 这个子程序(例 6-20)时，把符号地址 UNIT 所指定的这个读/写存储单元的地址装入寄存器对 H。然后把这个数的位数装入 C 寄存器里；这个数的位数是 5 。8080 从SETTOO 开始执行指令，把数据字节 260(B 0)移到五个连续的存储单元，它们是，UNIT(个)，TEN(十)，HUN(百)，THOU(千)和 TTHOU(万)，这五个存储单元将用来累加个位、十位、百位、千位和万位的数目；这些数由二进制数转换而成。

例 6-20 16 位的二进制数-ASCII 十进制数转换子程序

```
/这是双精度二进制数-
/ASCII 十进制的转换
/子程序。当 8080 需要调用这个子程序时，它
/必须把要转换的这个二进制数存入寄存器对
/D 里。等效的五位十进制数将由电
/传打字机打印或由 CRT 显示。
DPBDEC,   LXIH    /把一个地址装入寄存器对 H，
          UNIT    /此地址所指出的读/写存储器
          0       /用来存贮十进制数的五位结果。
          MVIC    /C 寄存器作为数位计数器使用，
          005     /有 5 位数字。
SETTO 0,  MVIM    /通过把 ASCII 值 0 分别存入五个存储
          260     /单元，把这五个存储单元
          INXH    /都置零。使这个地址加 1 。
          DCRC    /位数计数器减 1 。
          JNZ     /计数器为零吗？不，保存另一个 260 (B 0)。
```

```
SETTO 0
 0
DCXH     /存储地址减1。
LXIB     /现在，把－10,000(二进制补码)
360      /装入寄存器对B里。
330
CALL     /现在，把－10,000加到寄存器对
DIGIT    /D的内容上，直到出现借位为
 0       /止
LXIB     /现在，把－1000(二进制补码)
030      /装入寄存器对B。
374
CALL     /然后，把－1000 加到寄存器
DIGIT    /对D的内容上，直到出
0        /现借位为止
LXIB     /现在，把－100(二进制补码)
234      /装入寄存器对B里。
377
CALL     /重复加操作过程。
DIGIT
0
LXIB     /最后，8080 把－10(二进制补码)
366      /加到寄存器对D的内容上
377
CALL
DIGIT
0
MOVAE    /取个位的数目。
ADI      /把 260(ASCII 值 0)
012      /加到这个数上。
```

```
        MOVMA   /把该结果存入存储器里。
        LXIH    /数制转换后，打印这个数。
        TTHOU   /把存储最高有效字节
        0       /的地址装入寄存器对H里。
        MVIC    /有五位数将要打印。
        005
PRINI,  MOVAM   /取这个 ASCII 字符。
        CALL    /电传打字机打印或
        TTYOUT  /CRT 显示这个字符。
        0
        DCXH    /存储地址指示器减 1。
        DCRC    /数字位数减 1。
        JNZ     /位数不等于 0，所以
        PRINT   /打印另一个数字。
        0
        RET     /当位数等于 0 时，8080 返回。
DIGIT,  PUSHH   /把存储器指示器的内容保存在
        XCHG    /堆栈里。把要转换的
        DADB    /数装入H、L。加二进制补码
        JNC     /测试值。如果出现借位，进位是“真”。
        ADDIT
        0
        XCHG    /无借位，正确地取回H和 L、
        POPH    /D和E的内容。从堆栈弹出该
        INRM    /存储地址。存储器单元数加
        JMP     /1。再试减。
        DIGIT   /只有当进位是“真”时，
        0       /8080 退出该循环。
ADDIT,  MOVAC   /现在，形成 2 的补码
        CMA     /的补码。
```

```
        MOVEA
        MOVAB
        CMA
        MOVDA
        INXD    /现在，D和E存储的内容是正
        DADD    /数。把它加到这个测试数上。
        XCHG
        POPH    /从堆栈弹出存储器指示器的内
        DCXH    /容。现在，把存储器指示器的
        RET     /内容减1。返回到主程序。
UNIT,   0       /这五个连续读/写存储单元，从
TEN,    0       /较低的地址开始，到较高
HUN,    0       /的地址，这些单元用来存储
THOU,   0       /与寄存器对D的二进制数
TTHOU   0       /等效的ASCII十进制
                /数。
```

由于要被转换的这个二进制数存储在寄存器对D里，所以，把要被减的这个数装入寄存器对B里。为了便于进行这个16位的减操作，8080使用DAD指令进行双精度加运算时，使用这个数的二进制补码。应用DIGIT子程序，可以节省一个存储单元，该单元用来累加数字位数的总和。这样，就把寄存器对H空出来，因而，可以使用DAD指令。下面的操作步骤是通过利用一条DADB指令，从这个16位的二进制数“减去”这个测试值；该测试值被存储在寄存器对B里，而正要转换的二进制数在寄存器对H里。连续地把测试值的二进制补码数加到要转换的二进制数上，直到出现进位才停止。这就表明，这个二进制数的值比这个测试值少。每一个不产生进位的正确的加操作，使存储器里累加的结果加1。这种操作过程如下例所示。

被转换的数＝37521

第一个测试值＝10,000

TTHOU 的预置值＝260

```
  3 7 5 2 1          TTHOU＝260
－1 0 0 0 0
  2 7 5 2 1          无进位  TTHOU＝261
－1 0 0 0 0
  1 7 5 2 1          无进位  TTHOU＝262
－1 0 0 0 0
    7 5 2 1          无进位  TTHOU＝263
－1 0 0 0 0
－  2 4 7 9          进位    TTHOU＝263
                             （不加 1）
```

在最后一次减操作中，出现进位，所以，我们不能把10,000的二进制补码加到被转换的这个数上。TTHOU这个存储单元存储的内容是ASCII字符3，它是正确的ASCII值。下一步操作是减1000，但是，首先，必须把余数由－2479恢复成7521。这可以通过把10,000加回到－2479上来实现。我们可以用DIGIT子程序来完成这一任务。

8080把10,000加到余数上之后，寄存器对H的存储地址被减1，从而指示到THOU。然后8080回到这个调入程序。然后DPBDEC子程序的其余指令继续对这个二进制数的千位、百位、和十位进行测试，这些测试是通过把每个数的二进制补码装入寄存器对B，然后，调入DIGIT子程序来完成的。每个测试的结果分别使相应的存储单元THOU，HUN和TEN的内容加1。我们并不需要对UNITS（个位）进行测试；因为它是最后一次减的余数。

8080 最后一次调用 DIGIT 这个子程序之后，它把个位数留在 E 寄存器，十进制数 10 与它相加，其结果被保存在 UNITS 存储单元。8080 完成转换操作之后，用电传打字机打印或显示五个 ASCII 十进制字符。为了方便读者起见，我们将用来打印其结果的 DPBDEC子程序的这段指令列于例 6-21。

例 6-21 DPBDEC 子程序（例 6-20）的打印指令

```
        MOVME /把这个数存入存储器里。
        LXIH  /转换之后，打印这个数。
        TTHOU /把存储最高有效字节
        0     /的地址装入寄存器对 H。
        MVIC  /将有五位数字被打印。
        005
PRINT,  MOVAM /取这个 ASCII 字符，
        CALL  /用电传打字机打印或
        TTYOUT/CRT 显示这个字符。
        0
        DCXH  /存储器地址指示器减 1。
        DCRC  /数字位数计数器减 1。
        JNZ   /数字位数计数器的值不等
        PRINT /于零，所以，打印另一个数
        0     /字。
        RET   /当位数等于 0，则返回。
```

因为必须首先打印最高有效数字（打印机是从左向右打印的），所以，应该把 TTHOU 这个符号地址的地址装入寄存器对H里。然后把要打印的数字的位数（005）装入C寄存器里。把最高有效位数字装入A寄存器里，然后调用 TTYOUT 子程序。这个字符被打印之后，存储器地址减 1，使它指示到下一位较低有效位数字。数字计数器也被减 1；并且，如果减 1,它

还不等于零时，8080 再执行 PRINT（打印）这个循环。最后，C 寄存器的内容减到零时，8080 从 DPBDEC 子程序返回到调用程序。寄存器对D的内容已经转换成 ASCII 十进制数字，其中各个数字存储在读/写存储单元里。其转换的结果将也要在电传打字机上打印，或者显示在 CRT 显示器上。

问题在于转换还是不转换

有时候，读者可能需要 8080 给事件记数。这些事件可以是外部发生的；即有关的输入/输出，例如，通过旋转门的人的数量，或者通过交叉路口的车辆的数目。事件也可以是内部发生的，例如：215(8 D）这个数倍乘，或 302（C 2）这个数在一个 4 K 的存储块中所出现的次数。这种计数甚至可以是在 24 小时内中断发生的次数。这些事件一旦计数后，读者或许将需要把它们用某种输出设备打印出来。

我们可以使用一条简单的 INR 或 INX 指令对这些事件进行计数；可以利用一个程序来打印这个数。这种程序可以是我们刚刚讨论过二进制数转换成 ASCII 十进制数的转换子程序。但是，这些子程序的长短、复杂性和灵活性，可能会严重地限制了它们的应用。例如，如果必须把三倍精度（三个字节）的二进制数转换成以 ASCII 十进制数，那么，会发生什么情况呢？会需要一个长长的子程序；特别是，如果把例 6-20 所示的这个子程序扩展成处理 24 位的数据字时，更是如此。

为了解决这个问题，我们将要编写出灵活的、容易使用的计数器程序。对这个程序的要求之一是，累加的数要容易地转换成 ASCII 值，以便输出给电传打字机或 CRT。例如，让我

们来研制一个微型计算机控制的车辆控制器。这个控制计数器可以设置在任何交叉路口；用它对通过该交叉路口的车辆进行计数。在每一周的周末，计算机打印出计数的车辆的数目。如果把这些数目用二进制、八进制或十六进制数表示，那么，无论什么人使用此系统整理车辆计数的结果都是困难的。因此，结果必须用十进制数打印出来。第一个要问的问题是：一个星期内，预计有多少车辆可以通过交叉路口呢？我们将假设通过交叉路口的车辆的数目不大于 65,535。如果我们把软件编写得适当，那么就容易给 8080 微型计算机重编程序，能够计算更大的数。

这个车辆计数问题可以细分成两部分。第一部分是具体计数的软件；第二部分是打印其结果。这两部分的任何一部分，可能需要某些数制转换。因为车辆的数量不能超过 65,535，所以，编写出一个程序，用某个寄存器对作为计数器使用，这是很容易的。寄存器对作为计数器使用的程序如例 6-22 所示。

我们在一周的开始，起动例 6-22 所示的这个程序。把 000 000(0000) 这个数装入寄存器对H里，然后，8080 调入 CAR(车辆)子程序。**在马路上埋有传感器。只有车辆通过该传感器时，8080才从CAR子程序返回。**这时，知道如何用软件来完成这一任务并不是重要的。但是，当 8080 从 CAR 子程序返回时，它执行 INXH 指令，把寄存器对 H 的内容加 1；然后，8080 执行 JMP—NXTCAR 这段指令，这时，8080 等待另一辆通过这个传感器。到了周末为止，寄存器对H存储了通过传感器的车辆的数量（二进制表示的数）。然后把这个二进制数转换成为 ASCII 十进制数，并且把它们打印出来。为了完成该转换操作，可以使用双精度（16 位）DPBDEC 子程序（例 6-20）。在调入 DPBDEC 子程序之前，应把寄存器对H的内容

传送到寄存器对D里。但是，读者已经看到，这个子程序需要79个存储单元，另加五个用来存储与十进制数等效的ASCII读/写存储单元。如果通过交叉路口的车辆的数量可能超过65,535，那么，将会发生什么情况呢？处理大于65,535的数是困难的，因为它需要三倍精度（24位）的转换子程序。为了解决这个问题，我们可以采用更灵活的程序。该程序如例6-23所示。

毫无疑问，这个程序比例6-22所示的开始的计数器程序复杂。但是，这个程序所产生的计数在打印前未必一定要把它从二进制转换成为ASCII十进制数。请读者注意，为了使例6-22和例6-23这两个子程序尽可能简单，所以，并没有任何在周末打印计数值的指令。计数器程序和转换/打印机程序是分开的，有区别的。

那么，车辆计数器程序是怎样工作的呢？前六条指令用来把260(B 0)装入四个连续读/写存储单元，从符号地址UNIT开始装入。因为这个计数器程序所产生的数是要用电传打字机打印或CRT显示的。这些连续读/写存储单元都被预置成字符0的ASCII值，被预置成260（B 0）的存储单元的数目由MVIC指令的立即数字节的值来决定。MVIC指令位于符号地址INIT和NXTO之间。这四个存储单元被预置初值之后，8080一开始执行符号地址COUNT上的指令。8080的程序执行控制到达COUNT时，它把UNIT的地址装入寄存器对H。然后，8080调入CAR子程序。车辆通过街道上的传感器之后，8080才从CAR子程序返回。这种情况正如前一个计数器程序（例6-22）一样。但是，现在，当8080从CAR子程序返回时，它执行一组完全不同的指令。

程序执行控制到达CNTUP时，寄存器对H所寻址的存储

器单元的内容加 1。寄存器对H现在指示到符号地址 UNIT。存储单元的内容加 1 后，8080 把它送到A寄存器里，然后与立即数据字节 272(BA) 进行比较。如果存储单元的内容小于 272(BA)，那么，8080 执行 JC—COUNT 这段指令。读者注意，272(BA) 比 ASCII 值 9 大 1。因此，只有 UNIT 存储单元的内容加 1 到 272(BA)，超过了 ASCII 值 9 (271，B 9) 时，8080 才不执行JC-COUNT这段程序。这种现象出现时，8080 执行 MVIM 260 指令。它把 ASCII 值 0 装入 UNIT 存储单元。

例 6-22 把寄存器对H用作为车辆计数器的程序

```
        .
        .
        LXIH    /把 000 000(0000) 装入寄存器对 H，
        000     /因为它将用来存储
        000     /通过车辆的数目。
NXTCAR, CALL    /等待车辆通过
        CAR     /街道上的传感器。
        0
        INXH    /一辆车通过传感器，车辆计
        JMP     /数器加 1，然后转移
        NXTCAR/并等待另一辆车通过。
        0
        .
        .
```

然后，寄存器对H所存储的这个存储地址加 1，因此，它现在对分配给 TEN 这个符号地址的存储单元寻址。然后，8080 执行 JMP—CNTUP 这段指令，并且，使 TEN (十位) 存储单元的内容加 1。然后对它进行检查，看它是否小于 272(BA)。8080 继续执行 CNTUP 这个循环，直到一个存储单元内容加

1 到小于 272(BA) 时，8080 才停止。然后，8080 执行 JC—COUNT 这段指令，在 COUNT 时，寄存器对H的存储地址被预置初值，指示到 UNITS 这个存储单元。

例 6-23 车辆计数程序

```
/这个程序所用的思想
/与前面那些子程序所用的相似。
INIT,     LXIH    /把分配给符号地址 UNIT
          UNIT    /的存储单元的地址装
          0       /入寄存器对H。
          MVIC    /这个数有四位数字。
          004     /最大的数=9999。
NXTO,     MVIM    /把 260 保存在寄存器对H
          260     /所寻址的存储单元。
          INXH    /把这个存储地址加 1 。
          DCRC    /数字位数减 1 。
          JNZ     /这个数等于 0 吗？不，再减 1 。
          NXTO    /使用这些指令把
          0       /这四个存储单元的初值置成 ASCII 0。
COUNT,    LXIH    /把分配给符号地址 UNIT 的存储器
          UNIT    /地址装入寄存器对H。
          0
NXTCAR ,  CALL    /等待车辆通过街上的传感器。
          CAR
          0
CNTUP,    INRM    /存储单元的内容加 1 。
          MOVAM   /此数加 1 后超过 ASCII 9 吗？
          CPI
          272     /ASCII 9=271
          JC      /不，然后转移到 COUNT，并且
```

```
        COUNT   /等待另一辆车，通过街上
        0       /的传感器。
        MVIM    /是，然后把这个存储
        260     /单元重新预置初值。
        INXH    /将这个存储地址加1。
        JMP
        CNTUP   /然后，把下一个存储单元加1。
        0
UNIT,   0       /这些是读/写存储器的四个连续
TEN,    0       /存储单元，用来存储以 ASCII 数
HUN,    0       /字符号表示的车辆数目，
THOU,   0
```

394 辆车已经通过交叉路口以后，存储单元的内容应该是什么呢？

存储单元	八进制内容	十六进制内容	等效的 ASCII 值
UNIT	264	B4	4
TEN	271	B9	9
HUN	263	B3	3
THOU	260	B0	0

如果 2153 辆车已经通过了街上的传感器，上述四个读/写存储单元的内容应该是什么呢？

存储单元	八进制内容	十六进制内容	等效的 ASCII 值
UNIT	263	B 3	3
TEN	265	B 5	5
HUN	261	B 1	1
THOU	262	B 2	2

读者可以看到，从 UNIT-THOU 这四个读/写存储单元可

以作为一个并行计数器使用，其中每个存储单元存储0～9之间的任何一个数（以等效的ASCII值形式）。使用这四个存储单元，8080可以对0000～9999之间的数计数。现在，读者应该已经知道这个子程序是怎样工作的。至于为什么这四个存储单元置初值260(B 0)，而不是000(00)的问题，毫无疑问，读者应该明白。

既然，车辆的数目以ASCII数字字符（0～9）的形式存储在存储单元，所以，打印这些数字的子程序是很简单的，这个程序如例6-24所示。NMBOUT子程序的第一条指令，把分配给符号地址THOU用的这个地址装入寄存器对H。然后，把计数数值装入C寄存器里，计数数值是4位，因为要打印四位ASCII十进制数。然后，8080把寄存器对H寻址的存储单元的内容装入A寄存器里，并且，调入TTYOUT子程序。因为这些数是以ASCII字符形式存储在读/写存储单元，所以，并不需要调用二进制-十进制(以ASCII值为基础)的转换子程序。电传打字机打印这个ASCII字符之后，8080返回到NMBOUT子程序的那条DCXH指令。这条指令使寄存器对H所存储的地址减1，现在它能够指示到HUN这个符号地址。C寄存器的计数数字也要被减1，如果该数不等于0，8080执行JNZ-NXTOUT这段指令，打印下一个ASCII字符。8080继续执行这个子程序，直到四个ASCII字符都打印完毕时为止。四个字符都被打印完毕以后，8080返回到调入NMBOUT子程序的主程序。

为什么要首先打印THOU这个存储单元所存储的ASCII数字呢？首先打印这个数字，是因为我们已经假设打印机是从左向右打印的。因此，必须首先把作为结果的最高有效数字打印。

例 6-24 打印读/写存储器存储的 ASCII 表示的车辆数

```
/这个子程序用来把 THOU，HUN，TEN，
/和 UNIT 四个存储单元的内容打印在
/电传打字机上，或显示在 CRT 上。
/因为存储在存贮器单元的内容已经是 ASCII 字符，
/所以该子程序是非常简单的。让我们首先看
/看 THOU 单元的内容是怎样被打印的。
NMBOUT,  LXIH     /把用来存储 4 位数字
         THOU     /的地址装入寄存器对H里。
         0
         MVIC     /把数字计数器置成 4 。
         004
NXTOUT,  MOVAM    /从存储器取一个 ASCII 字符。
         CALL     /把它打印在电传打字机上
         TTYOUT   /或显示在 CRT 上。
         0
         DCXH     /这个存储地址减 1 。
         DCRC     /数字计数器减 1 。
         JNZ      /计数数字是 0 吗?
         NXTOUT   /不是。打印另一个字符。
         0
         RET      /这些字符都被打印完了，所以
                  /返回。
```

如果我们把 8080 微型计算机移到交叉路口，每一个星期通过该交叉路口的传感器的车辆高达 99,999 辆，那么，例 6-23 和例 6-24 这两个程序应做出什么改变呢？改编这两个子程序容易吗？是的，十分容易。其中，仅需改变数字计数器；程序中要改变的指令只有两条。我们应该把例 6-23 的 INIT 程序段中的那条 MVIC 指令的立即数字节从 4 改成 5，以便对

0～99999 之间的数计数。INIT 这段程序将四个 读/写 存 储单元预置为 260(B 0)。NMBOUT 子程序（例 6-24）中的 MVIC 指令的数据字节也应该必须从 4 改变成 5 。而且，为了能打印万位的数字，NMBOUT(例 6-24)的那条 LXIH 指令的地址字节应该增 1，利用 ASCII 十进制计数器的优点 是，能很容易改变这个程序，从而使它的计数数字或大或 小。 例 6-23 和例 6-24 这两个程序只需要变动三处， 而且这三处变 动 是很简单的。

在计数器程序和子程序中应用 DAA 指令

上一节我们刚刚讨论过的那个计数器 程序（例 6-23），可以称之为 ASCII/BCD 计数器。读者已经看到了两个不同形式的计数器，一种是二进制计数器，另一种是 ASCII/BCD 计数器。是否还有另一种计数器，容易在 8080 机上实现吗？是的。使用了这条 DAA 指令的 BCD 计数器就是其一。在例 6-25 中，程序中应用了 DAA 指令，该程序可以从 0 到 9999 进行计数。

因为这条 DAA 指令用来处理已压缩的 字（即每个 8 位的字有两个 BCD 数），所以只有两个存储单元必须被预置初值。这两个存储单元可以用来保存 0～9999 的任何一个数。因为要使用 DAA 指令，所以这些 存 储 单 元 预 置 成 000， 而不 是 ASCII 值 0。预置初值的操作是由例 6-25 的 INIT 部分的指令来完成的。

例 6-25 使用 DAA 指令的 BCD 计数器(0～9999)

```
INIT,    LXIH     /给两个存储单元预置初始值，
         UNITEN   /因为每个存储单元可以保存
```

```
          0          /两位 BCD 数字。
          MVIC       /C寄存器作为存储单元计数器
          002        /用，使用两个存储单元。
          XRRA       /把A寄存器置 000(00)。
NXTO,     MOVMA      /把A寄存器的内容保存在存储
          INXH       /器里。此存储地址加 1，
          DCRC       /存贮器单元计数器减 1。
          JNZ
          NXTO       /还有一个存储单元要预置初始值。
          0
COUNT,    LXIH       /寄存器对H指示到 UNITEN。
          UNITEN
          0
NXTCAR,   CALL       /等待一辆车通过
          CAR        /街道上的传感器。
          0
          MOVAM      /一辆车通过了这个传感器。
          ADI        /所以，车辆计数数字增量。
          001        /加 1，与增量是相同的。
CNTUP,    DAA        /对 BCD 结果进行十进制调整。
          MOVMA      /把调整了的结果保存在存储器
          JNC        /里。没有进位，所以
          COUNT      /继续等待另一辆车。
          0
          INXH       /使这个存储地址加 1。
          MOVAM      /取下面两个最高有效字节。
          ACI        /把进位加到这两个最高有效字
          000        /节。
          JMP        /然后，8080 转移到CNTUP，对结
          CNTUP      /果进行十进制调整，并且检查
```

```
0          /结果的另一位进位。
```

8080 到达 COUNT 这个符号地址时，实际给车辆计数的过程开始了。首先，指定 UNITEN 这个符号地址。这个存储单元是由符号地址 UNITEN 来标记的。这个存储单元用来存储车辆的数目的个位数(D_3～D_0 位)和十位数(D_7～D_4 位)。然后，8080 调用 CAR 子程序；当车辆通过这个传感器时，8080 返回到 MOVAM 指令。然后，把 UNITEN 这个存储单元的内容送到A寄存器里，并且把 001 加到它上面。8080 执行 DAA 指令时，对该加操作的结果进行十进制调整；然后，把调整的结果存回到同一个存储单元。如果执行这条 DAA 指令的结果并没有产生进位（即，车辆数是 99，或更少），那么，8080 执行 JNC-COUNT 这段指令。如果这个数从 99 增至100，那么，进位标识位置位，A 寄存器的结果等于 000。所以，8080 不执行 JNC-COUNT 这段指令。因而，寄存器对H的存储地址加 1；把进位的内容加到下一个连续的存储单元。8080 执行 JMP-CNTUP 这段指令，把该加操作的结果进行十进制调整，并且存回存储器里。

现在，我们一定要缩写一个子程序，把压缩的 BCD 数分开，并且用电传打字机打印或用 CRT 显示其结果。

读者可以看出，这个例行程序比 NMBOUT 例行程序（例 6-24）长得多（28 比 15）；这个程序中 NMBOUT 与 ASCII/BCD 计数器一起使用。这个例行程序比较长，是因为要把计数数字的两位数字压缩成一个字。所以，为了把这些数分开，8080 必须执行数条循环移位指令。此外，必须把这些数从 BCD 码转换成 ASCII 的十进制字符。即使读者给事件计数高达 20 位数字(10^{20})，那么，使用 ASCII/BCD 计数器和打印机软件(例 6-23,6-14)与使用这条 DAA 指令的计数器软件(例 6-25,

6-26)相比，前者所需要的存储单元比后者的少。

例 6-26 用来展开两位数字的 BCD 数据字的子程序

```
/这个子程序用来打印压缩的 BCD 数字（它们
/存储在两个存储单元）。
NMBOUT,  MVIC    /使用两个存贮器单元，
         002     /所以，把计数数置成002(02)。
         LXIH    /H和L指示到这两个最高有效
         HUNTHO  /字节(即百位和千位数字)。
         0
P 2 BCD, MOVAM   /把这个数从存储器传送到A寄存
         RRC     /器。将千位数字循环右移，
         RRC     /进入A寄存器的
         RRC     /四个最低有效位位置。
         RRC
         CALL    /屏蔽其余四位。
         BCDOK   /然后利用 BCDOUT 子程序
         0       /打印该结果。
         MOVAM   /再从存储器取这个数，
         CALL    /并且打印四位最低有效位的这个数。
         BCDOK
         0
         DCXH    /存储地址减 1。
         DCRC    /存储单元计数器减 1。
         JNZ
         P 2 BCD
         0
         RET
BCDOK,   ANI     /屏蔽其余四位
         017     /(四位最低有效位除外)
         JMP
```

```
BCDOUT
0
```

我们还有一些问题没有讨论，问题之一是：如果所出现的计数数字大于9999，而所编写的程序只允许四位十进制数出现时，那么，会发生什么现象呢？在 ASCII/BCD 子程序(例 6-23)里，THOU 之后的下一个存储单元应该加 1。因为这个存储单元还没有预置初值 260(B 0)，所以，没有办法知道这个存储单元的具体数目是多少。当然，给 8080 编程序，用 260(B 0)为八个存储单元(而不是四个)都预置初始值，这是很容易做到的。这就是说，不可预测的结果出现之前，可以计数的数字与 99,999,999 一样大。在一星期之内，通过交叉路口的车辆达到这个数字，则不会有什么危险。NMBOUT 这个子程序(例6-24)也必须加以改变，才能打印一个八位数字的计数数字(而不是 4 位数字的计数数字)。读者已经看到，进行这些改变是很容易的。

删除无效零

电传打字机打印或 CRT 显示数字时，不打印或显示数字前面的零，即删除它，这是经常出现的现象。这就是说，当需要打印 00302 这个数时，不是把 00302 打印出来，而是只打印 302。在前面五位数字的车辆计数程序例（例 6-22 和打印机程序例 6-20），需要打印诸如 00302 或 01579 这样的数字。编写消除无效零的子程序，并不困难。

这种消零子程序，必须有某种形式的标识位，用来表示已经打印了第一个非零字符。这个非零字符把这个标识位置位

后，才能打印处于较低有效数字位上所出现的零。如果需要打印 0302 这样的数，那么，把 ASCII 0 都忽略，则是不适当的；因为都忽略的话，就会把 0302 这个数打印成 32。正如我们刚刚所讨论的那样，利用一位标识位，实现消除前面的零，打印子程序如例 6-27 所示。

例 6-27 这个子程序可以用来打印 ASCII/BCD 计数器子程序(例 6-23)所产生的车辆计数数字。8080 调入 NMBOUT 这个子程序时，把分配给THOU这个符号地址的地址装入寄存器对H里。这个地址就是存储千位数的存储单元的地址。然后，不管计数数字的前面的无效零有多少位，把计数数字的位数都装入C寄存器。接着，把 000 装入D寄存器，因为它可以作为标识位使用，以表示已经打印了一个非零字符。

8080 第一次通过NXTOUT时，把存储在 THOU(千位)这个存储单元的 ASCII 字符放到A寄存器里，然后把它与 0 字符(260,B 0)的 ASCII 值进行比较。8080 执行 JNZ-OKPRNT 这段指令，打印非零的字符。字符 0(260，B 0) 需要再进行检查。原先已经打印了一个非零字符吗？如果已经打印了一个，那么需要打印刚刚从存储器取出的这个字符 0 。如果还没有打印一个非零字符，则打印一个空格，而不打印 0 。在这两种情况下，都从存储器取出下一个字符，而且，要对它进行测试。D寄存器的内容作为“标识”使用。当碰到字符 0 时，把D寄存器的数值与 0 进行比较。如果D寄存器的内容等于 0 ，则不打印 0 这个字符，而打印“空格”。

当碰到 0 这个字符的 ASCII 值时，对这个“标识”进行测试。当碰到第一个非零的字符时，该标志（D寄存器的内容）被置成非零值。这种情况表明，后面的 0 字符需要打印；也就是说，在 302 这个数里，3 和 2 之间的这个 0 需要打印。8080

执行符号地址 OKPRNT 上的 MVID 377 这条指令，从而使该标识位置位。

例 6-27 不打印无效零的子程序

```
/它是前一个打印出子程序的改进程
/序。用它把一个四位数字的数
/前面的无效零消除。例如 0302，只打印成 302，
/不打印出0302。前面这个 0 不打印，只打印
/出一个空格。
NMBOUT,  LXIH     /把千位数字的地址
         THOU     /装入寄存器对H。
         0
         MVIC     /把计数数字的位数
         004      /装入C寄存器里。
         MVID     /用D作为
         000      /无效零标识寄存器。
NXTOUT,  MOVAM    /从存储器取一个字符。
         CPI      /这个字符是ASCII 字符 0 吗?
         260
         JNZ      /不是，然后打印这个字符。
         OKPRNT
         0
         MOVBA    /是的，把 ASCII 0 存入 B 寄存器。
         MOVAD    /ASCII零标志的状态是什么?
         ORAA     /根据A寄存器的内容把这些标
         MOVAB    /识位置位。把这个ASCII 0 送回
         JNZ      /到A寄存器里。这个字符等于
         OKPRNT   /0,但是已打印一个非零的字符，
         0        /所以，打印这个字符
         MVIA     /0(它不是无效0)。它是无效零，
```

```
            240         /所以应打印一个空格，而不打
            JMP         /印零。
            PRNTSP      /打印这个空格。
            0
OKPRNT,     MVID        /它不是ASCII字符0，
            377         /把D寄存器的内容置为377(FF)。
PRNTSP,     CALL        /打印这个字符。
            TTYOUT
            0
            DCXH        /存储器地址减1。
            DCRC        /数字计数器减1。
            JNZ         /该数等于零吗？不是。
            NXTOUT      /打印另一个字符。
            0
            RET         /四位都被打印完毕，
                        /因此，返回。
UNIT,       0           /用例6-23
TEN,        0           /这个程序，把ASCII
HUN,        0           /数字值装入这四个存储
THOU,       0           /单元。
```

小　　结

不用说，我们已经偏离了原来讨论数制转换程序的目标。我们讨论了 ASCII/BCD 计数器子程序，从而讨论并研究了计数器/转换问题的不同的解决办法。读者已经看到，用来把二进制数转换成 ASCII 十进制数的程序是很长的，人们常常感到难于理解。然而，就程序的可扩展性而言，这种程序又是

十分灵活的。采用略有不同的办法来解决同一个问题，这样ASCII/BCD计数器子程序可以用最少的程序指令来解决计数问题。

本章还讨论了消除无效零子程序，因为该子程序是一个有趣的程序。如果ASCII/BCD子程序的计数能力增加到99,999,999，把消零子程序合并到8080的车辆计数程序中去，则是很有用的。

我们在本章阐述了各种数制转换子程序，有了这些子程序，读者给8080微型计算机编制程序，输入八进制数十六进制数或十进制数，都很方便。这些数被"处理"之后，8080很容易把二进制结果转换成八进制、十六进制、或十进制的结果，然后用电传打字机打印，或者用CRT显示。

第七章　微型计算机的输入/输出(I/O)

这一章，我们将要探讨的领域是外部（或外围）设备与8080微型计算机连接的接口技术。在前面几章，我们已经假设把电传打字机或CRT显示器与8080微型计算机连接。为了读出由电传打字机键盘或纸带阅读机送入的一个8位数据的ASCII字符，8080等待标识位信号变成逻辑1电平，此后，它把这个ASCII字符输入到A寄存器里。电传打字机具体怎样与8080微型计算机连接呢？为了回答这个问题，正如在前面几章所采取的方式一样，我们将探讨软件的设计和开发问题；并且还要探讨某些硬件的设计和执行问题。硬件是一个广义的术语，用来描述电子器件，电阻，门电路，驱动器和锁存器等。这些元器件在电气上把外部设备和微型计算机的总线连接起来。**这就是说，本章的内容包括：常用的电子器件电路的具体方框图。**对一本阐述汇编语言程序设计的书来说，包括这些内容似乎是罕见的。但是，我们相信，汇编语言的程序设计员不仅要重视使用软件与外部设备通信，而且，他们也要重视学习如何具体地把外部设备与微型计算机连接起来。

我们将不探讨接口技术的某些细则；例如：用不同的方法给器件地址译码，怎样把要译码的器件地址与8080微型计算机所产生的同步脉冲信号（如：$\overline{\text{IN}}$和$\overline{\text{MEMW}}$）结合起来等。许多这些细则，在《数字电子器件实验指南》(Introductory Ex-

periments in Digital Electronics) 及《8080 A 微计算机程序设计和接口技术》(8080 A Microcomputer programming and Interfacing[1]) 这两本书中都讨论过。如果读者对门电路，译码器和三状态的集成电路器件还不熟悉的话，那么，这两本书向读者提供了如何运用这些器件详细的说明和用这些器件实现的实用电路。

微型计算机的输入/输出 (I/O) 为什么如此重要呢？它之所以重要是因为输入/输出 (I/O) 是 8080 和外部设备进行通信，或传送数据的唯一途径。如果 8080 不能与外部设备进行通信，对于 8080 来说，完成实际的任务，即使并不是不可能，但是，也是非常困难的。这就是说，如果 8080 不能利用电传打字机，CRT 显示器，纸带阅读机/穿孔机，模拟—数字转换器，固体电路继电器、键盘、软磁盘，或其它任何外部设备，那么，它的作用就很小。

我们已经讨论过的，能够用来和外部设备进行通信的唯一方法是：让 8080 执行 IN (输入) 和 OUT (输出) 这两条指令，借助于累加器输入/输出来实现和外部设备通信。这两条指令是双字节的指令；该指令的第二个字节是 8080 与它进行通信的外部设备的**地址**。例如，当 8080 执行 OUT 这条输出指令时，把 A 寄存器的 8 位内容送到一条 8 位的双向数据总线；把要接收这个数据的器件的地址送到地址总线 A_0～A_7，也可以送到 A_8～A_{15}。8080 执行这条 IN 指令，从外部设备输入 8 位数据，送到双向数据总线，然后选通，进入 A 寄存器。8080 在执行这条 IN 指令时，该指令的第二个字节所表示的外部设备的地址送到地址总线 A_0～A_7 或送到 A_8～A_{15}。当 8080 执行任何一条累加器输入/输出指令时，8080 微型计算机产生两个控制信号中的一个。外部设备电子器件用这两个控制信号使 8080 和外部设备

传送数据同步。这两个控制信号通常叫做$\overline{\text{IN}}$和 $\overline{\text{OUT}}$，或$\overline{\text{I/O READ}}$和$\overline{\text{I/O WRITE}}$。

用来把 8080 微型计算机和外部设备连接的另一种方法叫做存储器映象输入/输出。这种方法也可以用来在 8080 微型计算机和外部设备之间传送数据。当使用存储器映象输入/输出技术，把外部设备与 8080 微型计算机连接时，外部设备看起来象一个存储单元，或一组存储单元。这就是说，一个 16 位的地址与每个存储器映象输入/输出设备相关联。8080 微处理器集成电路和存储器之间的 8 位数据字的流动用什么控制信号来控制呢？我们用存储器读（$\overline{\text{MEMR}}$）和存储器写（$\overline{\text{MEMW}}$）这两个信号。这两个控制信号，也可以用来使存储器映象输入/输出设备和 8080 微型计算机之间的数据传送同步。这就是说，是从某个存储器映象输入/输出设备读出数据，还是从具体某个存储单元读出数据，8080 是不能区别的。对于 8080 来说，它们好象是同一回事。把数据是写到存储器映象输入/输出设备，还是读/写存储器的存储单元，8080 也不能区分。8080 微型计算机并不知道，它正在把数据写入的是何种外部设备。**用微型计算机程序或软件指令，把存储器映象输入/输出设备和真正的存储单元区别开来，这是程序设计员的任务。**

每当给 8080 编程序，来完成它和存储器之间的数据传送时，8080 微型计算机必须总是产生什么信号呢？8080 微型计算机必须产生一个 16 位的地址和适当的控制信号，用来决定双向数据总线上数据流的方向。在我们的许多程序设计实例中，或者用寄存器对 B、D，或者用寄存器对 H 来提供一个 16 位的地址，都用作为一个存储地址，供 8080 微型计算机和存储器之间传送数据用。也可以用这些寄存器对的内容提供一个地址，使 8080 能够与存储器映象输入/输出设备通信。对于存储器映

象输入/输出设备而言，为数据的传送提供所要求的同步操作，我们使用 $\overline{\text{MEMW}}$ 和 $\overline{\text{MEMR}}$ 这两个控制信号，而不用$\overline{\text{OUT}}$和$\overline{\text{IN}}$这两个控制信号。

为了 8080 能够把数据传送给存储器映象输入/输出设备，和从该种设备传送数据给 8080，我们可以使用什么指令呢？8080 的全部的存储器访问指令都可以使用。这就是说，有 33 条**不同的**指令可以用来控制存储器映象输入/输出设备。至于累加器输入/输出技术，能够利用的数据传送指令只有两条：IN 和 OUT。由于存储器映象输入/输出技术常常可以采用各种控制指令，所以，这种存储器映象技术似乎是富于魔力的选择对象。每一种技术的一些特征如表 7-1 和表 7-2 所示。

当读者考虑使用存储器映象输入/输出和累加器输入/输出时，有一些要点需要读者牢记。虽然存储器映象输入/输出技术使 8080 的指令系统的更多指令赋予活力，然而，有一些指令并不是特别有用的。例如：MOVMH 和 MOVML 这两条指令，只有我们需要把存储地址中一个字节传送给存储器映象输入/输出的设备时，它们才有用处。而且，MOVLM 和 MOVHM 这两条指令需要“改写”存储在 L 寄存器或 H 寄存器的地址部分。INRM 和 DCRM 这两条指令，只有把存储器映象输入/输出设备连接成同一个 16 位地址的输入端口和输出端口时，它们才是十分有用的。

存储器访问指令：除了 LDA，STA，LHLD 和 SHLD 这四条之外，其余各条指令都需要把一个 16 位的地址存储在寄存器对 B，D 或 H 里。这就是说，必须把存储器映象输入/输出设备的这个 16 位地址装入所要求的寄存器对。这就是说，就一个简单的输入/输出传送而言，一定要执行两条指令，这两条指令需要用四个存储单元存储；这两条指令，一条是 LXIH，

表 7-1　　累加器输入/输出的特征一览表

8080的指令	IN 和 OUT（两者都是双字节指令）。
控制信号	$\overline{\text{IN}}$ 和 $\overline{\text{OUT}}$。
数据传送	在累加器和输入/输出设备之间。
设备译码	在地址总线上（A_0～A_7，或 A_8～A_{15}），一个 8 位的设备地址，这个 8 位的地址包含在 IN 或 OUT 指令的第二个字节。
术语说明	I/O 方式叫做输入/输出方式。被译码的设备地址与同步脉冲信号（$\overline{\text{IN}}$ 或 $\overline{\text{OUT}}$）结合时，把它叫做设备选择脉冲。用这个信号控制外部设备。

表 7-2　　存储器映象输入/输出的特征一览表

8080的指令	MOVAM	MOVME	ANAM		
	MOVBM	MOVMH	XRAM		
	MOVCM	MOVML	ORAM		
	MOVDM	STAXB	CMPM		
	MOVEM	STAXD	INRM		
	MOVHM	LDAXB	DCRM		
	MOVLM	LDAXD	MVIM	⟨B_2⟩	
	MOVMA	ADDM	STA	⟨B_2⟩	⟨B_3⟩
	MOVMB	ADCM	LDA	⟨B_2⟩	⟨B_3⟩
	MOVMC	SUBM	SHLD	⟨B_2⟩	⟨B_3⟩
	MOVMD	SBBM	LHLD	⟨B_2⟩	⟨B_3⟩
控制信号	$\overline{\text{MEMR}}$ 和 $\overline{\text{MEMW}}$				
数据传送	在8080的任何一个通用寄存器和存储器映象输入/输出设备之间进行数据传送，也可以把立即数字节写入存储器。				
设备译码	地址总线（A_0～A_{15}）上的一个16位地址。可以把这个地址存入寄存器对 B，D或H；这个地址也可以出现在 LDA，STA，LHLD 或 SHLD 这些指令的第二和第三个字节。				
术　　语	存储器映象输入/输出技术叫做读和写，而不叫输入和输出。器件的译码地址，当与同步脉冲信号（$\overline{\text{MEMR}}$ 或 $\overline{\text{MEMW}}$）结合时，叫做地址选择脉冲，而不叫设备选择脉冲。				

另一条是 MOVBM。

存储器映象输入/输出技术为我们提供了多于 256 个输入设备和 256 个输出设备的寻址能力。但是，如果只有这个特征，我们没有几个人愿意使用存储器映象输入/输出技术。实际上，选择一种输入/输出技术，而抛弃另一种输入/输出技术，从软件的观点来看，这并没有突出的理由。我们一直喜欢采用累加器输入/输出；我们已经发现，采用这种输入/输出技术不能解决的问题没有几个。存储器映象输入/输出的用户们对这种技术或许也有同样的感觉。

重要的是：读者应该注意存储器映象输入/输出和累加器输入/输出这两种技术本身，它们相互之间并没有根本的不同。两者都需要一个设备地址和一个同步脉冲信号，用来为每个输入/输出设备提供设备选择控制信号。为了更深入地讨论和了解更多的例子，我们建议读者参考《数字电子器件的实验指导和 8080 A 微型计算机程序设计与接口技术》(Introductory Experiments in Digital Electronic and 8080 A Microcomputer programming and Interfacing) 这本书的第二册第 20 节和 21 节。

I/O 数据传送—总线控制

8080 执行一条输出指令时，A 寄存器的内容在数据总线上需要保留多长的时间呢？A 寄存器的一个 8 位的数据字在数据总线上大约需要停留 1 微秒的时间。1 微秒时间，对于电传打字机打印一个字符，或者对于我们读七段发光二极管显示器 (LED) 上的一个数，显然是不够用的。为此，当 8080 执行一

条 OUT（输出）指令时，数据总线上的八位内容由外部设备的接口电子器件锁存起来。用来锁存数据的器件，实际上就是一种存储器件。这种器件一般有一个控制输入端和若干个数据输入端。当该控制输入端由脉冲启动或时钟启动时，该器件的各输入端上的数据被选通进入该器件，并且保持它，一直到再给控制信号线输入脉冲信号或打入时钟时为止。一般说来，用 $\overline{\text{OUT}}$ 脉冲控制信号和 8 位的设备地址经控制门组合后所形成的信号控制这条控制线。同类器件也可以用于存储器映象输入/输出的外部设备的接口电子器件。晶体管—晶体管—逻辑（TTL）和金属—氧化物—半导体（MOS）集成电路器件都可以用来锁存数据，它们锁存数据的数量几乎都不受限制。

从存储器或存储器映象输入/输出设备读出数据时，或者当数据值从累加器输入/输出设备输入时，来自这些外部设备的数据实际被选通，进入数据总线，需要停留多长时间呢？这些设备的数据在数据总线上大约停留 1 微秒时间；只有当对这些外部设备适当地寻址时，来自这些设备的数据才能被置于数据总线上。这就是说，两个存储器单元或两个输入/输出设备不能同时送到数据总线上。当没有对具体某个存储器单元，或某个输入/输出设备进行选择时，绝不能把数据置于数据总线上。因此，我们用三状态集成电路器件把存储器映象输入设备或累加器输入设备与 8080 的双向数据总线连接起来。有许多可以用于微型计算机接口的三状态器件。如果读者对于三状态器件，锁存器不熟悉，或者对我们上面已经提到的其他一些概念不熟悉，或者如果读者要用一些集成电路做实验，«Introductory Experiments in digital Electronics and 8080 A Microcomputer Programming and interfacing»一书的第一、二册为读者提供了应用这些器件的大量的实验。

8080与简单的I/O设备

我们能够用来与8080微型计算机连接的最简单的输出设备，也许是带有发光二极管显示器（LED）的一个八位的输出端口。两个锁存器的8个输入端与8080的8位数据总线连接；这两个锁存器的输出端经过限流电阻与发光二极管显示器连接。这个接口电路方框图如图7-1所示。

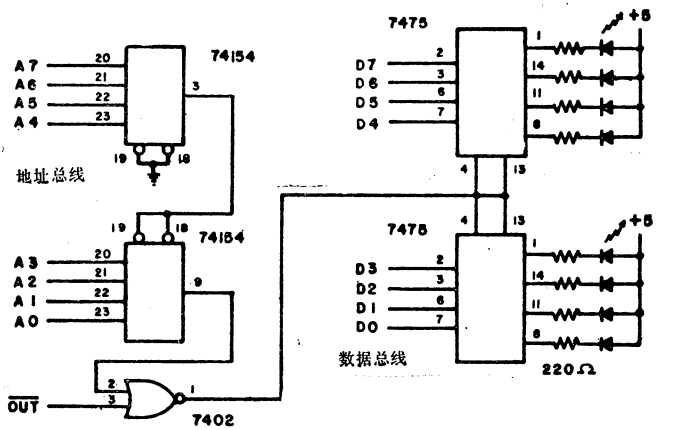


图 7-1　备有LED的8位输出端口

8080执行OUT 050这条指令时，两个SN 7475型锁存器只锁存数据总线的内容。8080执行这条输出指令（OUT 050）时，如果A寄存器存储的内容是377（FF），那么，八段的发光二极管显示器将全都被点亮。8080执行这条输出指令时，如果A寄存器存储的内容是000(00)，那么，八段的发光二极管显示器将都被熄灭。我们使8080用二进制数计数，然后把这

个数显示在与输出通道 050 连接的发光二极管显示器 (LED) 上，完成这些操作的程序是非常简单的；该程序如例 7-1 所示。

例 7-1 二进制计数与显示程序

```
/这个程序用二进制计数
/并且把该计数值显示在带有 LED 的输出端口上。
COUNT,  INRA    /A 寄存器的内容加 1。
        OUT     /把这个计数数字输出给两个锁存器
        050     /和 LED (050＝十六进制 28)。
        JMP     /返回；再使这个数加 1，
        COUNT /并显示其结果。
        0
```

如果 8080 执行这个程序，那么，8080 以多快的速度将这个计数数字增量呢？例 7-1 中的那个循环需要 25 个时钟周期，或者 12.5 微秒 (每个周期为 500 毫微秒)。这就是说，计数频率为 80 kHz。当然，这个计数速度太快，实际上，不论是谁，也看不过来。因此，为了使 8080 的计数速度降低，我们必须给这个程序添加一个延时循环程序。

因为A寄存器在例 7-2 这个程序的延期程序段中要使用，所以，用B寄存器来存储要显示的这个计数数字。因此，8080 执行 INRB 指令，把这个数加 1。然后，把这个数送到A寄存器里，并输出给输出通道 050 (十六进制数 28)，该通道带有发光二极管显示器。然后把 000000 (0000) 装入寄存器对D。因为寄存器对D的内容被减量到 377 377(FFFF)，然后再减量到 000000 (0000)，所以，下面的四条指令将被 8080 执行 65,535 次。当寄存器对D的内容最终等于 000000(0000) 时，8080 执行 JMP—COUNT 这段指令。8080 的程序执行控制到

达 COUNT 时，将 B 寄存器的数加 1，然后显示。我们可以改变这个程序，让 8080 能向下计数而不是向上计数吗？我们只要把 INRB 这条指令换成一条 DCRB 指令，就能给 8080 编程序，使它向下计数。在这两个程序（例 7-1 和 7-2)中，当8080开始计数时，我们并没有考虑 A 寄存器或 B 寄存器的内容是什么。我们只是需要把这个数加 1 或减 1。如果需要从某一具体值开始计数，那么，我们只要给 8080 编程序，把一个立即数字节或者装入 A 寄存器，或者装入 B 寄存器就行了。

我们可以给 8080 的接口电子器件增添某种附加电路，使 8080 能够把数据输入给 A 寄存器，这种电路如图 7-2 所示。为了把 8 个开关的逻辑 1 状态或逻辑 0 状态输入给 A 寄存器，8080 可以执行一条 IN 050 指令。只有 8080 执行 IN 050 这条指令时，才能把开关的状态选通，送入数据总线。在其他任何时候，输入通道好象是与数据总线断开的。那么，例 7-3 这个程序的功能是什么呢？

8080 执行这条 IN 050 指令时，例 7-3 示出的这个程序只用来把八个开关的状态输入到 A 寄存器里。然后 8080 执行一条 OUT 指令，把 A 寄存器的内容（八个开关状态）输入到这两个 SN 7475 型锁存器。接着，发光二极管显示器显示 A 寄存器的二进制数。因此，八个开关的状态实际上显示在输出端口上。因为 8080 继续输入开关的状态，把这些数据输出给发光二极管显示器，所以，只要观察发光二极管显示器的状态，就可以看到开关状态的变化。

例 7-2 较慢二进制计数与显示

```
/这个程序用来进行二进制计数，并显示结果，
/但是，已经给这个程序添加了延时，因此，
/实际上可以看到计数的变化。
```

```
COUNT,  INRB    /B里的这个数加1。
        MOVAB   /把这个数送到A。
        OUT     /然后把它输出给锁存器
        050     /和显示器（LED），(050＝十六进制28)。
        LXID    /把 000000＝十六进制 0000
        000     /装入寄存器对D。
        000
WAIT,   DCXD    /寄存器对D的内容加1。
        MOVAD   /把这个数的最高有效字节送到A。
        ORAE    /把这个数的最低有效字节与它相"或"。
        JNZ     /如果结果不等于0
        WAIT    /则转移，并且再执行
        0       /这条DCXD指令。
        JMP     /当寄存器对D的内容是
        COUNT   /000000(0000)时，增量
        0       /并且再显示这个数。
```

例 7-3 微型计算机的输入/输出程序

```
/这个程序用来把开关的逻辑电平
/输入到A寄存器里，然后把A寄存器
/的内容输出给两个锁存器和LED。
QUESS, IN       /把开关的逻辑电平输入到
       050      /A寄存器（050＝十六进制数28），
       OUT      /然后立即把同一个数据输出给
       050      /两个锁存器和LED。
       JMP      /返回，再执行同样的
       QUESS    /指令序列。
       0
```

当然，一旦这些开关与8080微型计算机连接时，也能够用它们来输入数据。例7-4所列出的这个程序用来等待与数据

总线的 D_7 位连接的那个开关置成逻辑 1。当这个开关置逻辑 1 时，其余开关的状态（逻辑电平）保存在 B 寄存器里。然后 8080 等待同一个开关的逻辑电平变成逻辑 0 状态。当这种情况出现时，把其余七个开关的逻辑电平加到 B 寄存器的内容上，然后把这个结果显示在发光二极管显示器（LED）的输出端口上。

例 7-4 输入、加和输出的简单程序

```
/这个程序用来从开关输入两个七位的二
/进制数，加这两个数，然后把加得的
/结果显示在 LED 的输出端口上。
INPUT, IN       /从开关输入这个数据。
       050      /050 =十六进制数 28。
       CPI      /最高有效位开关
       200      /处在逻辑 1 状态吗?
       JC       /不。然后保存这个数。
       INPUT
       0
       ANI      /它处在逻辑 1 状态，
       177      /保存 7 位最低有效位。177=十六进制 7 F。
       MOVBA    /把这 7 位数存入 B 寄存器里。
IN 1,  IN       /现在，等待开关回到逻辑 0 状态
       CPI      /后，才取最后这个数。
       200
       JNC      /这个开关还是处在逻辑 1 状态，
       IN 1     /所以继续等待。
       0
       ANI      /这个开关处在逻辑 0 状态，
       177      /所以把 7 位LSB 保存。
       ADDB     /加另 7 位数。
```

```
OUT     /然后，输出其结果。
050
HLT     /于是停机。
```

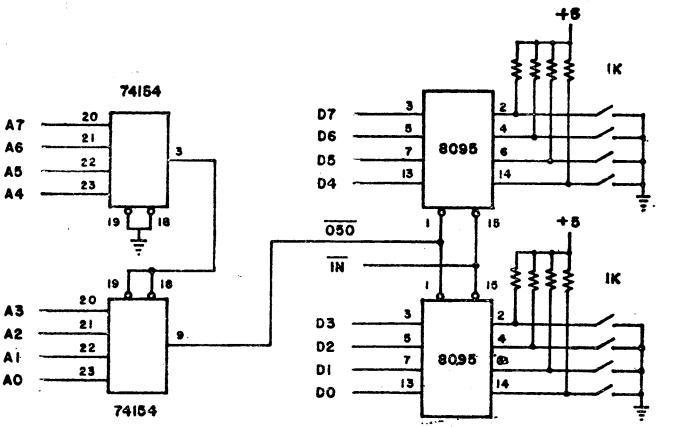


图 7-2　带有开关的 8 位输入接口

8080 与 键 盘

到现在为止，读者已经看到了相当数量的输入/输出程序设计实例。在前面许多程序实例中，我们使用电传打字机或CRT显示器作为外部设备。纸带阅读机/穿孔机、键盘、电传打字机的打印机、键盘和CRT的显示逻辑（打印机）都已经用于这些程序实例中去了。对于8080微型计算机的一些系统而言，无论电传打字机和CRT显示器这两个输入/输出设备的哪一个，其价格都可能太高。但是，作为把数据值送入到8080微型计算机的方式，它们还是需要的。这些数据值或许是数据

文件，或许是不同程序的始地址，或者甚至是程序步（8080指令系统的二进制、八进制或十六进制指令的操作码）方式输入，因此，我们将要讨论两种不同的，可以用来把键盘和8080微型计算机连接起来的方法，并讨论8080和键盘通信所需要的软件。

用硬件编码器的键盘的软件和硬件

有十五只键的键盘，它的硬件编码器，及其接口的电路方框图如图7-3所示。键盘的实际的键是在该图的右边。用两块**优先权编码器**集成电路（SN 7148）把每一个闭合的键编成二进制代码。这两块集成电路所产生的二进制代码与四个两输入端的与非门(SN 7400)结合起来，产生每一只键的唯一的代码。这四个与非门的输出端与DM 8095(SN 74365)集成电路连接。DM 8095是一种**三状态缓冲器**。这种集成电路用来控制8080的双向数据总线上的键盘的状态位和键的二进制代码，当8080执行一条IN 000指令时，把它们送入A寄存器。

读者注意，从这个编码器到DM 8095这个三状态器件有五条连接线。既然15只键可以用一个四位的二进制代码来表示，那么，为什么要有五条连线呢？五条线中的四条被二进制键代码使用；第五条线被用来表示：一只键已被按下。所以，当8080执行一条IN 000指令时，它把一个4位的键代码和键盘的状态输入。如果8080执行一条IN 000指令之后，读者可以检查A寄存器的内容，那么读者会看到表7-3所示的各个数值。

读者从表7-3应该可以意识到：当键盘的状态位是逻辑1

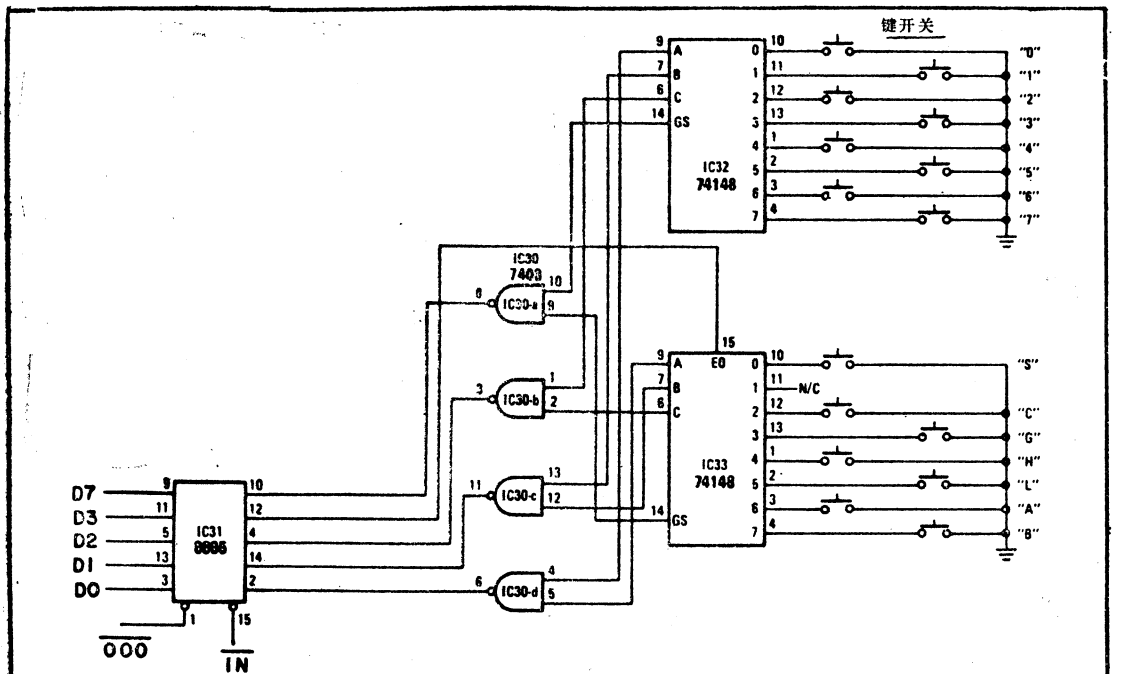


图 7-3　有十五只键的键盘接口与编码逻辑

时，代码位才代表一个有效的代码。此外，应该注意，这个键代码和状态位包含在一个 8 位的字之内。用电传打字机或 CRT 显示器的键盘输入 ASCII 字符时，并非属于这种情况。8080 微型计算机的软件必须对两种不同的端口进行存取，一个端口供存取状态位使用，另一个端口供存取 8 位的代码使用。

表 7-3　　从键盘输入后，A寄存器的内容

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
S	X	X	X	C_3	C_2	C_1	C_0

S＝键盘的状态

如果 S＝1，一只键被按下。

如果 S＝0，没有键被按下。

C_3＝当一只键被按下时，键代码的最高有效位是D_3

C_2＝当一只键被按下，下一只键代码位是 D_2

C_1＝当一只键被按下，下一只键代码位是 D_1

C_0＝当一只键被按下，这个键代码的最低有效位是 D_0

在这只键代码被输入之前，等待状态位变成逻辑 1；该子程序如例 7-5 所示。

例 7-5　键盘的简单输入程序

```
/该程序等待 D7 位成为逻辑
/1，然后输入 4 位
/的键代码。
KEYIN, IN       /输入这个 8 位的字，
        000     /该字包括状态位和数据位。
        ANI     /屏蔽状态位(D7)以外的各位，
        200     /(200＝十六进制数 80)。
        JZ      /D7 位是逻辑 0，所以
        KEYIN   /继续等待这一位变成逻辑 1。
        0
```

```
IN      /D7 位是逻辑 1，所以
000     /再输入这个 8 位字。
ANI     /然后屏蔽 4 位最高有效位。
017     /(017＝十六进制数 0 F)
RET     /把这只键代码存入A寄存器里，
        /然后返回。
```

读者可以看到，例 7-5 这个程序与第三章例 3-20 所给出的 TTYIN 和 TTYI 这两个子程序很相似。唯一的不同之点是：(1) 在A寄存器的内容中，只有一位不同，这一位表示键盘的状态；(2) 一个输入端口包含状态位和键代码。当然，例 7-6 所列出的这个程序也可以用来输入这个键代码。例 7-7 所列出的这个程序也可以用来读出键盘的状态位，但是，8080 不必执行第二条输入指令，或者暂时把这个键代码保存在另一个寄存器里。

例 7-6 另一个简单的键盘输入程序

```
/这个程序等待 D7 位变成逻辑 1。
/当它是逻辑 1 时，把被存储在 B 寄存器的
/键代码送到A寄存器里。
KEYIN, IN       /输入这个 8 位字，该字包
        000     /括状态位和数据位。
        MOVBA /把这个字存入 B 寄存器。
        ANI     /屏蔽各位(状态位 D7 除外)
        200     /200＝十六进制数 80。
        JZ      /D7 位等于逻辑 0，所以
        KEYIN  /继续等待这一位变成逻辑 1。
        0
        MOVAB/把 B 寄存器的这个 8 位字取到A
        ANI     /寄存器。屏蔽 4 位最高有效位。
        017     /(017＝十六进制数 0 F)。
```

```
        RET     /把这个键代码存入A寄存器里，
                /然后，返回。
```

例 7-7　最简单的键盘输入程序

```
/这个子程序等待 D7 位变成逻辑 1。
/它是通过把 200 (十六进制 80)加到输入端
/口的数据上的方式来检测的。当这个
/状态位是逻辑 1 时，进位是逻辑 1，
/四位键代码仍旧保留在A!
KEYIN,IN        /输入一个包含状态位和
        000     /数据位的一个八位字。
        ADI     /把 200(十六进制 80)加到这个字上。
        200     /如果进位是 0，则该标志位是 0!
        JNC     /进位是 0，所以该标识位是 0。
        KEYIN   /因此继续等待该标识位成为 1
        0
        ANI     /用 017(十六进制 0 F)屏蔽四位
        017     /高有效位，然后把这只键代码
        RET     /存入A寄存器而返回。
```

例 7-7 所示的这个子程序，我们使用了一个有趣的方法。我们把键盘的状态位和键代码输入以后，将 200（80）加到A寄存器的内容上；进位标识位被置位或者被清零，这取决于键盘的状态位的状态。如果状态位（D_7）是逻辑 0（没有键被按下），那么，由于加操作的结果，进位标识位将是逻辑 0。但是，如果状态位是逻辑 1，那么，加操作以后，进位标识位将是逻辑 1。因此，弄清 8080 微型计算机怎样检测键是否被按下，这是一件容易的事情。一旦某只键被按下，8080 则执行 ANI 这条指令，从而可屏蔽掉 D_7～D_4 位。

假设读者希望能够把键代码送入 8080 微型计算机，并把它

们保存在存储器里。为了完成这个任务，8080 微型计算机可以执行例 7-8 所给出的那个程序。当执行这个程序时，将会发生什么操作呢？第一条指令被用来把读/写存储器的一个 16 位地址装入堆栈指示器里。这条 LXIH 指令用来把这个读/写存储器地址装入寄存器对H里，该地址将是键代码所存储的读/写存储单元的地址。然后，8080 调入键盘输入子程序（KEYIN）。在 KEYIN 这个子程序中，8080 等待一只键按下。当某只键被按下时，8080 把已被按下的这只键的代码存入A寄存器里，然后从这个子程序返回。8080 然后把A寄存器的内容保存在存储器里，于是存储器地址加 1，然后 8080 执行 JMP-GETIT 这段指令。

例 7-8 把键代码输入 8080 微型计算机，并把它们保存在存储器里

```
START, LXISP     /把读/写存储器的一个 16 位地址
       100       /装入堆栈指示器里。
       100       /100100＝十六进制 4040。
       LXIH      /把用来存储键代码的
       200       /一个 16 位的读/写存储器地址
       100       /装入寄存器对H里，100200＝十
GETIT, CALL      /六进制 4080。把这个键代码存入
       KEYIN     /A寄存器，在返回之前，要等待
       0         /键盘的标识位成为逻辑 1。
       MOVMA     /把这个键代码保存在存储器里。
       INXH      /使读/写存储器地址加 1。
       JMP       /然后，取另一个键代码，并把
       GETIT     /它保存在读/写存储器里。
       0
KEYIN, IN        /输入一个 8 位字，该字
```

```
000         /包括状态位和数据位。
ADI         /把 200(十六进制 80)加到这个字上。
200         /如果进位是 0，标识位是 0!
JNC         /进位等于 0，所以标识位等于 0。
KEYIN       /因此，继续等待
0           /标识位变为逻辑 1。
ANI         /用 017(十六进制 0 F)屏蔽四位
017         /最高有效位，然后，把这个键代
RET         /码存入A寄存器，再返回。
```

读者可以用什么速度来按键盘上的键和释放键盘上的键呢？你可以用 50 或 100 毫秒(ms)来按键和释放键吗？在例 7-8 这个程序里，8080 只需要 40 微秒（每个周期为 500 毫微秒）来读出这只键，输入这个键代码，然后，从这个子程序退出，并且，把这个键代码保存在存储器里，将这个存贮器地址加 1，返回到 GETIT 子程序和再次调入 KEYIN 子程序。因为要用 50 毫秒或 100 毫秒按一次键，所以，在这个时间内，可把同一只键代码读出并且保存在 1000～2000 存储单元。正如我们在前面的章节中所描述的那样，UART（通用异步接收器发送器）连接到电传打字机上或者 CRT 上。当 ASCII 字符被从 UART 输入时，我们为什么没有碰到同样的问题呢？在 UART 和 8080 微型计算机之间的接口上，**有附加的接口逻辑电路，当 8080 输入一个键代码时，该电路用来给状态位，即状态标识清零**。电传打字机的键盘输入的典型子程序如例 7-9 所示。

例 7-9 电传打字机或 CRT 输入的典型子程序

```
TTYI,   IN      /输入 UART 的状态位。
        001
        ANI     /只保存接收器的状态位。
        001
```

```
JZ      /如果 A=001，一只键被按下。
TTYI    /如果 A=000，没有键被按下。
0       /如果没有键被按下，则继续等待。
IN      /一只键按下，所以输入
000     /这个字符的 ASCII 码，
RET     /把该 ASCII 码保存在 A 寄存器里，
        /然后返回。
```

当 IN 000 这条指令被执行来输入这个键代码时，用 8080 微型计算机产生的外部设备选择脉冲给标识位清零。这个标识位表示一只键已经被按下了。用这种方法，如果一只键实际上被按下了两次，那么这个键代码才输入两次。请读者**记住，这种清零功能是由硬件来完成的**。最后一点，8080 无论以什么速度调入 TTYI 子程序，只有目前被按下的这只键被释放了，另一只键被按下或同一只键再被按下时，8080 将才从该子程序退出。

可是，这个有 15 只键的键盘的硬接口的性能并不完善，所以，我们必须给 KEYIN 子程序增加一些软件指令，这样，不管一只键实际被按下去多久，8080 都能把这只键代码保存在存储器里。这个例子是**硬件/软件折衷**的另一个例子。在进行硬件和软件的折衷时，我们曾自我发问："增添一些硬件容易完成这个任务，还是软件指令容易完成这个任务呢？"在这个事例中，我们应该采用增添软件指令的方法来解决这个问题。部分地解决这个问题的程序如例 7-10 所示。

例 7-10 返回之前，等待要释放的键

```
/这个子程序用来等待 D7 位(状态位)
/变成逻辑 1。然后，把这个键代码保存在
/B 寄存器里。8080 再等待这只键释放
/并把这个键代码存入 A 寄存器
```

```
/之后，才从该子程序返回。
KEYIN,  IN      /输入这个 8 位字，该字包括
        000     /状态位和数据位。
        ADI     /把 200（十六进制 80）加到这个字上；
        200     /如果状态位是逻辑 1，则
        JNC     /产生进位。
        KEYIN   /如果标识位是逻辑 0，
        0       /则执行这条 JNC 指令。
        ANI     /屏蔽四位最高有效位。
        017     /(017＝十六进制 0 F)。
        MOVBA   /把这个 4 位的键代码保存在 B 寄
RELESE, IN      /存器里。再输入同一个字，
        000     /等待这只键被释放。
        ADI     /把 200（十六进制 80）加到这个字上，
        200     /可能产生进位，也可能不产生进
        JC      /位。当这只键被按下，状态
        RELESE  /(标识)位等于逻辑 1 时，8080 才
        0       /执行这条 JC 指令。
        MOVAB   /不再按这只键。
        RET     /把这只键代码存入
                /A 寄存器里，然后返回。
```

这个 KEYIN 改进型子程序（例 7-10)的前面三条指令被用来使 8080 等待键盘的状态位或标识位变成逻辑 1。如果这个标识位是逻辑 0，8080 执行这条 ADI 指令，来把进位标识位清零，从而，8080 能够执行 JNC-KEYIN 这段指令。当一个键被按下时，状态标识位被置逻辑 1；由于加法操作的结果，进位标志位将是逻辑 1。所以，8080 不执行 JNC-KEYIN 这段指令。然而，8080 把 A 寄存器的内容和 017(0 F)进行“与”操

作，并把其结果保存在B寄存器里。这只键仍然处在被按下的状态，所以，8080执行一些指令，检测这只键何时被释放。只有这只键被释放时，8080才从该子程序返回。因此，不论这只键按下去多久，只对它读一次。

我们在8080微型计算机的某个系统上执行这个程序时，每当一只键被按下一次，只有10～20个相同键代码，而不是1000～2000相同键代码被保存在存储器里。(8080执行例7-8这个程序时，则有1000～2000个相同键代码被保存在存储器里。)毫无疑问，例7-10这个程序比例7-8这个程序好。但是，每当一只键被按下时，为什么不能只把一个键代码保存在存储器里呢？这个问题是由于按键的弹跳作用所引起的。按键弹跳现象，是商用键盘上所使用的许多键所有的特征之一。当一只键被按下时，读者希望从这个键得到下面的输出波形：

遗憾的是，这些键并不是理想的开关，所以，开关的输出电平在逻辑1和逻辑0之间变动。这种现象如下一个波形所示：

读者可以看到，当键被按下，和被释放时，键在逻辑 1 和逻辑 0 这两个逻辑电平之间跳动。读者可以猜想得到，跳动的次数和跳动的持续时间是不固定的。我们假设，即使键跳动，它跳动的时间不大于 10 毫秒。这种假设是真实可信的。我们怎样修改 KEYIN 这个程序，使 8080 在键闭合后，只读一次，而不读上图所示的九次呢？最简单的解决办法是给 8080 编程序：让 8080 第一次检测到一个键被按下时，以及第一次检测到一个键被释放时，让 8080 执行一个短的延时子程序。

例 7-11 用延时子程序克服键闭合的抖动

```
/这个子程序给出了用来克服抖动作用
/所需要的软件指令，从而，使 8080 微型计算机
/识别一只键闭合，而不象是多只键闭合。
/请注意，当一只键被按下和一只
/键被释放时，8080
/调用克服抖动的软件。8080 把这
/个键代码存入 A 寄存器后，将返回。
KEYIN,  IN      /输入一个 8 位字，该字包含
        000     /状态位和数据位。
        ANI     /屏蔽各位，标识位(D7)除外。
        200     /200＝十六进制 80。
        JZ      /该标识位是逻辑 0，所以
        KEYIN   /等待它变成逻辑 1。
        0
        CALL    /然后，调用 10毫秒延时子程序，
        DELAY   /在该键代码输入之前，
        0       /克服键的抖动作用。
        IN      /10 毫秒之后，输入这个键代码。
        000
```

```
        ANI     /去掉各位，(数据位除外)。
        017     /017＝十六进制 0F。
        MOVBA   /把该键代码保存在B寄存器里。
RELESE, IN      /再输入状态位和数据位。
        000
        ANI     /只保存状态位，即标识位，
        200     /它是 D7(200＝十六进制数 80)。
        JNZ     /这只键还处于按下状态
        RELESE  /所以，继续等待，直到
        0       /它被释放时为止。
        CALL    /然后，调用 10 毫秒延时子程序，
        DELAY/  /从而消除键的抖动
        0
        MOVAB   /把该键代码从B寄存器
        RET     /传送到A寄存器，然后返回。
DELAY,  PUSHPSW/把 PSW 保存在堆栈里。
        PUSHD   /然后，把寄存器对D的内容保存
        LXID    /在堆栈。把 003 101(十六进制 0341)
        101     /装入寄存器对D。
        003
WAIT    DCXD    /这个数减 1。
        MOVAD   /把最高有效字节送到A。
        ORAE    /把 LSBY 与它相"或"。
        JNZ     /如果结果不等于 0，则
        WAIT    /转移到 DCXD 指令。
        0
        POPD    /结果等于 0，弹出D的内容，
        POPPSW  /然后弹出 PSW。
        RET/然后返回(不改变寄存器内容)。
```

例 7-11 所列出的这个子程序是很简单的。当它被调用时，8080 执行位于该子程序开始的三条指令所构成的循环，直到一只键被按下后才停止。当键盘的状态位（标志）是逻辑 1 时，A 寄存器的内容的 D_7 位也是逻辑 1，所以，8080 不再执行这条 JZ 指令。相反，8080 调入 DELAY 这个子程序。DELAY 子程序把 PSW（处理机状态字）和寄存器对 D 的内容保存在堆栈上；把一个计数数装入寄存器对 D；然后执行 DELAY 子程序的循环指令；直到寄存器对 D 的这个计数数被减量到零时为止。当这个数等于 0 时，寄存器对 D 的内容和 PSW 被从堆栈弹出，然后 8080 从 DELAY 子程序返回。假设 8080 的指令周期为 500 毫微秒，那么，完成这个执行过程所需要的时间是 10 毫秒。

当 8080 从 DELAY 这个延时子程序返回时，把这个键代码输入到 A 寄存器里，A 寄存器的四位最高有效位都被置零，然后把一个四位的键代码保存在 B 寄存器里。8080 接着开始执行在 RELESE 符号地址上的指令。程序执行控制到达 RELESE 时，8080 等待这个被按下的键释放。当这个键释放以后，A 寄存器的 D_7 位不再是逻辑 1；8080 再一次调入 DELAY 子程序，所以，这个键被释放之后 10 毫秒，8080 才执行 MOVAB 指令。这个键代码被传送到 A 寄存器之后，8080 从 DELAY 延时子程序返回。如果使用这个子程序，不论一只键按下去多久，只要这只键的跳动时间不大于 10 毫秒，那么，就只能有一只键代码送入 8080 微型计算机。

假设读者要把键盘和 8080 微型计算机连接，但是，该键盘上的键的跳动时间高达 20 毫秒。读者还能使用例 7-11 所列出的那个子程序吗？是的，正是这个子程序可以使用。但是，读者应该改变这个 DELAY 子程序所产生的延期时间。怎么

办呢？我们只要改变 DELAY 子程序的 LXID 这条指令 的 两个数据字节，就可以产生 20 毫秒的延期时间。

如果需要的话，读者把 DELAY 子程序合并到 KEYIN子程序里去，就能够节省几个存储单元。这个子程序 如 例 7-12 所示。如果把这两个子程序合并起来，可以节省 CALL 和RET 这两条指令。请读者注意，这条 MOVAB指令仍然靠近 KEY-IN 子程序的末尾。这就是说，如果该程序的任何其他 一 段指令调用 DELAY 子程序；恰好是在 8080 从 DELAY 子程序返回之前，总是把 B 寄存器的内容传送到 A 寄存器里。

既然我们已编制了 KEYIN 子程序，所以，我们可以用键盘来输入数值数据，指令操作码，或者甚至是 8080 所 转 移的地址。这个键盘接口的局限之一是它需要若干块集成电路。如果用这个键盘接口来组装 2000 台 8080 微型计算机系统，那么寻找其它的方法，把键盘和 8080 微型计算机连接起来， 则 更为有益。是否可以把 SN 74148 优先权编码器集成 电 路 和 SN 7400“与非”门集成电路从该接口设计中去掉，这是值得考虑的十分重要的一点。在本章的下一节，我们将要讨论多路转换的(即扫描的)键盘。这种键盘的接口与我们前面所用的许多键盘接口设计比较，则更为简单。但是，多路转换的键盘所需要的软件则更为复杂。

例 7-12 缩短键输入时间和消除键抖动的子程序

```
/这个子程序提供了消除键抖动所需要
/的指令，这样，一只
/键的闭合时，8080 微型计算机不会看作是多只键的闭合。
/注意，当一只键被按下以及
/释放时，8080 调入消除抖动子程序。
/8080 将把键代码存入 A 寄存器，然后返回。
```

```
KEYIN,  IN      /输入一个 8 位的字，
        000     /它包含状态位和数据位。
        ANI     /屏蔽各位（标识位 D7 除外）。
        200     /200＝十六进制 80。
        JZ      /该标识位是逻辑 0，
        KEYIN   /所以等待它成为逻辑 1。
        0
        CALL    /然后，调入 10 毫秒延时子程序，
        DELAY   /在这个键代码被输入之前，
        0       /消除该键的抖动。
        IN      /10 毫秒之后再输入这个键代码。
        000
        ANI     /去掉各位（数据位除外）。
        017     /017＝十六进制 0 F。
        MOVBA   /把这个键代码保存在 B 寄存器里。
RELESE, IN/     /再一次输入状态位和数据位。
        000
        ANI     /只保存状态位即标识位，
        200     /该位是 D7(200＝十六进制 80)。
        JNZ     /该键仍旧是被按下的，
        RELESE  /所以，继续等待，直到
        0       /它被释放为止。
DELAY,  PUSHD   /把寄存器对D的内容保存在堆
        LXID    /栈里。把 003 001 (十六进制 0341)
        101     /装入寄存器对D里。
        003
WAIT,   DCXD    /这个计数数减 1。
        MOVAD   /把这个最高有效字节送到 A。
        ORAE    /把这个最低有效字节与它相“或”。
        JNZ     /如果其结果不等于 0，
```

```
WAIT    /则转移去执行 DCXD 指令。
0
POPD    /其结果等于 0，弹出D的内容。
MOVAB   /把这个 4 位的键代码从 B 寄存
RET     /器传送到A寄存器，然后返回。
```

软件驱动的多路转换（扫描的）键盘

在前面那些 KEYIN 子程序中，用接口电子器件为键盘上每只键产生不同的键代码。因为我们现在的设计从接口上去掉了优先权编码器集成电路，所以，实际上，单值的键代码的产生是由软件来完成的。这是硬件/软件折衷的另一个例子。我们的讨论将从有 16 只键的键盘开始。该键盘的结构是 4×4 的矩阵，如图 7-4 所示。

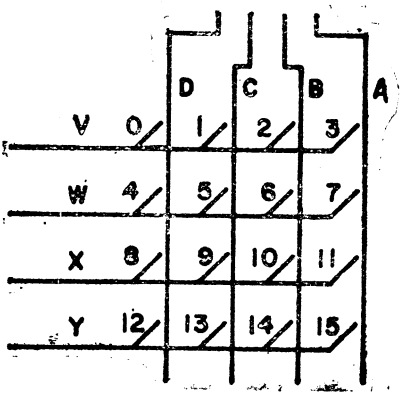


图 7-4　4 × 4 矩阵的线路框图

请读者注意，这个键盘与我们前面的程序例中所采用的键盘是完全不同的。那么，这种新的键盘是怎样操作的呢？假设A线保持地电平(逻辑0)，B，C，和D这三条线保持在+5伏电平。然后，8080可以输入和测试V、W、X和Y这四条线上的逻辑电平。如果这个键盘上的这四条线(V、W、X和Y)的输出电平都是逻辑1，那么，既不是3，7，11某一只键被按下，也不是键15被按下了。如果这种情况出现，A线的电平变为逻辑1，那么，8080则把线B的电平置于逻辑0。现在，8080对键盘上的四条输出线进行测试，这样，它就能决定键2，6，10或14中是否有一只被按下。8080连续地把键盘上的C线，然后D线置于逻辑0电平，它就能决定键1、5、9或13是否有一只已被按下；或者决定键0、4、8或12是否有一只键已被按下。如果16只键中一只也没有按下，并不考虑输入是逻辑0状态，所有的输出将都是逻辑1。只有8080检测到输出是逻辑0电平时，才有一只键按下。

读者可以看到，为了确定是否有一只键被按下，再没有个别的状态位即标识位，可以供监控用了。只要改变键盘的四个输入端的逻辑电平，并监控键盘的逻辑电平的输出，8080微型计算机就能够确定是否有一个键按下。用8080微型计算机控制键盘的接口电子器件的方框图如图7-5所示。

8080怎样具体确定哪只键已经被按下，然后产生与这只键相应的一个二进制代码呢？开始，8080必须把XXXX 1110(X＝不考虑)装入A寄存器里，然后，把这个数输出给锁存器SN 7475。该锁存器用来驱动键盘的A、B、C和D这四条输入线。逻辑0电平送给键盘的A线，或者送给3、7、11和15这四个键。8080经过DM 8095集成电路，输入键盘的V、W、X和Y各条线的逻辑电平，这样，它就能够确定键3、7、11或

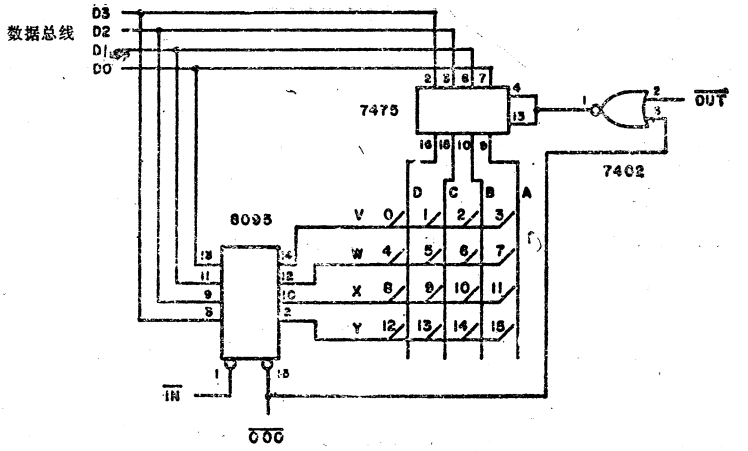


图 7-5　4 × 4 扫描型键盘的电子接口

者 15 是否按下。如果这四只键的某一只被按下，那么，输入端口 000 与 8080 微型计算机的连接一个输入端将是逻辑 0 电平。如果键 3、7、11 这三只键没有一只被按下，键 15 也没有被按下，那么，A 寄存器的四位最低有效位将是逻辑 1 。因为输入端口 000 没有连线和数据总线的 D_4～D_7 连接，所以，当 8080 执行 IN 000 指令时，A 寄存器的 D_4～D_7 将是逻辑 1。但是，8080 还必须检查四位最低有效位，才能确定某只键是否被按下。我们刚刚讨论的这些操作，是用例 7-13 所列出的程序来完成的。

例 7-13　4× 4 矩阵键盘的扫描子程序

```
/这个子程序用来对矩阵结构的
/键盘扫描，该结构为四排，
/每排有四只键，即 4×4
/的矩阵。
```

```
KEYSCN, MVID    /把可能被检测的第一只
        003     /键的代码装入D寄存器里。
        MVIB    /把用来一次启动一排键
        376     /的这个字装入B寄存器里。
NXTGRP, MOVAB   /取这个测试字，然后
        OUT     /把它输出给键盘。
        000
        RLC     /把该测试字循环左移一位，
        MOVBA   /然后，把它保存在B寄存器里。
        IN      /从这四排键输入数据。
        000
        ANI     /只保存四位最低有效位，
        017     /它包含排测试数据。
        CPI     /把 017 (0 F)与这个输入
        017     /字比较，确定这一排是否有一只键被
        JNZ     /按下。这一排的一只键被按下了，
        NXTKEY  /所以确定是哪只键被按下了。
        0
        DCRD    /在被测试的这一排没有键被按
        MOVAD   /下，所以键代码减 1，然后
        CPI     /判断四排键是否都已经测试
        377     /完毕。377，十六进制 FF。
        JNZ     /四排键还没有全部测试完，
        NXTGRP  /所以，测试另一排。
        0
        JMP     /四排键都被检测完毕，
        KEYSCN  /没有键被按下，所以，继续
        0       /扫描。
NXTKEY, RRC     /把这排测试数据循环右移，进入进位。
        RNC     /当进位等于0时返回。
```

```
PUSHPSW/否则，把 PSW 保存在堆栈上，
MOVAD  /然后使D寄存器的键代码加 4 。
ADI
004
MOVDA  /把这个新键的代码保存在 D 里。
POPPSW /把 PSW 弹出堆栈，
JMP    /然后，再试，进位是否
NXTKEY /是0。
0
```

在这个子程序(例 7-13)的开始，我们把 003(03)装入D寄存器里，把 376(FE)装入B寄存器里。用D寄存器保存第一只键的键代码；当这只键被按下时，8080 可以检测到它。B寄存器存储这个字(测试字)，这个字将输出给键盘。程序执行控制到达 NXTGRP 时，把B寄存器的内容输出给键盘。现在，键盘的哪些输入线的电平是逻辑 1 或逻辑 0 呢？A寄存器的内容是 11111110，所以，只有键盘的A线的逻辑电平是 0。B、C和D这三条线的逻辑电平都是 1 。 OUT 这条输出指令被执行之后，把A寄存器的内容循环左移 1 位。因此，现在A寄存器的内容是 11111101。然后把这个数存储在B寄存器里，它是下一个将输出给键盘的测试字。

8080 把第一个测试字输出给键盘后，输入V、W、X和Y这四条输出线的状态并进行测试。然后， 8080 执行这条 ANI 指令，把A寄存器的四位最高有效位(MSB)屏蔽掉。接着，8080 执行 CPI 指令，根据键 3、7、11 或 15 这四只键中的哪一只是否被按下，把标识位置位或者清零。如果“A”纵行中没有一只键被按下，那么，8080 扫描键盘的B行。如果 3、7、11 和 15 纵行没有一只键被按下，那么，D寄存器的键代码减 1 ，然后检查它是否已经被减量到 377 (FF)。减量到 377 表明， 键盘

的四个纵行都已经被扫描过。如果D寄存器的内容不等于377，那么，必须测试键盘的键矩阵的下一个纵行。如果D寄存器所存储的内容是377(FF)，那么，8080执行JMP-KEYSCN这段指令。当程序执行控制到达KEYSCN时，8080再次开始对键盘扫描。

如果某一只键被按下，8080则转移到NXTKEY，在NXTKEY，8080把键盘的四条输出线(V、W、X和Y)的逻辑值循环移入进位标识位（这四个输出值已经被输入)。当逻辑0循环移入进位时，8080把被按下的这只键的二进制代码存入D寄存器里，然后从KEYSCN这个子程序返回。相反，如果逻辑1被循环移入进位，则把004(04)加到D寄存器的内容上，以便为在同一纵行的下一只键产生键代码。8080继续执行该子程序的NXTKEY这个程序段内的指令，直到逻辑0循环移入进位标识位后才停止。

假设8080刚刚调入了KEYSCN这个子程序，并且按下“7”这只键。把测试字1110输出给锁存器，从而可以测试3、7、11和15这一纵行的那些键。请读者记住，用这个程序循环移动并存储这个测试字，目的在于建立一个测试字，供下一纵行的测试用。要输入和测试键盘的四条输出线的逻辑电平，确定被测试的这一纵行是否有键按下。在这个例子中，8080已经输入测试图形1101，因为“7”这只键已被按下。该子程序的NXTKEY这部分把这个输入字循环移入进位。如果进位等于0，8080返回到调入KEYSCN的程序。相反，如果把逻辑1循环移入进位，则把004(04)加到这一纵行开始的键代码上，为所测试的这一纵行的下一只键产生键代码；在这个例子中，这只键是“7”这个键。然后，把这个测试字再次循环移位。第二次循环移位之后，在进位标识位检测到的值是0，然后，8080把

被按下的这只键的键代码存入D寄存器里，于是，从这个子程序返回。如果测试一个纵行，而没有发现被按下的键，那么，D寄存器存储的起始键代码减 1，以便和下一纵行的起始键代码相一致。

假设同时按下去两只键；例如："9"和"12"这两只键，8080从KEYSCN这个子程序返回后，把哪只键代码存储在D寄存器呢？8080将把键代码011(09)存储在D寄存器里，然后返回。这是因为，8080在测试D和Y两条线的组合信号之前，测试C和X这两条线的组合信号。读者能确定所有各个键的优先权吗？也就是说，如果同时按下去两只或两只以上键，读者能够编制一个表用来表示哪只键代码被存储在D寄存器里吗？我们所规定的优先权如下：

3>7>11>15>2>6>10>14>1>5>9>13>0>4>8>12

如果我们能够使用25只键的键盘（其结构是5×5只键的矩阵），这个键盘能够和8080微型计算机连接，并可以一起使用，那么，对于硬件和软件而言，我们必须做出什么改变呢？我们必须在接口电子器件上增添一位输出端口位和一位输入端口位。该键盘的接口电路如图7-6所示。

读者可以看到，我们还需要增添一块四位的锁存器集成电路(SN 7475)；当8080执行适当的输出指令（OUT）时，能够锁存数据总线的附加位D_4。我们并不需要再增添一块DM 8095集成电路，因为每一块DM 8095集成电路包含有六个独立的三状态缓冲器。因此，只要把键盘增添的一排键与DM 8095那些原来不用的一个三状态缓冲器连接，并且，把这个三状态缓冲器的相应的输出与数据总线的D_4位连接就行了。

我们完全不需要对原来所使用的子程序（KEYSCN）增添

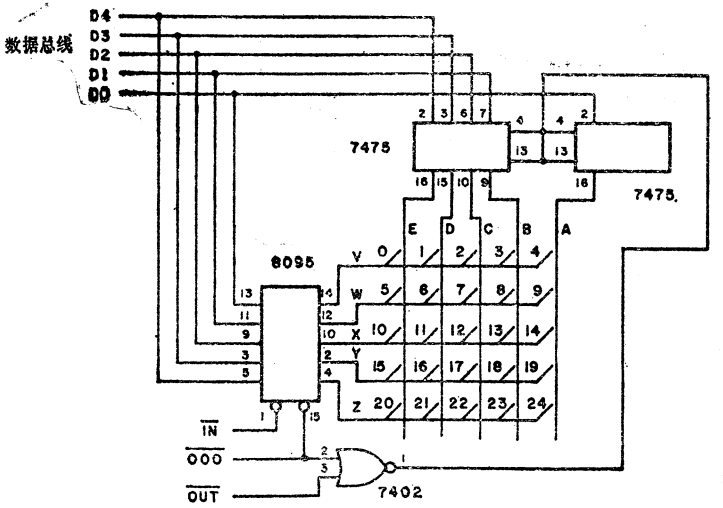


图 7-6　5 × 5 矩阵键盘的接口电路

任何指令，这是令人十分奇怪的！但是，必须对这个子程序做出一些改变。我们必须把 MVID 这条指令的数据字节从 003(03) 变成 004(04)，因为“4”这只键是可能被检测的第一只键。而且，还必须把 ANI这条指令的数据字节从 017 (0 F) 改变成 037(1 F)；因为，现在，键盘有五条数据输出线：V，W，X，Y 和 Z。我们还必须把这条 CPI 指令的数据字节从 017(OF)改变为 037(1 F)。最后，把该子程序的 NXTKEY指令段的 ADI 指令的数据字节从 004(04)变为 005(05)。例 7-14 所列出的这个 KEYSCN 子程序，我们已经把这些改变之处编进去了。

不管我们所使用的键盘的结构是 5×5 或是 4×4 的矩阵，对于这两个子程序（例 7-13 和 7-14 ）而言，都存在着一个严

重的问题。**这两个程序并没有提供软件来消除键闭合所产生的跳动作用**。我们以 4×4 矩阵键盘的扫描子程序为例，例 7-11 所采用的 10 毫秒延时子程序将可以用来消除键闭合 所 产生的跳动作用。

该子程序（例 7-15）读闭合键所用的方法与前面 的 子程序例所用的方法是相同的。这个子程序的唯一的改变是在该子程序的开始，即 NXTKEY。当 8080 读被按下的一只键时，它转移到 NXTKEY，以便准确地确定哪一只键被按下。 但是，8080 首先调用 DELAY 这个延时子程序，以 便抵消该键释放所产生的跳动作用。 当 8080 从 DELAY 这个延时子程序返回时，它执行 AGAIN 上的 RRC 指令。 如果这条指令把逻辑 0 移入进位，8080 则执行 JNC—RELESE 这段指令， 而不执行 RNC 指令（在前面的子程序中，我们使用了这条指令）。

当 8080 执行 RELESE 上的指令时， 它正在等待被按下的这只键释放。当这只键被释放时，键盘的四条输出线将处于逻辑 1 电平，因而 8080 的四条输入线的逻辑电平也是 逻 辑 1 。因此，8080 将继续执行 RELESE 这个循环中的指令， 直到所有这些线被读出的逻辑电平都是逻辑 1 为止。当这只键被释放时，8080 第二次调用 DELAY 这个延时子程序，从而 8080 能够消除因这只键被释放所产生的键的跳动作用。

读者在表 7-4 可以看到，用软件指令取代硬件（集成电路和连线），有许多优点，也有一些缺点。

例 7-14 5×5 矩阵键盘的扫描子程序

```
/这个子程序用来扫描矩阵键盘，
/该键盘的结构有 5 排，每排有
/ 5 只键，即 5×5 只键的矩阵。
KEYSCN, MVID     /把可能被读出的第一只
```

```
          004        /键的代码装入 D 寄存器里。
          MVIB       /把用来一次启动一排键的
          376        /这个字装入 B 寄存器里。
NXTGRP, MOVAB        /把这个测试字从 B 取到 A。
          OUT        /然后把它输出给键盘。
          000
          RLC        /把这个测试字循环左移一位，
          MOVBA      /然后把它保存在 B 寄存器里。
          IN         /从五排键输入
          000        /数据
          ANI        /只保存 5 位最低有效位，
          037        /它包含排测试数据。
          CPI        /把 037 (1 F)与这个输入字进行
          037        /比较，确定是否有被按下的
          JNZ        /键。这一排的一只键被按下，
          NXTKEY     /所以，确定它是哪一只键。
          0
          DCRD       /被测试的这一排没有键按下，
          MOVAD      /所以，这个键代码减 1，然
          CPI        /后判明五排键是否都已经被测
          377        /试完毕。377＝十六进制 FF。
          JNZ        /并不是五排键都被测试了，
          NXTGRP     /所以，测试另一排。
          0
          JMP        /五排键都已经被测试完毕，并
          KEYSCN     /且没有键被按下，
          0          /所以，继续扫描。
NXTKEY,RRC           /把这排测试数据移入进位。
          RNC        /当进位等于 0 时，返回。
          PUSHPSW    /否则，把 PSW 保存在堆栈上。
```

```
MOVAD   /然后，把 D 里的键代码加 5 。
ADI
005
MOVDA   /把这个键代码保存在 D 寄存器
POPPSW  /里。从堆栈弹出 PSW，
JMP     /然后再试，
NXTKEY  /直到进位标志为 0 时为止。
0
```

ASCII 键盘与 8080 微型计算机的连接

许多键盘有 50 只甚至更多的键，适合于商用。这些键盘都带有编码逻辑电路，所以，当一只键被按下时，这只键的给定代码就会产生。一般说来，这些键盘产生 7 位或 8 位的 ASCII 字符。那么，这种键盘怎样与 8080 微型计算机连接呢？

例 7-15 具有消除抖动作用的键盘（4×4）扫描子程序

```
/这个子程序用来扫描 4×4 矩阵的
/键盘。这种键排列结构分为四排，
/每排有 4 只键。此外，这个子程序还
/增添了一些指令，用来消除键的抖动
/的影响
KEYSCN, MVID    /把可能被读出的第一只键
        003     /的代码装入 D 寄存器里。
        MVIB    /把被用来同时启动一排键
        376     /的字装入 B 寄存器。
NXTGRP, MOVAB   /把这个测试字从 B 取到 A，
        OUT     /然后把这个字输出给键盘。
```

```
          000
          RLC       /把这个测试字循环左移1位，
          MOVBA     /然后把它保存在B寄存器里。
          IN        /输入四排键
          000       /的数据。
          ANI       /只保存4位最低有效位(LSB)，
          017       /它包含排测试数据。
          CPI       /把 017(0 F)与这个输入字
          017       /比较，判明是否有键被按下。
          JNZ       /在这一排，有一只键被按下，
          NXTKEY    /所以确定它是哪一只键。
          0
          DCRD      /在被测试的这一排，没有一只
          MOVAD     /键被按下，所以使这个键字减
          CPI       /1，然后判断四排键是否都已
          377       /经被测试完毕。377＝十六进制 FF。
          JNZ       /四排键还没有全部测试完，
          NXTGRP    /所以，测试另一排。
          0
          JMP       /四排键都已经被测试完毕，
          KEYSCN    /并没有键被按下，所以，
          0         /继续扫描。
NXTKEY,   CALL      /一只键被按下，所以执行
          DELAY     /10 MS 的延时子程序。
          0
AGAIN,    RRC       /把排测试数据循环移入进位标识位，
          JNC       /找到了这只键，所以，等待它
          RELESE    /被释放后，才从该子程序返回。
          0
          PUSHPSW   /否则，把 PSW 保存在堆栈上，
```

```
        MOVAD   /并把 D 寄存器的键代码加 4 。
        ADI
        004
        MOVDA   /把这个新键代码保存在 D 里。
        POPPSW  /从堆栈弹出 PSW，
        JMP     /然后，再试，以便使
        AGAIN   /进位等于为零。
        0
RELESE  IN      /再一次输入这个数据字。
        000
        ANI     /只保存代表四排键的
        017     / 4 位数据。
        CPI     /当没有键被按下时，把得到的
        017     /数与该数比较(017＝十六进制 0F)。
        JNZ     /如果这只键还是被按下的，
        RELESE  /则转移，然后等待它被释放。
DELAY,  PUSHPSW /把 PSW 保存在堆栈上。
        PUSHD   /然后，把寄存器对 D 的内容保
        LXID    /存在堆栈上。把数 003 101(十六
        101     /进制 0341) 装入寄存器对 D 里。
        003
WAIT,   DCXD    /这个数减 1 。
        MOVAD   /把这个数的最高有效字节传送到
        ORAE    /A 。把它与最低有效字节进行
        JNZ     /“或”操作。如果其结果不等于 0 ，
        WAIT    /则转移到 DCXD 指令。
        0
        POPD    /当结果等于 0 时，从堆栈弹出
                /D 和 E 的内容，
        POPPSW  /然后，从堆栈弹出 PSW，
```

```
RET     /把这个键代码存入 D 寄存器，
        /然后返回。
```

表 7-4　　用软件代替硬件的优缺点

优　点
1. 微型计算机生产成本低，因为所需要的硬件较少。 2. 所需要的印刷电路板较少，只要增添软件不需要增加大量的存储器。 3. 硬件电路不是很复杂，这就意味着，生产中所需要的调试时间较短。 4. 复制软件比复制硬件容易。别人可以为你代编 ROM 程序。 5. 灵活性好。比较容易使软件适合某些方面的需要，因为只要插入一些附加存储器集成电路即可。在应用方面修改硬件，有时可能很简单，有时可能很复杂。
缺　点
1. 执行的速度较低。在确定适当的键时，用软件对键盘扫描与用硬件编码器比较，前者需要用去较多的时间。 2. CPU 的执行速度或周期对可能实行快速的输入/输出操作带来了限制。 3. 开发软件的成本可能是开发硬件的成本的许多倍。由于软件代替了硬件，节省成本主要来源于微型计算机系统的生产。

把这样的一个键盘和把带有硬件编码逻辑电路的 16 只键的键盘（如本章第一个键盘程序例）与 8080 微型计算机连接，其连接的难易程度几乎相当。如前面的键盘程序例所示，一定有某种方法（例如，或者通过硬件或者通过软件）告诉 8080 微型计算机，一只键已经被按下。大多数 ASCII 键盘都有一条状态线用来表示是否有键被按下。因为这是硬件状态位，也就是标识位，所以我们必须将它输入给 8080；8080 执行几条程序指令，就一定能够确定标识位的状态。

让我们假设与 8080 连接着的键盘所产生的 ASCII 字符是 8 位的字符。因此，这个接口需要两个输入端口。一个输入端

口用来输入键盘的状态标识位，另一条输入端口用来输入键盘的编码逻辑电路所产生的 8 位的 ASCII 字符。该键盘的接口如图 7-7 所示。

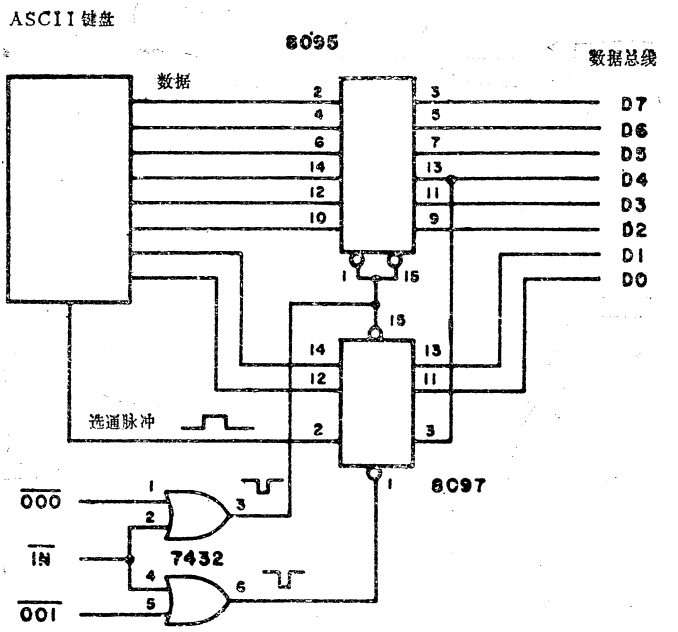


图 7-7　8 位 ASCII 键盘的接口

如果 8080 执行一条 IN 001 指令，则会把**键盘的状态位**输入给 A 寄存器(D_4位)。当 8080 执行一条 IN 00 0指令时，所按下的这只键的 8 位 ASCII 字符将被输入到 A 寄存器里。读出闭合的键所需要的软件，以及输入 8 位 ASCII 字符所需要的软件，与我们前面用来读出带硬件编码器的十六只键的键盘的键代码所用的软件很相似。

读者检查这个程序，就应该能够确定，键盘的某只键被按下时，和没有键被按下时，键盘的状态线的状态。当键没有被按下时，键盘的标识位，即状态线的逻辑电平等于 0；当键盘上的某一只键被按下时，键盘的标识位，即状态线的逻辑电平等于 1。当然，有些 ASCII 键盘，其状态线输出的逻辑电平与此正好相反。这种键盘也可以使用例 7-16 这个程序工作，但是，应该把 JZ—ASCKEY 改换成 JNZ—ASCKEY。我们做出这样的改变比给接口电子器件增添一块反相器集成电路或许容易些。

例 7-16 检测和输入 ASCII 键盘的键代码的程序

```
/这个子程序用来检测 ASCII 键盘上的
/闭合键的代码，然后并行输入这个 8 位
/ASCII 键代码。
ASCKEY,  IN       /输入一个数据字，该字包含
         001      /ASCII 键盘的状态位。
         ANI      /只保存键盘
         020      /的状态位。
         JZ       /如果该状态位等于 0，
         ASCKEY   /则转移到 ASCKEY，
         0        /因为没有键被按下。
         IN       /一只键被按下，所以把
         000      /这个 ASCII 代码输入到 A。
         RET      /然后返回到调用程序。
```

我们已经假设，该键盘产生一个选通脉冲，这个脉冲的宽度为 15 或 20 微秒。但是，有些 ASCII 键盘所产生的逻辑电平，与我们前面已经讨论过的 16 只键的键盘所产生的逻辑电平相似。如果是这种情况，则将必须执行软件指令，检测被按下的和被释放的 ASCII 键。读者希望把 ASCII 键盘与 8080

微型计算机连接，必须执行的软件指令实际上将由所连接的ASCII键盘的种类来确定。

8080和发光二极管显示器

在读者编程序的某点上，你或许需要8080把数据输出给外部设备。正如我们前面已经讨论过的，这种外部设备可能是电传打字机，CRT显示器，数字录音机，或者软磁盘。但是，与微型计算机连接的最普通的外部设备之一是某种形式的发光二极管显示器。因此，我们将要讨论的是把一些七段发光二极管显示器与8080微型计算机连接。这些显示器与用于袖珍计算器和数字时钟的显示器，都属于同一种类型。

对于8080的某一具体应用而言，读者需要8080把十位BCD数输出给七段的发光二极管显示器。完成这一任务的方法之一是，每一位数字使用一块SN 7475锁存器集成电路，一块SN 7447译码器集成电路(或一块SN 7448)，7个电阻和一个7段发光二极管显示器。把这些器件连接到一起，供输出一位数字用的电路如图7-8所示。

读者从这个图可以看到，锁存器SN 7475的输入线与8080的数据总线连接。当8080执行OUT 125这条指令时，SN 7475锁存数据总线的四位最低有效位。这四位数据值出现在该锁存器的输出端，用来驱动SN 7447译码器。此译码器决定显示器的哪几段发光二极管应该点亮，从而实际地显示被锁存的数据值。一旦数据值被SN 7475锁存后，只需要译码器用大约50毫微秒时间给这个数译码，并且点亮发光二极管显示器的相应的显示管。为了更详细地了译解码器和七段发光二极管显

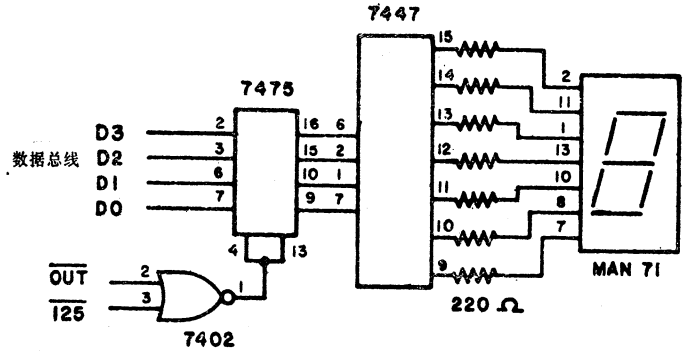


图 7-8　带有七段显示器的 4 位输出端口

示器，读者应该参考«逻辑与存储器实验用 TTL 集成电路»(Logic and Memory Experiments using TTL Integrated Circuits)的第一册和第二册。第一册第五章全部用来讨论译码器；第二册第六章全部用来讨论发光二极管显示器(LED)。这两册书为读者提供了可以用这些器件来做的许多实验。

如果另外还用一个锁存器，译码器和显示器(图 7-9)与一个锁存器(该锁存器与数据总线的 D_4～D_7连接)的输入端连接，8080就能同时把两位 BCD 数输出给同一条输出端口。这样，就能有效地利用 8 位数据传送线来传送两位 BCD 数字，(一位 BCD 数字包括 4 位二进制数据)。

为了在这些显示器上显示 39 这个数，8080 必须执行什么程序指令呢？为了显示这个数，8080可以执行例 7-17 所示的程序指令。这个程序只是把一个立即数字节装入A寄存器里，然后把这个数据字节输出给显示器。一个锁存器与数据总线的 D_4～D_7 位连接，用来锁存0011这个数；另一个锁存器与数据总线的 D_0～D_3 连接，用来锁存 1001 这个数。这就是说，两

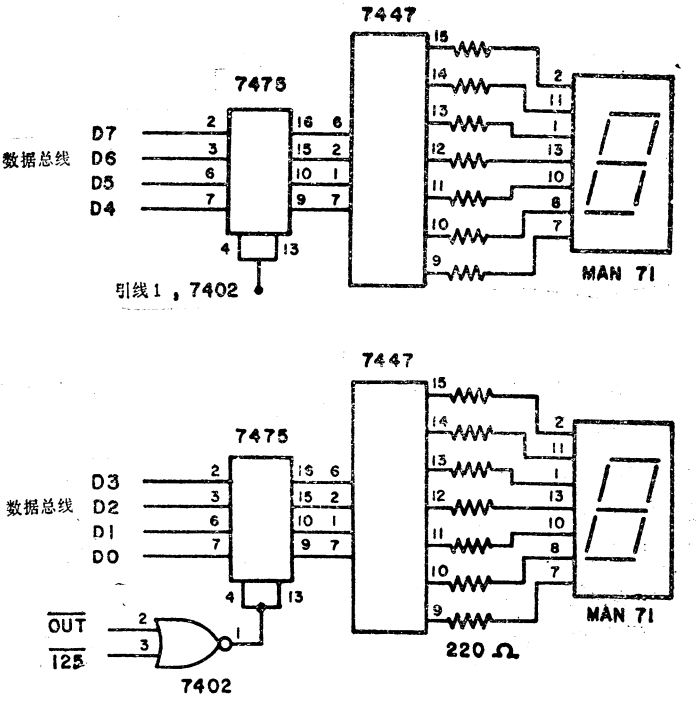


图 7-9　与 8080 连接的两位数字的七段 LED 显示器

位 BCD 数字可以被压缩成一个八位字。

例 7-18 所列出的这个程序是用来干什么的呢？假设 8080 的时钟周期是 500 毫微秒。这个程序使这个被显示的数大约每隔四分之三秒增量一次。这个计数数从 00 开始，最大达到99。当这个计数数达到 99 时，它再回到 00，然后继续进 行计数。每当这个计数数加 1 时，这个程序的 DELAY(延时)程序段产生四分之三秒的延时时间。加上这个延时时间以后，我们才能看得见显示器上的计数数的变化情况。实际 上为 了产生 这个数，应该把001(01)与存储在 B 寄存器的这个数 相 加，然后，8080执行 DAA 这条指令，对其结果进行 调整。这就是说，B 寄存器的内容总是保持两位 BCD 数字，其值在 00 和 99 之间。

例 7-17 怎样把 39 这个数输出给七段的 LED 显示器

```
/这一程序段用来把数据图形00111001
/(八进制071，十六进制39)输出给输出
/端口，该端口装有两个七段 LED 显示器
  .
  .
MVIA/把下一个立即数字节(十六进制39，
017   /二进制00111001)装入 A 寄存器里。
OUT   /再把它输出给带有锁存器
125   /的七段 LED 显示器。
.     /继续执行这个程序
.     /的其余指令。
```

例 7-18 在两个七段 LED 显示器上显示计数数字

```
START,  MVIB   /把000(十六进制000)装入
        000    /B 寄存器，B 用作暂存器。
DISPLY, MOVAB/把 B 的内容传送到 A 寄存器里。
        OUT    /把 A 的内容输出给带有
```

```
        125     /锁存器的七段 LED 显示器。
        ADI     /把数 1 加到 A 寄存器
        001     /的内容上。
        DAA     /对其结果进行十进制调整，
        MOVBA/然后把该结果保存在 B 寄存器里，
        CALL    /调用这个延时子程序，它需要执行
        DELAY /大约 3/4 秒。
        0
        JMP     /然后返回到该程序
        DISPLY /的显示用部分。
        0
DELAY,  LXID    /把 000 000(十六进制 0000)
        000     /装入寄存器对 D。
        000
WAIT,   DCXD    /寄存器对 D 的内容减1 。
        MOVAD/把最高有效字节传送到 A 寄存器。
        ORAE    /把该字节与最低有效字节相"或"。
        JNZ     /如果该结果不等于 0，
        WAIT    /则返回到 WAIT。
        0
        RET     /否则，返回到主程序。
```

为了研究更复杂的问题，让我们来设计硬件和编写软件，显示被存储在存储器的 10 位 BCD 数字。为了完成这一任务，必须把前面的五个电路(图7-9)组合起来。因为每个显示器具有显示两位数字的能力，所以用五个电路可以显示 10 位数字。当然，为了显示一个 10 位数字的数，五个电路必须具有不同的器件地址。

我们把这个数存储在五个连续存储单元，它以两位压缩的 BCD 数字形式存放每个存储单元。2,389,421,365 这个数的

存储情况如表 7-5 所示。读者可以看到，应用十六进制数制的优点之一是容易处理 BCD 数。为了显示这个数，我们需要使用 10 个 SN 7475 锁存器，10 个 SN 7447 译码器，70 个电阻，10个七段的 LED 显示器，以及一些附加门电路，才能组成这个接口。我们假设，这些锁存器的器件地址是 125，126，127，130和131(55，56，57，58 和 59)，其中器件地址131(59)用来锁存A寄存器所存储的压缩的 BCD 数，以便进入显示器的两位最高有效位。用来取存储器的数，并把它们输出给 LED 显示器的一个程序如例 7-19 所示。

在这个程序(例7-19)中，8080只从存储器取一个数，装入到A寄存器里，然后，把 8 位数送给输出端口，使存储器地址加 1，然后，把下一个存储单元的内容输出给另一条输出端口。一旦五个数据字(每个字有两位 BCD 数字)都已经被输出以后，8080才能从 DISPLY 这个子程序返回。被显示的十位数字将一直处于显示状态，直到 8080 再调用 DISPLY 子程序为止。从当时起，这种情况一分，一时或一日都可能发生。

表 7-5　　把2,389,421,365 这个数存入存储器

八进制		十六进制	
存储单元	内容	内容	存储单元
004120	145	65	0450
004121	023	13	0451
004122	102	42	0452
004123	211	89	0453
004124	043	23	0454

例 7-19　10 位数字的 LED 显示器的程序

```
/这个子程序用来显示五个
/连续的存储单元的内容，
/其中每一个单元存储两位被压缩的 BCD 数字。
DIPLY, LXIH     /把要显示的
       120      /这个数的存储器地址
       004      /装入寄存器对H。
       MOVAM    /从存储器取一个字。
       OUT      /把这个字输出给两位显示位。
       125
       INXH     /这个存储器地址加 1 。
       MOVAM    /从存储器取一个字。
       OUT      /把该字输出给两位显示位。
       126
       INXH     /这个存储器地址加 1 。
       MOVAM    /从存储器取一个字。
       OUT      /把这个字输出给两位显示位。
       127
       INXH     /这个存储器地址加 1 。
       MOVAM    /从存储器取一个字。
       OUT      /把这个字输出给两位显示位。
       130
       INXH     /这个存储器地址加 1 。
       MOVAM    /从存储器取一个字。
       OUT      /把这个字输出给两位显示位。
       131
       RET      /从这个子程序返回。
```

用这种方法显示存储器的内容，其最大的问题或许是：显示器接口所需要的集成电路较多。如果批量生产，十个锁存器、十个解码器和七十只电阻需要二十多美元。这还不包括完

成该接口所需要的显示器、器件地址译码器和各种集成电路的费用。读者可以看到，这种显示器是比较花钱的。

如果我们采用显示器多路转换的方法，显示同样数量的信息，所花的费用则少得多。显示器多路转换就是一次只从多个显示数字中显示一位。但是，显示器的数码管的导通和截止是非常快的，人眼看上去，显示器好象一直处于显示状态。一次只显示一位数字，这样，则需要两个锁存器和两个译码器来驱动显示器。有了两个译码器，不仅可以为数据总线译码，使显示器的所需要的字段导通，而且，要使电流按时以某一种时间间隔流过显示器的一个显示管。这种方法好象是一种显示数据的复杂方法，但是，用电子技术来实现，则是相当简单的。因此，**电子计数器和数字表的数字显示就是采用显示器多路转换的技术**。实际上，除非你用烛光或电光在读那些数字，那么，你正在使用的光确实是在时亮时灭，每秒钟为 60 次。这个速度确实太快，所以，你的眼睛不能检测它。你的电视机屏幕上的图象也是如此。

十个数字多路转换的 LED 显示器的电路如图 7-10 所示。读者分析这个电路图可以看到，该接口只采用了一个 8 位输出端口。这就只需要两块 SN 7475 锁存器集成电路，一块 SN 7442 译码器和一块国家半导体公司(National Semiconductor)生产的 DM 8857 七段显示器的驱动器。国家半导体公司生产的 NSA-5140 显示器是专门设计的，用来以多路转换的方式显示七段数据字。虽然这个显示器可以显示十四位数字，但是，我们仅使用十位。

为了在这种显示器上显示数据，我们必须把数字选通码(我们将要显示的这个数字)装入 A 寄存器的 D_4～D_7。为了在第三位数字上显示“5”这个数，我们应该把 01010010 装入 A 寄存

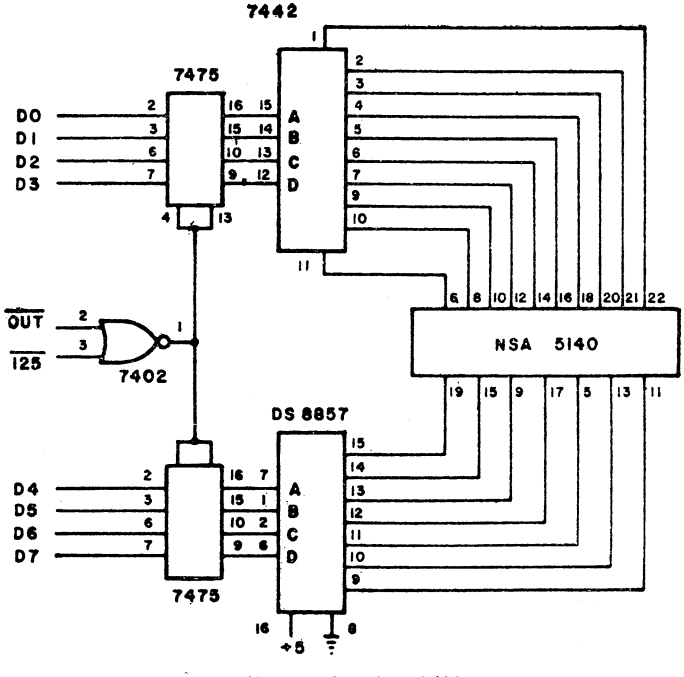


图 7-10　微计算机控制的十位数字多路转换的 LED 显示器

器里，然后把它输出给接口电路。因为我们要使用的显示器有十位数字，所以数字选通码是从 0000～1001，这就是说，第三位数字的选通码是 0010。当 8080 执行一条 OUT 125 指令后，0101 被 SN 7475 锁存(SN 7475 与 DM 8857 集成电路连接)；0010 被 SN 7475 锁存。SN 7475 与 SN 7442 译码器连接，SN 7442 被用来作为数字选通译码器用。

按照集成电路与 8080 微型计算机连接的方式，数字选通码为 0000 的这位数是在最右边(最低有效位数字)。选通码为 1001 的这位数字在最左边(最高有效位数字)。2,389,421,365

这个数仍旧被存贮在存贮器里，为了显示这个数，8080 可以执行例 7-20 所列出的这个程序。在这个程序的开始，把一个存储地址装入寄存器对H里；这个存储地址用来存储两位最低有效数字(LSD)。为了存储这个十位数，只需要五个存储单元，因为它是以压缩的 BCD 格式被存储的。然后将D寄存器(用来存储数字选通码的)置零。最后，D寄存器的内容将由接口硬件锁存，用来一次只显示一位数字。如果用执行速度很快的软件来完成这一任务，那么在显示器上显示的各位数字好象是同时出现似的。8080 程序执行到达 DISPL 1 时，调入 DIGIT 子程序。8080 的程序执行控制到达 DIGIT 时，它把寄存器对H寻址的存储单元的内容传送到A寄存器。因为数值是以被压缩的 BCD 数字形式存储的，所以，将必须用这个子程序把它们展开。我们必须首先显示A寄存器的四位最低有效位的这位数。因此，要把A寄存器的内容循环左移四次。现在，被显示的这位数字存储在A寄存器的 D_4～D_7 位。然后，8080 调用 OUTIT 子程序。

程序执行控制到达 OUTIT 时，8080 执行一条 ANI 指令，把A寄存器的四位最低有效位屏蔽掉。现在，只是把要显示的这位数字存储在A寄存器的 D_4～D_7。这个数字选通码存储在D寄存器里，然后，把它加到A寄存器的内容上，接着，该加操作的结果被输出给输出端口 125 (55)。当这个字被输出时，用四位最低有效位确定哪位数字将要选通；四位最高有效位(MSB)代表着在被选通的这位上要显示的这个数。如果寄存器对H寻址的存储单元存储着被压缩的 BCD 字 37(十六进制 37，八进制 67，二进制 00110111)，那么，8080 执行 OUTIT 子程序的这条输出指令时，A寄存器所存储的内容是什么呢？A寄存器的内容是 01110000 (八进制 160，十六进制 70)。这就是

说，“7”这个数字将在数位0这个位置上显示。这是显示器的最右边的那位数字，即最低有效位数字(LSD)。

在OUTIT这个子程序的末尾，D寄存器的内容加1，即从000(00)增至001(01)。这就是说，要被选通的下一位数字将恰好是现行显示的这个数字的左边的数字。请读者记住，这位数字将持续显示，一直到8080执行另一条输出指令，使显示器内的另一位数字选通时为止。当8080从OUTIT这个子程序返回时，它返回到正好在OUTIT子程序之前的那条MOVAM指令。

这条MOVAM指令，把上面这个压缩的两位BCD数字从存储器传送到A寄存器；但是这一次，必须把这个字的最高有效的BCD数字进行显示。因此，并不执行循环移位指令。所以，在OUTIT处，这条ANI指令，把A寄存器的D_0～D_3位置0，而把即将被显示的第二位数字保留在A寄存器的D_4～D_7位。然后把D寄存器的选通数字码加到A寄存器的内容上，并且，把该结果输出给显示器接口电子器件。这时，数字选通码是001(01)，所以，启动显示器上的第二位数字。只有这种情况出现时，第一位数字才消失。然后，8080执行INRD指令，使D寄存器的内容增量到002(02)；这条RET指令把程序执行控制送回到这条INXH指令。

例 7-20 多路转换的10位数字的七段LED显示器用子程序

```
/这个程序用来驱动
/七段发光二极管
/显示器，使十位数字多路转换输出。
DISPLA, LXIH      /把这个存储地址(存
        120       /BCD数)装入寄存器对H。
```

```
        004         /004120=十六进制 0450 。
        MVID        /把要选通的第一位数
        000         /字装入D寄存器里。
DISPL1,CALL         /显示头两位被压缩的
        DIGIT       /BCD 数字。
        0
        INXH        /这个存储地址加 1 。
        MOVAD       /把这个数字选通字取到A。
        CPI         /把它与第十一位数字选通计数
        012         /数进行比较。
        JNZ         /十位数字还没有全部
        DISPL 1     /显示完毕，所以再显示两位。
        0
        JMP         /十位数字都显示完了，
        DISPLA      /所以，再一次全部显示它们。
        0
DIGIT, MOVAM        /把这个被压缩的 BCD 字取到A。
        RLC         /把这四位最低有效位循环左移
        RLC         /进入四位最高
        RLC         /有效位。
        RLC
        CALL        /然后，显示这位数字。
        OUTIT
        0
        MOVAM       /再一次取同一个字，
OUTIT, ANI          /只保存四位最低有效位。
        360         /(360 十六进制 F0)。
        ADDD        /加这数字选通位。
        OUT         /输出这个 8 位数。
        125
```

```
INRD     /使数字选通位加1。
RET
```

这条INXH指令使存储在寄存器对H的地址加1，从而它对存储在存储器的下一个被压缩的BCD字寻址。然后，8080进行检查，看存储在D寄存器的数字选通码是否等于012(0A)。这个数比需要显示的第十位，即最高有效位所需要的数字选通码大1。如果十位数字还没有被全部显示，8080则执行JNZ-DSPL 1这段指令，从而显示存储在一个存储单元的下两位BCD数字。如果十位数字都已经被显示完毕，8080则返回到DISPLA。在DISPLA处，为数字选通代码和存储器地址预置初值，从而重复显示过程。这就是说，8080继续往下执行指令，把十位数字信息全部显示。

在我们的8080微型计算机上执行这个程序时，我们发现显示器显示的第二位，第四位，第六位，第八位和第十位数字的亮度比其余五位数字更强。对于这种显示亮度的不同，读者能够解释吗？有些显示数字的亮度比其它一些显示数字的亮度强，其理由是，由于8080显示一位数字相对于另一位数字所用的时间不同。取和显示最低有效BCD数字与取和显示包含在同一个8位字内的最高有效BCD数字两者比较，哪一种情况占有8080的时间较长呢？取和显示被压缩成一个字的两个最低有效数字占用8080的时间稍长。读者知道其理由吗？理由是：从存储器取一个最低有效BCD数字，把它移位，进入A寄存器的D_4～D_7位，然后显示，占用8080较长的时间。此外，一个字的最高有效BCD数字被显示后，8080必须从DIGIT子程序返回。然后把这个存储地址加1，并检查数字使能码，看它是否等于012(0 A)。如果不相等，则8080再调入DIGIT子程序，以显示下一个8位字的最低有效的BCD数字。所以，

处理这些最低有效BCD数字占用8080的时间较长，因此，显示最高有效的BCD数的时间也就较长了。

为了解决这个问题，我们可以在这个显示用程序中加入一个延时子程序，这个延时子程序所产生的延时时间必须比显示一个8位字所包含的两位数字所用的时间差大许多倍。因此，即使显示最高有效BCD数字的时间比显示最低有效BCD数字的长，但是，这两个时间的实际差别是非常少的。例7-21列出了一个DISPLA程序，其中包含这个延时子程序。

最高有效数字(在显示器的左边)不能首先显示，有什么理由可说吗？绝对没有。例7-22所列出的这个程序正好回答了这个问题。这就是说，数字使能将从011(09)开始的，逐渐减到000(00)，然后再到377(FF)。这意味着，从存储器读出一个字时，首先显示这个字的最高有效BCD数字，然后显示这个字的最低有效BCD数字。

例7-21 带有增加亮度指令，十位数字的，多路转换的显示器程序

```
/这个程序驱动十位数字的多路转换的
/七段发光二极管显示器(LED)。
/另外，还增添了一些指令，从而使
/每一位显示数字的亮度相同。
DISPLA,   LXIH    /把这个存储地址装入寄存器对H。
          120     /该地址用来存储BCD数字。
          004     /004 120，十六进制0450。
          MVID    /把要被使能的第一位数字
          000     /装入D寄存器里。
DISPL1,   CALL    /显示前两位被压缩的
          DIGIT   /BCD数字。
            0
```

```
          INXH     /使这个存储地址加1。
          MOVAD    /把数字使能字取到A，
          CPI      /然后，把它与第十一个使能数字
          012      /进行比较。
          JNZ      /十个数字还没有全部被
          DISPL1   /显示完毕，所以再显示两位。
             0
          JMP      /十位数字都显示完了，
          DISPLA   /所以，重复上述显示过程。
             0
DIGIT,    MOVAM    /取这个被压缩的BCD字，装入A。
          RLC      /把四位最低有效位循环左移。
          RLC      /进入四位最高有效位。
          RLC
          RLC
          CALL     /然后显示这个数字。
          OUTIT
             0
          MOVAM    /取这个被压缩的字到A。
OUTIT,    ANI      /只保存四位最高有效位。
          360      /(360，十六进制F0)。
          ADDD     /加这一位使能数字。
          OUT      /输出这个8位数。
          125
          DCRD     /数字使能字减1。
INTENS,   MVIE     /把一个数装入E寄存器里。
          100      /100二十六进制40。
INTEN1,   DCRE     /这个数减1。
          JNZ      /如果它不等于0，
          INTEN1   /则执行这条TNZ指令，
```

```
        0       /(返回到INTEN1)。
    RET         /当E=0时，则返回。
```

例 7-22 所列出的这个程序，就其长度而言，正好与例 7-21 所列出的程序相同；而例 7-21 的程序首先显示这位最低有效 BCD 数字。但是，如果首先要显示这位最高有效 BCD 数字，那么，就必须对这个程序做出许多修改。我们必须改变位于该程序的开头的这条 LXIH 指令的两个地址字节，才能把存储这两位最高有效数字的地址装入寄存器对H。这是要显示的头两位数字。MVID 这条指令的数据字节也必须改变，这样才能首先使显示器的最高有效数字处于使能状态。因为首先显示最高有效数字，所以 8080 必须执行 DCXH 和 DCRD 这两条指令，而不是执行 INXH 和 INRD 这两条指令。所以，8080 调入 DIGIT 子程序之后，将寄存器对H的存储器地址减 1。这个地址减 1 之后，8080 把D寄存器的内容与 377(FF)进行比较。这是因为所用的最后一个数字使能码是 000(00)，它是显示器的右边的这位数的数字使能码。

例 7-22 首先显示最高有效数字的程序

```
/这个程序用来驱动十位数字的、
/多路转换的七段发光二极管显示器。
/另外，还增添了一些指令，以便使每一位
/显示数字的亮度相同。
/这个程序用来首先显示最高有效数字。
DISPLA,   LXIH    /把这个存储地址装入寄存器对
          124     /H，该地址用来存储最高有效
          004     /位 BCD 数字，004 124，十六进
          MVID    /制 0454。把要使能的第一
          011     /位数字装入D寄存器。
DISPL1,   CALL    /显示头两位压缩
```

```
          DIGIT    /的 BCD 数字。
             0
          DCXH     /存储器地址减 1 。
          MOVAD    /把这个数字使能字送到A寄存器。
          CPI      /把它与第 11 个使能数字比较。
          377
          JNZ      /十个数字还没有全部显示完毕，
          DISPL1   /所以显示另外两个数字。
             0
          JMP      /十个数字都已经显示完了，
          DISPLA   /所以，再重复这一显示过程。
             0
DIGIT,    MOVAM    /取这个压缩的 BCD 字，送到
          CALL     /A寄存器。首先显示这个字
          OUTIT    /的最高有效数字。
             0
          MOVAM    /从存储器取同一个字到A寄存
          RLC      /器里。把四位最低有效位循环移
          RLC      /入四位最高有效位。
          RLC
          RLC
OUTIT,    ANI      /只保存四位最高有效位。
          360      /(360，十六进制 F 0)。
          ADDD     /加使能数字。
          OUT      /输出这个八位数。
          125
          DCRD     /使能数字减 1 。
INTENS,   MVIE     /把一个数装入 E 寄存器。
          100      /100 (十六进制 40)。
INTEN1,   DCRE     /这个数减 1 。
```

```
JNZ       /如果它不是0，则执行
INTEN1    /JNZ 指令(返回到 INTEN1)。
   0
RET       /E 寄存器的内容＝0时，
          /则返回。
```

十位数字的多路转换方式的显示器和需要十个锁存器和十个译码器的显示器，它们的硬件和软件的主要差别是什么呢？当 8080 把这个十位数输出给前面所讨论的显示器的接口的十个锁存器时（这十位数的显示器，没有给出方框图；可参考图 7-9），仅须把这些信息输出一次，因为使用了十个锁存器，每个锁存器锁存一位数字。一旦输出后，8080 可以执行其他任务，因为十位数字的显示器继续显示锁存的数。只有8080第二次执行显示器子程序时，显示器才能改变显示的内容。

当 8080 执行多路转换的显示程序时，必须不断地更新显示的数据。这就是说，8080 必须不断地改变数字使能代码和数据字。必须这样做是因为，如果 8080 把任何一位使能数字保留的时间太长(5 秒或 10 秒)，LED 显示器将可能烧毁，这是由于驱动显示器的电流很大。显示器的每一位只能显示这个时间的十分之一，所以，为了获得相同的亮度，必须把大于十倍的电流加到该显示器上。如果 8080 用 20～30 秒时间执行其他某个操作，用该软件不要使用软件连续驱动显示器；在 8080 执行这个长长的指令序列之前，它应该把显示器关断。只有当 8080 有时机再进入显示器程序段时，显示器才会再次显示。当 8080 必须执行其他任务时，它不必考虑显示器显示与否；只要给图 7-10 所示的接口电子器件添加一些东西，就能完成这种相同的任务。

图 7-11 所示的这个电路使用了一块再触发的单稳态集成

电路 SN 74123。请读者注意，使 SN 7475 锁存器锁存来自数据总线(图 7-10)的 8 位数据的同一个脉冲，也可以用来触发这块集成电路。当这种情况出现时，该单稳态电路的输出在大约 10 ms 之内仍然处于逻辑 1 状态(引线 13)。该输出送到七段显示器的译码器(DM 8857)的使能输入端(引线 5)。如果该译码器的这个输入端总是被置成逻辑 0，那么，它的输出端将不会给显示器的任何字段提供电流。怎么能防止显示器上的一位数字显示得太久呢？如果 8080 每隔大约 10 ms 不执行 OUT 125 指令，该单稳态电路的 Q 输出端将回到逻辑 0，使显示器消隐或者熄灭。如果 8080 常常在小于 10 ms 时间内执行这条 OUT 125 指令，显示器将显示被锁存的数据。如果 8080 必须执行大于 10 ms 的时间的指令序列，那么，显示器将会熄灭。只有当 8080 执行 DISPLA 这个程序的指令时，显示器才显示。

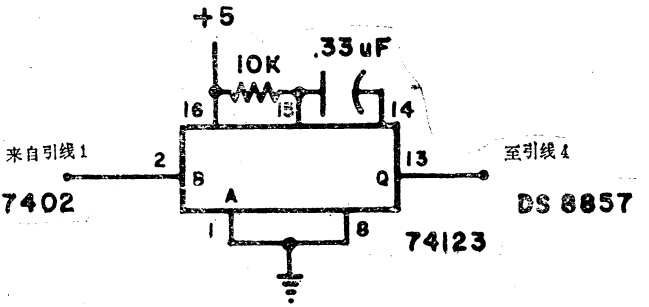


图 7-11　多路转换显示器电流自动截止电子器件

到目前为止，读者已经看到了许多不同硬件和软件的例子，这些实例可以与键盘和显示器一起使用。有些方法所需要的硬件较多，有些方法则需要的软件较多。至于选择什么方法，将由下列因素决定：(1) 读者需要设计和制造硬件的多

少；(2) 读者自己已经制成或购买的具体接口所具有的编程序的能力；(3) 把我们已经讨论过的软件程序实例应用于你的具体接口的能力。最后，当然并非不重要的一点是，你选择的硬件设计或软件设计，应该取决于该设计是不是完成你的具体接口任务的最好方法。

存储器映象输入/输出

到现在为止，读者对累加器输入/输出应该很熟悉了。当实行累加器输入/输出时，8080 微型计算机利用 IN 和OUT 这两条指令在 8080 微处理器的A寄存器和它的外部设备之间传送数据。如果把某个外部设备与 8080 微型计算机的地址总线及数据总线连接，那么，其连接的方式和连接存储器器件一样；如果控制信号$\overline{\text{MEMR}}$和$\overline{\text{MEMW}}$用来选通和取走数据总线上的数据，那么，利用 8080 的存储器访问指令，可以在 8080 和外部设备之间传送数据。

当外部设备与使用累加器输入/输出的微型计算机连接时，必须对八位地址译码，才能一次只启动一个输入/输出设备。对于存储器映象输入/输出而言，必须对十六位地址译码。为了完成这一任务，需要相当多的集成电路。因此，8080 微型计算机的一些厂家和一些用户常常把大量的存储地址划给存储器映象输入/输出设备。例如，有些用户把较上部分的 32 K的存储空间留下来，供存储器映象输入/输出设备使用。这就是说，当 8080 对存储器映象输入/输出设备寻址时，存储器地址位A_{15}是逻辑 1；当 8080 对存储器寻址时，A_{15}是逻辑 0。一个译码器可以供存储器映象写和存储器映象读设备使用，最高可

以达到 16 个设备，其电路如图 7-12 所示。

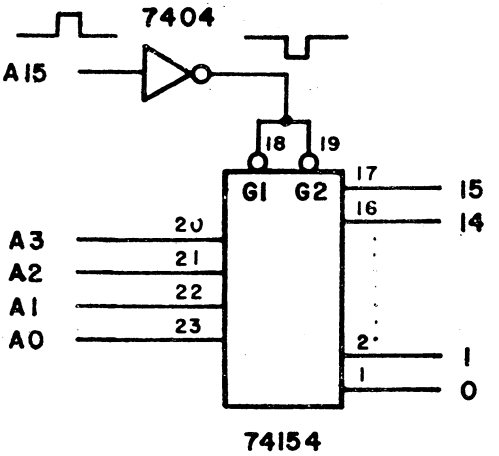


图 7-12　不完全存储器映象输入/输出设备地址的译码器

这个译码器的逻辑 0 输出，根据具体的存储器映象输入/输出设备的要求，仍然必须用$\overline{\text{MEMW}}$信号或者$\overline{\text{MEMR}}$信号进行控制。图 7-12 所示的译码器的输出什么时候置逻辑 0 呢？当下面的存储单元

10000000　　00000000

和

10000000　　00001111

之间的某一个被寻址时，译码器的输出将置逻辑 0。这些地址（10000000 00000000～10000000 00001111）对应于 200 000～200 017（8000～800 F）。当然，因为这个译码器是**不完全地址译码器**，（这就是说，不是 16 位地址位都被译码），所以，将使该译码器的输出为逻辑 0 电平的地址有 2^{12}个。例如，该译码器不能区分表 7-6 所列出的这些地址。所有这些地址将在引

线 1 输出端 0 都产生逻辑 0 输出。

表 7-6　　图7-12的译码器不能辨别的地址

10000000	00000000
10000001	00000000
10000000	11000000
10010001	10000000
11111111	11110000
11111111	00000000

正如读者在译码器的电路图（图7-12）可以看到的，不能对地址位 A_{14}～A_4译码。所以，可以把它们假设是任何数；当地址位 A_{15} 等于逻辑 1 时，该译码器将仍然对地址位 A_0～A_3 译码。

利用$\overline{\text{MEMW}}$或$\overline{\text{MEMR}}$信号来启动该译码器（图 7-13），

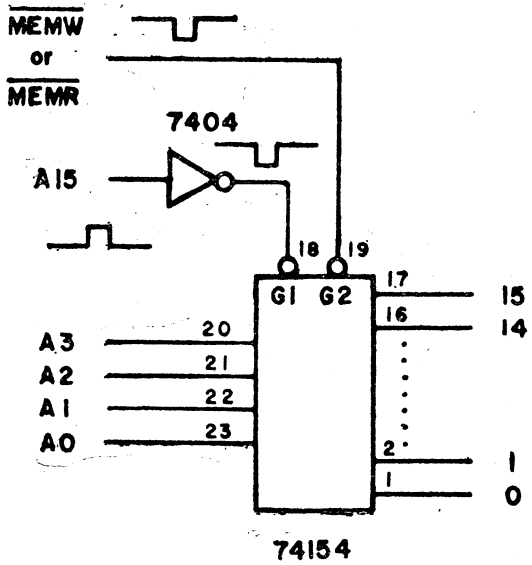


图 7-13　用$\overline{\text{MEMW}}$或$\overline{\text{MEMR}}$信号选通 SN 74154 译码器

这样可以把存储器映象输入/输出接口的译码器部分简化。在大多数 8080 微型计算机系统中，应该需要两个这样的译码器，一个用来启动器件，8080 从中读出数据，一个用来启动器件，8080可以向它们写入数据。用 8080 写入数据的器件把$\overline{\text{MEMW}}$信号接到译码器，用 8080 从译码器读出的器件把$\overline{\text{MEMR}}$接到译码器的这两个信号不仅能够用于 8080 和存储器映象输入/输出的设备之间数据传送，而且，它们也能够作为控制信号。通常，$\overline{\text{MEMW}}$ 信号启动的译码器的输出可以供这种用途使用。对于 8080 怎样与存储器映象输入/输出的设备通信，其程序实例，有些将可以采用本章开头部分所用过的一些程序实例。

带有硬件编码器的存储器映象输入/输出键盘

读者应该记得，这个接口（表 7-3）的 D_7 位是这种键盘的状态位。当 D_7位等于逻辑 1 时，这个字的四位最低有效位包含了被按下的这只键的键代码。如果D_7位等于逻辑 0，则没有键被按下。如果把该键盘作为存储器映象输入/输出的设备与 8080 连接，那么，例 7-23 所列出的任何一个子程序都可以用来等待某只键闭合，并且读这个键代码。

例 7-23 等待，然后读键代码的三种方法（存储器映象 I/O）

```
KEYIN, PUSHH      KEYIN, PUSHB      KEYIN, PUSHD
       LXIH              LXIB              LXID
       000               000               000
       200               200               200
```

```
KEY1,  MOVAM    KEY1,  LDAXB    KEY1,  LDAXD
       ADI             ADI             ADI
       200             200             200
       JNC             JNC             JNC
       KEY1            KEY1            KEY1
       0               0               0
       ANI             ANI             ANI
       017             017             017
       POPH            POPB            POPD
       RET             RET             RET
```

这三个子程序确实很相似。唯一的不同是用哪个寄存器来存贮存储器映象输入/输出的设备的地址。如果以用寄存器对H的这个子程序为例子，那么，当8080调入KEYIN这个子程序时，它首先把寄存器对H的内容保存在堆栈上。这一操作完成之后，8080把该输入/输出设备的16位地址装入寄存器对H。这个地址是什么呢？键盘的这个地址是200000(8000)。8080把这个地址装入寄存器对H以后，把键盘的这个8位字读入A寄存器。当8080执行这条MOVAM指令时，把存储在寄存器对H的这个地址送到地址总线上，并且，把脉冲加到$\overline{\text{MEMR}}$信号线。因为这个地址是键盘的地址，所以，把键盘的这个数据送到数据总线，并且选通，进入A寄存器里。正如不能同时把两个累加器输入/输出设备的数据放在数据总线上一样，不能同时把两个存储器映象输入/输出设备的数据放在数据总线上。所以，存储器（读/写或只读存储器）不能有写存储器映象输入/输出设备相同的地址。

当键盘的数据在A寄存器里时，8080把200（80）加到A寄存器的内容上。由于该加法操作的结果，其进位是逻辑0时，那么，D_7位是逻辑1，这就是说，有一个键被按下了。如

果由于该加操作的结果，进位是逻辑 1，那么，D_7位仍旧是逻辑 0，则没有键被按下。因此，如果没有键被按下，那么，8080 则执行 JNC-KEY1 这段指令。如果有一只键被按下，8080 则执行 ANI 指令，屏蔽这四位最高有效位。然后，8080 从堆栈弹出寄存器对H的内容，随后，8080 把这个四位键代码存入A寄存器里，从该子程序返回。

读者可以看到，完成同样功能，该存储器映象输入/输出的键盘的子程序和用累加器输入/输出的子程序比较，前者长。例 7-24 这个程序包含两个等效的子程序，一个用于存储器映象输入/输出，另一个用于累加器输入/输出。其他硬件编码器键盘改变成存储器映象输入/输出的例子这里没有涉及和讨论。

存储器映象输入/输出的、多路转换（扫描的）键盘

多路转换(扫描的)键盘(例 7-13) 从累加器输入/输出转换成存贮器映象输入/输出，所要求做出的变动是很少的。

在这个程序(例 7-25)中，8080 把寄存器对H的内容压入堆栈，然后，把键盘的 16 位地址装入寄存器对H里。把写到键盘的第一个测试字(376，FE)装入寄存器对 B，把可以被测试的键的键代码(003，03) 也装入寄存器对 B。用B寄存器存储这个测试字，用C寄存器存储这个键代码。在前面的扫描键盘程序例子中，应用了 B 和D这两个寄存器，所以，不能应用 LXIB 指令。

在 NXTGRP 上的第一条指令被用来把 B 寄存器所存储的测试字输出到键盘的接口电子器件。8080 只能执行这条指令，

因为键的 16 位地址被存放在寄存器对H里。然后，把B寄存器的内容送到A寄存器，并循环左移一位，再存回到B寄存器。这样做，正是为了产生下一个测试字——如果需要的话，它输出给键盘。然后，8080 执行一条 MOVAM 指令，从键盘读出一个四位的数据字，送入A寄存器。8080 把A寄存器的四位最高有效位屏蔽掉，并且把该结果与 017(0 F) 进行比较。正在被测试的一纵行键，如果没有键被按下，那么，A寄存器存储的内容是 017(0 F)。如果这一纵行中有一只键被按下了，那么，8080 将必须确定是哪一只键被按下了。

例 7-24 累加器输入/输出子程序和存储器映象输入/输出子程序的比较

存储器映象 I/O

```
/这个子程序用来等待 A 寄存器的 D7 位
/置逻辑 1。这是通过 200(十六进制 80)
/与这个端口数据相加来检测的。
/当标识位是逻辑 1 时，进位等于逻辑 1，
/四位的键代码仍旧存储在 A 寄存器里!
/这个键盘与 8080 微型计算机连接，作为
/存储器映象输入/输出设备。
KEYIN,  PUSHH   /把寄存器对 H 的内容保存在堆栈。
        LXIH    /把输入/输出设备的 16 位
        000     /地址装入寄存器对 H。
        200     /
KEY 1,  MOVAM   /从键盘读一个字符，送到 A 寄存器里。
        ADI     /把 200(十六进制 80)与它相加。
        200     /如果这位进位等于 0，则标识位
        JNC     /是 0! 该进位是 0，所以标识位
        KEY 1   /也是 0。因此，继续等待标识位
```

```
        0           /变为逻辑 1。
        ANI         /用 017 (十六进制 0 F)屏蔽
        017         /四位最高有效位。
        POPH        /从堆栈弹出寄存器对 H 的内容。
        RET         /把这个键代码存入 A，返回。
```

累加器 I/O

```
/这个子程序用来等待 A 寄存器的 D7 位
/成为逻辑 1。这是通过把 200(十六进制 80)
/加到这个输入端口数据上而
/检测的。当状态位等于逻辑 1 时，
/进位等于逻辑 1，四位的键代码仍旧
/存储在 A 寄存器!
KEYIN,  IN      /输入一个 8 位字，该字
        000     /包括状态位和数据位。
        ADI     /把 200(十六进制 80)与这个字相加。
        200     /如果进位等于 0，那么，标识位
        JNC     /等于 0! 进位是 0，所以，标识
        KEYIN   /位等于 0。所以，继续等待标识
        0       /位成为逻辑 1。
        ANI     /用 017(十六进制 0 F)屏蔽掉
        017     /四位最高有效位，
        RET     /把这个键代码存入A寄存器，
                /然后返回。
```

例 7-25 所列出的这个程序的大部分指令与例 7-13 的程序指令相同。但是，不同的是：因为寄存器对 H 的内容在子程序的开始被压入堆栈里，所以 8080 不能执行正在 NXTKEY 之后的那条 RNC 指令。然而，如果不把被按下去的这只键的逻辑 0 循环移入进位，那么，8080 则要执行 JC-ADDIT 这段指令。当 8080 把逻辑 0 移入进位时，则从堆栈弹出寄存器对

H的内容；然后，8080把被按下去的这只键的键代码存入C寄存器，从这个子程序返回。在前面的累加器输入/输出的、扫描的键盘子程序，用了D寄存器存储被按下的一只键的键代码。存储器映象输入/输出键盘扫描的子程序和给累加器输入/输出键盘扫描的子程序比较，前者所需要的存储单元比后者多五个。

例 7-25 存储器映象输入/输出的、4×4矩阵的键盘扫描子程序。

```
/这个程序用来对矩阵结构的键盘
/扫描，该键盘有四行，每行
/有四只键，即4×4的矩阵。采用
/存储器映象输入/输出技术，把该键
/盘与8080连接。
KEYSCN,  PUSHH    /把寄存器对H的内容保存在堆栈。
         LXIH     /把分配给键盘的16位
         005      /地址装入寄存器对H。
         200
         LXIB     /把376 003 (FE 03)装入寄存
         003      /器对B。B存储测试字，
         376      /C寄存器存储第一只键的键代码。
NXTGRP,  MOVMB    /把测试字送给键盘。
         MOVAB    /然后，把它送到A寄存器。
         RLC      /把该字循环左移1位。
         MOVBA    /然后把它保存在B寄存器里。
         MOVAM    /把键盘的输出读入到A寄存器里。
         ANI      /只存储四位最低有效位，
         017      /它包含排数据。
         CPI      /把017(0F)与输入字比较，
         017      /看是否有一只键被按
```

```
        JNZ     /下。这一行的一只键被按下，所
        NXTKEY  /以确定它是哪一只键。
        0
        DCRC    /被测试的这一排键中没有键被按下，
        MOVAC   /所以，键代码减 1。
        CPI     /检查四排是否已经都被测试完毕。
        377     /377＝十六进制 FF。
        JNZ     /四排键尚未全部测试完毕，
        NXTGRP  /所以，测试另一排。
        0
        JMP     /四排键都被测试完了。
        KEYSCN  /没有键被按下，
        0       /所以，继续扫描。
NXTKEY, RRC     /把排数据循环移入进位。
        JC      /进位等于逻辑 1，所以
        ADDIT   /试另一只键。
        0
        POPH    /进位等于逻辑 0，从堆栈
        RET     /弹出寄存器对H的内容，然后返
ADDIT,  PUSHPSW /回。否则，把 PSW 保存在堆栈
        MOVAC   /，使C寄存器的键代码增 4。
        ADI
        004
        MOVCA   /把这个新键代码保存在C寄存器
        POPPSW  /里。从堆栈弹出 PSW，
        JMP     /然后再试，
        NXTKEY  /直到进位为 0。
        0
```

存储器映象输入/输出发光二极管显示器

输出数据给一对七段发光二极管(LED)显示器的基本程序之一，使显示的数每隔$\frac{3}{4}$秒加1。一旦把接口电子器件从累加器输入/输出技术改变为存储器映象输入/输出时，可以用存储器访问指令把改变的数传送给显示器。

例7-26所列出的这个程序把两位数字的存储器映象输入/输出的LED显示器的16位存储地址装入寄存器对H。然后，8080把B寄存器清零，因为B寄存器要用来存储被显示的数。在DISPLY处，8080把B寄存器的内容写到LED显示器，然后又把它送到A寄存器。把1加到A寄存器的内容上；8080执行这条DAA指令，对该结果进行调整。然后，把这一结果存回B寄存器。此后，8080调入DELAY延时子程序。大约经$\frac{3}{4}$秒之后，8080从该DELAY子程序返回。8080再返回到DISPLY，把增量的数输出给显示器。

在其它显示程序例之一中，8080把五个连续存储单元的被压缩的BCD内容输出到十个七段显示器。每个显示器备有一个锁存器SN 7475，一个译码器SN 7447。如果必须把这同一个数据输出给存储器映象输入/输出显示器，那么，8080将必须有两个存储器指示器；一个用来把数据存入存储器用，另一个供不同的七段显示器用。

例7-27所列出的这个子程序首先把LED显示器的输入/输出地址装入寄存器对B。把存储在存储器的数据的地址装入

寄存器对H。然后，8080从存储器取一个八位数，送到A寄存器；寄存器对H的内容作为存储器地址使用。接着8080执行这条STAXB指令，把A寄存器的内容写出到存储器映象输入/输出的LED显示器。然后把这两个存储地址加1，并且把下一个数据字从存储器送到显示器。读者可以猜想得到，如果利用一个循环，可以大大简化这个子程序。

在例7-28中，我们把存储器映象输入/输出地址装入寄存器对B里，把存储数据(数)的存储器地址装入寄存器对H。然后，把005(05)装入D寄存器，因为五个存储单元供存储数据用，有五对发光二极管显示器。在DLOOP处，8080从存储器读一个数据字，送入A寄存器里，然后输出给发光二极管显示器。然后，8080执行INXH，INXB指令，把存储地址加1，而把D寄存器存储的字数减1。如果这个数不等于零，8080执行JNZ-DLOOP这段指令，把另一个字从存储器传送给发光二极管显示器，当把五个字都输出给显示器以后，8080从DISPLY子程序返回。

这个子程序体现了存储器映象输入/输出技术的另一个优点。它可以使用循环把数据写到有连续地址的存储器映象输入/输出的设备，或者从其中读出数据。如果输入/输出设备是采用累加器输入/输出技术的设备，那么，这是不可能的，因为这些设备地址实际上是指令的一部分。所以，不能使用循环对有连续地址的累加器输入/输出设备进行存取。但是，8080微型计算机的大多数系统没有大量的输入/输出设备，需要用这种方式来进行存取。因此，这一优点常常并非重要。

例7-26 发光二极管显示器(存储器映象输入/输出设备)的计数器程序

```
COUNT ,   LXIH      /把一个16位存储地址装入
```

```
          125       /寄存器对 H，该地址是分配
          200       /给两位数字的显示器（8055）。
          MVIB      /把 000（十六进制 00）装入 B 寄存器，
          000       /B 寄存器供暂存用。
DISPLY,   MOVMB     /把 B 的内容写到显示器。
          MOVAB     /把数据字送到 A。
          ADI       /把 1 加到 A 的内容上。
          001
          DAA       /对该结果进行十进制调整，
          MOVBA     /然后把结果保存在 B 寄存器里。
          CALL      /调用 DELAY 子程序，
          DELAY     /它大约需要 3/4 秒的
          0         /执行时间。
          JMP       /然后，返回到该程序
          DISPLY    /的显示程序段。
          0
DELAY,    LXID      /把 000 000（十六进制 0000）
          000       /装入寄存器对 D
          000
WAIT,     DCXD      /寄存器对 D 的内容减 1
          MOVAD     /把最高有效字节送到 A。
          ORAE      /把最低有效字节与它相“或”。
          JNZ       /如果其结果不等于 0，则
          WAIT      /返回到 WAIT。
          0
          RET       /否则，返回。
```

例 7-27 十位数字的发光二极管显示器（存储器映象输入/输出设备）的子程序

```
/这个子程序用来显示五个连续存储
/单元的内容，每个存储单元用来存储
/两位被压缩的 BCD 数字。
/发光二极管显示器与 8080 连接。
/作为存储器映象输入/输出设备。
DISPLY, LXIB     /把分配给显示器的第一个地址装
        125      /入寄存器对 B，
        200      /200 125，十六进制 8055。
        LXIH     /把存储地址装入寄存器对 H，
        120      /这里，存储着要被
        004      /显示的一些数。
        MOVAM    /从存储器取一个字到 A。
        STAXB    /把该字送给显示器。
        INXB     /I/O 设备地址加 1。
        INXH     /存储地址加 1。
        MOVAM    /从存储器取一个字到 A。
        STAXB    /把该字输出给显示器。
        INXB     /把该输入/输出设备地址加 1 。
        INXH     /存储地址加 1 。
        MOVAM    /从存储器取一个字到 A 寄存器。
        STAXB    /把这个字输出给显示器。
        RET      /从这个子程序返回。
```

例 7-28 十个数字的发光二极管显示器（存储器映象输入/输出设备）的循环程序

```
/这个程序应用了一个循环，
/把存储器的内容输出给十位数
/字的显示器。这个显示器和 8080
/微型计算机连接，作为存储器
/映象输入/输出设备。
```

```
DISPLY, LXIB    /把分配给显示器的第一个地址
        125     /装入寄存器对B，
        200     /200 125＝十六进制 8055。
        LXIH    /然后，把用来存储被显示
        120     /的数的存储地址
        004     /装入寄存器对H(004 120，十六进制
        MVID    /0450)。把字的数目装入D寄存器，
        005     /005＝05。
DLOOP,  MOVAM   /把数从存储器送到A寄存器。
        STAXB   /然后把这个数写到显示器上。
        INXH    /存储地址加 1。
        INXB    /I/O 设备地址加 1。
        DCRD    /字数减 1。
        JNZ     /如果这个字数不等于 0，则执行
        DLOOP   /JNZ—DLOOP 指令，
        0       /把另一个字输出，写到显示器上。
        RET     /当字数等于 0 时，则返回。
```

十位数字的多路转换显示器

读者从前面的多路转换的显示器的程序例子知道，当数据字从存储器里取出以后，这些程序的大多数指令都涉及到这种数据字操作的。事实上，在 DISPLA 整个程序（例 7-20，7-21 和 7-22）中，只有一条输出指令 OUT。这条输出指令编排在 OUTIT 程序段。这就是说，DISPLA 这个程序不需作什么修改，就可以把这些数据字输出给存储器映象输入/输出设备——十位数字的多路转换显示器。

前面那个程序（例 7-21）所需要的修改只有一处，只要修

改这一处，就可以使用存储器访问指令来把数据字/数字使能字输出，到显示器接口。这一处修改只涉及到OUTIT程序段，即只要将一条STA指令代替OUT指令就行了。程序的其余部分是相同的。请读者注意，这个程序例子是从右到左（最低有效数字——最高有效数字）显示数字的，以增强所显示的数字的亮度。

例 7-29 采用存储器映象输入/输出、多路转换技术，十个数字的七段显示器亮度增强型程序

```
/这个程序用来驱动十个数字的多路转换
/七段发光二极管显示器。
/另外，为了使每个显示的数字的亮度
/相同，还增添了一些指令。
/这个显示器与 8080 连接，作为
/存储器映象输入/输出设备。
DISPLA,   LXIH     /把一个存储器地址装入寄存器对H，
          120      /该地址用来存储 BCD 数字。
          004      /004 120，十六进制 0450。
          MVID     /把即将启动的第一
          000      /个数字装入D寄存器。
DISPL1,   CALL     /显示头两位被压缩
          DIGIT    /的 BCD 数字。
          0
          INXH     /这个存储地址加 1。
          MOVAD    /取数字选通字到A。
          CPI      /把它与第十一个选通数字
          012      /进行比较，
          JNZ      /十个数字还没有全部被显示完，
          DISPL1   /所以显示另外两个数字。
          0
```

```
        JMP     /十个数字已经全部被显示完了，
        DISPLA  /所以，重复上述显示过程。
        0
DIGIT,  MOVAM   /取被压缩的 BCD 字，
        RLC     /送到A寄存器。把四
        RLC     /位最低有效位循环移
        RLC     /入四位最高有效位。
        RLC
        CALL    /然后显示这个数字。
        OUTIT
        0
        MOVAM   /再取同一个字，送到A寄存器。
OUTIT,  ANI     /只保存这个字的四位最高有效位。
        360     /(360，十六进制 F 0)。
        ADDD    /加这位选通数字。
        STA     /把这个选通数字的代码
        125     /和 BCD 数输出给
        200     /多路转换显示器（200 125，十六进
        INRD    /制 8055)。使D的选通数字加 1。
INTENS, MVIE    /把这个数装入E寄存器。
        100     /100，十六进制 40。
INTEN1, DCRE    /使这个数减 1。
        JNZ     /如果该数不等于 0，则
        INTEN1  /执行 JNZ 指令，返回
        0       /到 INTEN1。
        RET     /E寄存器的内容＝0，
                /则返回。
```

小　结

正如读者已经看到，有许多指令，可以用来与存储器映象输入/输出设备进行通信，也可以用来与累加器输入/输出设备进行通信，但是，用于前者的指令更多一些。事实上，我们可以用两条存储器访问指令，在 8080 和存储器，以及和存储器映象输入/输出设备之间传送 16 位的字。这两条指令是什么呢？是 SHLD 和LHLD。但是，存储器映象输入/输出设备的接口电子器件可能是比较复杂的，特别是器件地址译码器更为复杂。实际上，是选择这种方法，还是选择另一方法，应该视哪一种方法最容易完成接口的任务而定。接口的成本和数据通过接口传送的速度以及驱动该接口所需要的软件的成本都必须予以通盘考虑。时值今天，大多数微型计算机生产厂为 8080 提供输入/输出设备，供 8080 用作累加器输入/输出操作的设备。也有一些生产厂提供输入/输出设备，这些设备可以和 8080 连接，或者作为累加器输入/输出设备或者作为存储器映象输入/输出设备。根据我们的意见，你将会发现：把大多数外部设备和以 8080 为基础的微型计算机连接时，累加器输入/输出是简单的，而且是有效的。在应用某些微型计算机时，你对于输入/输出技术没有选择的余地。例如，在 6800 为基础的和 6502 为基础的微型计算机系统中，只有存储器映象输入/输出技术是适用的。