6809

汇编语言程序设计

上　册

Lance A. Leventhal

机械工业部重庆工业自动化仪表研究所

一九八二年十月

出 版 说 明

许多微计算机户程序都使用汇编语言编制程序，大多数工业上微计算机用程序都需要用汇编语言来提供各种控制功能。几乎所有的微计算机设计者都必须具有一些汇编语言知识。了解汇编语言有助于各种高级语言的分析和评价。本书详尽地介绍了汇编程序和汇编程序设计，列举了在实际应用中的各种典型实例，并且附有代表性的习题。内容丰富具体，深入浅出，通俗易懂。对于从事微计算机应用的工程技术人员是一门很有价值的参考书，也是微型机学习班的好教材。

该书原文为《6809 Assembly Language Programming》，作者是美国Lance A. Leventhal博士，(1981年出版的最新板本)。我们组织了从事这方面实际工作、并有一定实践经验的工程技术人员翻译了这本书。原书共22章和一个附录，译本分上下两册出版，上册包括第一章至第十五章；下册从第十六章至第二十二章和一个附录。

本书由下列工程师分章翻译：郑宗汉（第一章至三章），余奔（第四章至第九章），杜芝君（第十章至第十五章），贾永乐（第十六章至第二十章），胡纯阳（第二十二章和附录），校对郑宗汉，编辑李连雨。由于时间短促，加之我们的水平不高，难免在译文中有不妥之处，望读者多加批评指导。

机械工业部重庆工业自动化仪表研究所

1982.4.

关 于 作 者

Lance A. Leventhal是Emulative System公司的合股人，后者是在San Diego建立的，专门以微处理机和微程序设计的咨询商行。他是IEEE在微处理机上的国立讲师，写过十本书，在微处理机上有60篇以上的论文，是Simulation以及Microcomputing这样一些刊物的定期撰稿人。也是Society for Computer Simulation的技术编辑，Digital Design杂志的稿件编辑。

Leventhal博士以前在这方面写过四本书，现在又投入了一个新的工作：Some Common Asscmbly Language Programs。他在圣路易斯被华盛顿大学授与学士学位。在San Diego被加利福利亚大学授与硕士和博士学位，他是SCS、ACM、IEEE以及IEEE计算机团体的成员之一。

序

Burrough公司的Irvin Stafford先生，为本书组配了一个以6809为基础的计算机，试验了本书里所有的例子，校正了很多错误，提出了很多改进。Osborne/McGraw-Hill的Curt Ingraham先生、Susanna Jacobson女士、Denise Penrose女士以及Janice Enger女士，对这个工作也作出了很大的贡献。Susanna Jacobson及Curt Ingraham坚决主张问题的叙述应该高度的清楚和准确。Motorola技术信息中心（Phoenix）的Lothar Stern先生和Marsball Rothen先生，慷慨地提供了材料，Sorrento Valley协会的Marielle Carter女士、Romeo Favreau先生及Gary Hankins先生也给了很多的帮助，我的妻子Donna，在她的允许和理解下，给了我很大的支持。

特别要感谢Motorola（奥斯汀、得克萨斯）的Terry Ritter先生，他是6809微处理机的原设计者，他相当友爱地审查了手稿。也要感谢得克萨斯大学的Jack Lipovsxi博士，他在奥斯汀为我提供了他所编写的，以6809为基础的Microcomputer Interfacing: Principles and Practices (Lexington Books, Lexington, Mass., 1980) 的初版本，这是一本极好的书，我从它里面的思想中，得到了大量的借览。

加拿大温尼柏的SDS技术服务有限公司的Allan Robbins先生，为本书第三章、第二十二章以及附录，提供了材料。

我应该借此机会感谢事先审查过这本书的那些人，特别是应该提到Mark Bernstein, Jim Butterfield Art Childs、James Demas及Philip Hooper。当然，对他们的否定意见，我原来的反应是采用守势的。但是，我的编者埋怨我，在他的推动之下，我在这本书里，就反映出了对他们的评论。我合理地修订了几章，并且着重清楚、简洁的说明，以及有趣的例子。回顾是一种徒劳的工作，因此我希望这些人了解到，我是学习了他们的成果的。

这本书献给在加利福利亚Solana Beach南方大路101一起工作的朋友们：Don及Hazel Cahoon，Lou及Marge DiCarlo、Bob及June Vallery。

—Lance A. Leventhal

目　　录

第一部分 基 本 概 念

这本书叙述汇编语言的程序设计。本书假定读者熟悉微计算机引论第一卷——基本概念（Berkely：Osborne/McGrow—Hill、1980）。特别是这本书的第六章和第七章关系更大。这本书不讨论计算机、微计算机、编址方式、或指令集的一般特性；读者可以参考微计算机引论第一卷的有关资料。

这一部份共有三章，它提供了汇编语言的一般基本知识，特别是6809汇编语言的基本知识。第一章讨论汇编语言的目的，以及它和高级计算机语言的比较。第二章讨论汇编程序，并简单地讨论装入程序。第三章叙述微计算机6809的结构，它和类似处理机的比较，并讨论Motorola 6809汇编程序的重要特性。

第一章 汇编语言程序设计引言

计算机程序毕竟是一系列数字，因此，对人们的意义不大。在这一章里，我们将讨论这样的计算机程序，这种计算机程序，可以用类似于人们这一级别的语言来表达。此外，我们将进一步地讨论汇编语言的使用及其理由，这就是这本书的内容。

指令的意义

微处理机的指令集是一组二进制的输入信号，它在一个指令周期里，产生所规定的动作。对微处理机来说，指令集是对门、地址、或移位寄存器这样一些逻辑设备的一张操作表。当然，在响应它的指令的输入信号时，微处理机所执行的动作，比逻辑设备响应这些输入信号所执行的动作，远为复杂得多。

二进制指令

指令是一种二进制数字形式——它必须能够以数据输入给微处理器，以便在适当时间解释为指令。例如，当6809微处理机接受形式为01001111的8位二进制数字，作为在取指令操作时的输入信号，这种形式意味着：

“清（把零放在）累加器A”

类似地，形式为10000110意味着：

“把数取到累加器A”

微处理机（类似于任何其他计算机）只识别二进制形式作为指令或数据，它不能识别字，或八进制、十进制、或十六进制数。

计 算 机 程 序

程序是一指令序列，它使计算机去执行特定的任务。

实际上，计算机程序包括很多指令；它也包含数据和存贮器地址，这些地址，是微处理机为了完成指令所规定的任务而需要的。很清楚，如果微处理机要执行加法，它就必须有二个数去进行相加，并且，有一个地方去存放结果。除了执行操作外，计算机程序还必须决定数据来源，以及存放结果的目的地址。

所有的微处理机都是顺序地执行指令的，除非指令改变了执行的顺序，或停止了处理机的执行。即处理机从下一个存贮地址去取得下一条指令，除非当前指令规定它直接去执行别的东西。

最后，每一个程序都是一组二进制数。例如，下面是一个 6809 的 程 序，它把存贮单元0060_{16}和0061_{16}的内容相加，并把结果存放在存贮单元0062_{16}里：

```
10110110
00000000
01100000
10111011
00000000
01100001
10110111
00000000
01100010
```

这是机器语言程序，或目标程序。如果把这个程序放进6809微计算机的存贮器里，微计算机就能够直接地执行它。

二进制程序设计问题

按照目标程序、或机器语言程序那样产生的程序，将引起很多困难。下面就是其中的一些问题：

1. 程序难于理解和调整（特别是你已经检查过它们之后，过几个小时再去看这些二进制数，它们看起来又都是千篇一律的，不清楚它们的意义是什么了）。
2. 把程序放进计算机去很慢，因为必须对每一位都设置一个面板开关，并且，得一次装入存贮器一个字节。
3. 程序不能以类似人们可阅读的任何形式，来叙述人们希望计算机去执行的任务。
4. 程序很长，写起来令人讨厌。
5. 程序设计者经常发生疏忽错误，很难于找出和修改这些错误。

例如，下面这种形式的加法程序，有一位包含着错误。试试把它找出来：

```
10110110
00000000
01100000
10111011
00000000
```

```
01110001
10110111
00000000
01100010
```

尽管计算机可以很容易地处理二进制数，但人们不行。人们发现，二进制程序很长、很混乱，并且意义不明显。最后，程序设计者虽然可以着手记住一些二进制代码，但是，这种努力，要花费大量的精力。

使用八进制或十六进制

用八进制或十六进制来书写指令，而不用二进制来书写，我们就可以稍微地改善一下这种情况。在这本书里，我们将使用十六进制数，因为它们较为简短些，也因为它们对微处理机行业来说，也是标准的。表 1-1 说明了十六进制数字以及它们等价的二进制数，现在，对二个数相加的6809程序，用十六进制表示，就变成：

```
B6
00
60
BB
00
61
B7
00
62
```

表1—1　　十 六 进 制 转 换 表

十六进制数字	等价的二进制数	等价的十进制数	十六进制数字	等价的二进制数	等价的十进制数
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	A	1010	10
3	0011	3	B	1011	11
4	0100	4	C	1100	12
5	0101	5	D	1101	13
6	0110	6	E	1110	14
7	0111	7	F	1111	15

至少，用十六进制形式书写较为简单，检查起来也不会太令人厌倦。

在十六进制数字序列中，就稍微容易发现错误。上面的那个错误形式的加法程序，用十六进制表示，就变成：

```
B6
00
60
BB
00
71
B7
```

00
62

其错误就远为明显得多。

用十六进制编制程序，我们要做些什么呢？微处理机只能理解二进制的指令代码。如果我们的面板上有十六进制键盘，而不是位开关，我们就可以直接用键把十六进制程序放进存贮器——键盘把十六进制数字转换成二进制数。但是，如果我们的面板上只有位开关呢？我们可以由自己把十六进制数字转换为二进制数字，但这是一种重复的，令人厌倦的任务。人们企图去进行这种转换，将产生各种微小的错误。例如，看到另外错误的一行去，丢掉了一位，对调了一位，或一个数字。此外，我们一旦把我们的十六进制程序转换好了之后，我们仍然必须通过面板上的开关，把每一位都放进存贮器去。

十六进制装入程序

可是，这些重复的、使人精疲力尽的任务，对计算机来说，却是很熟练的工作。计算机决不会感到厌倦或烦恼，也决不会糊里糊涂地产生错误。于是，可以想像，编写出一个程序来，这个程序接受十六进制数字，把它们转换为二进制数，并把它们放进存贮器去。这就是在很多微计算机里所提供的标准程序，它叫做十六进制装入程序。

十六进制装入程序类似于其他任何程序。它要占有存贮空间。在一些系统里，它保存在存贮器里，长度正好足以装入另外的程序；在另外一些系统里，它占有一个保留区，一部份只读存贮器。在微计算机面板上，可以没有位开关，甚至可以不要面板。这反映了机器设计者的决定。二进制程序设计，不仅仅是冗长乏味的，而且也完全没有必要。在我们的系统里，十六进制装入程序，可以是一个大程序的一部份，这个大程序叫做监控程序，监控程序也提供了很多调整和分析程序的工具。

十六进制装入程序，当然不能解决程序设计的所有问题。十六进制形式的程序，仍然难于阅读和理解。例如，它不能把操作码和地址、或数据区别开来，也不能列出程序表格，以提供任何暗示，表明这个程序是做什么的。B6或3F当成什么解释？记住卡片上的全部代码，并不是一种诱人的建议。此外，对不同的微处理机来说，代码也将是完全不同的，并且，程序将要求有大量的文挡。

指令代码记忆符

给每个指令代码分配一个名字，就明显地改善了程序设计。指令代码的名字叫做“记忆符”，或记忆提醒符。指令记忆符以极少量的字符，来叙述这一指令是做什么的。

记忆符的设计

实际上，所有的微处理机厂家（他们也不能记住十六进制代码）都为微处理机指令集提供了一套记忆符。可以不遵照厂家所提供的记忆符，它们并不是神圣不可侵犯的。可是，对给定的微处理机，它们是标准的，因此，为所有的用户所理解。有一些指令代码，可以在手册、卡片、书本、文章、以及程序上找到。随着指令记忆符的选择，出现了一个问题；并不是所有的指令，都有“明显”的名字。有些指令有（例如：ADD、AND、OR），另外一些是明显的缩约词（像减法的SUB，按位加的XOR），还有一些二者都不是。像WMP，PCHL，甚至SOB，这样的记忆符，就是这种情况。大部份厂家都提出了一些合理的名字，

而有一些则不可能。可是，用户设计自己的记忆符，很少能做得更好。

跟指令记忆符一起，制造厂家通常也给CPU寄存器分配一个名字，正如指令名字一样。有些寄存器的名字也是明显的（例如作为累加器的A），而另外一些，可能只有历史上的意义。再说一遍，我们将使用制造厂家的建议，以促使其标准化。

标准的记忆符

在这里提出一套标准的汇编语言记忆符。[1]还不能断定，它能得到大量的使用，但是，在对指令集进行比较时，它至少可以作为一个基础。并为将来的进一步处理，为记忆符提供一种选择。

汇编语言程序

如果我们使用标准的6809记忆符和寄存器记忆符，正如Motorola所定义的那样，我们的6809加法程序，就变成：

```
LDA    $0060
ADDA   $0061
STA    $0062
```

这个程序还不是很明显，但至少有一部份是可以理解的。ADDA比起BB来，有很大的改进。LDA和STA表示把内容装入累加器，或把累加器的内容送存。现在，我们看到，在这个程序里，有一些是表示操作的，而另外一些表示数据或地址。像这样的程序，就是汇编语言程序。

汇编程序

我们怎样把汇编语言程序取进计算机呢？我们要把它翻译成十六进制数，或二进制数。我们可以手工地，一条指令一条指令地翻译一个汇编语言程序。

表1-2阐明了这个加法程序的手工汇编。

表1—2

指令记忆符	编址方式	等价的十六进数
LDA	扩展直接	B_6
ADDA	扩展直接	BB
STA	扩展直接	B_7

正如十六进制转换为二进制一样，手工汇编是一种机械的任务，它是一种令人厌烦的、重复的工作，并且经常产生很多小错误。例如：挑选了错误的一行，数字对调了，忽略了指令，看错了代码，等等，仅仅是我们可能产生的少数错误之一。大多数微处理机，由于具有不同长度的指令，就使得这种任务甚至变得更加复杂了。有些指令是一字节长，而有一些指令，则可能有二个到五个字节长。有些指令，在第二和第三字节，要求的是数据，而有一些，要求的则是存贮地址、寄存器号，或者另外一些什么东西。

汇编，是我们可以分配给微计算机去完成的另一种机械的任务。在代码翻译时，微计算机决不会产生任何错误；它总知道，每条指令需要多少字节，有什么样的格式。完成这个工作的程序叫做“汇编程序”。汇编程序把用记忆符写成的用户程序，即“源程序”，翻译成

微计算机可以执行的机器语言程序，即目标程序。汇编程序的输入是源程序，它的输出是目标程序。

汇编程序是一个程序，正如十六进制装入程序一样。可是，汇编程序比十六进制装入程序价格更高，占用更多的存贮器，需要更多的外部设备，和执行时间。虽然，用户可以（并且经常）写出他自己的装入程序，而很少考虑要去编写他自己的汇编程序。

此外，汇编程序有其固有的法则，必须学会这些法则。这些法则包括：在适当的地方使用一些记号（例如间隔、逗号、分号、或冒号那样），正确的拼写法，适当的控制信息，甚至可能还包括名字和数字的正确安排。这些法则通常是很简单的，可以很快地学会。

汇编程序的附加特性

早期的汇编程序，只是把指令和寄存器的名字记忆符，翻译为它们等价的二进制数。可是，现在大部份汇编程序，都提供了这样一些附加特性：

- 允许用户给存贮单元、输入输出设备、甚至是指令序列，分配一个名字。
- 把数据或地址由各种数制系统（例如十进制或十六进制）转换为二进制，把字符转换为它们的ASCII或EBCDIC二进代码。
- 作为汇编处理的一部份，执行一些算术运算。
- 告诉装入程序，目标程序或数据应放在存贮器的那一部份。
- 允许用户分配一个存贮区域，作为临时数据存贮区，并把固定的数据，放进程序存贮区。
- 在当前程序里提供一些信息，以便把程序库里的标准程序、以及在另外某个时候所编写的程序包括进来。
- 允许用户选择程序清单的格式，以及所使用的输入输出设备。

汇编程序的选择

当然，所有这些特性，都牵涉到存贮器的额外费用。通常，微计算机的汇编程序，比起较大的计算机的汇编程序简单。但是，汇编程序的规模总是趋向于增大的。将经常要对汇编程序进行选择。重要的标准并不是汇编程序具有多少特性，而是在实际使用中的方便程度如何。

汇编语言的弱点

汇编程序，类似于十六进制装入程序，不能解决程序设计中的所有问题。一个问题是：微计算机的指令集，和微计算机要去执行的任务之间，还有很大的距离，计算机的指令，倾向于去执行类似如下的事情：二个寄存器的内容相加，累加器的内容移一位，或者是把一个新的值放到程序计数器去。在另一方面，用户通常希望，微计算机去完成类似如下的某些事情：检查模拟量输入是否超过某一限度，等待由电传打字机来的某一特定命令，并响应它，或在适当的时间里去触发一个继电器。汇编语言程序设计者把这样的任务，转换成一个简单的计算机指令序列。这种转换，可能是一种困难的，耗费时间的工作。

此外，如果我们用汇编语言进行程序设计，我们就必须具有我们所使用的，特定的微计算机的详细知识。必须知道，微计算机有什么样的寄存器和指令，指令是怎样影响各种寄存器的，计算机使用了什么样的编址方式，以及其他大量知识，这些知识没有一个和微计算机

最终必须执行的任务有关。

不可移植

此外，汇编语言程序是不能移植的。每一个微计算机都有它自己的汇编语言，它反映了它自己的结构。为6809编写的汇编语言程序，将不能在6502、Z80、8080、或3870微处理机上运行。例如，为8080编写的加法程序是：

```
LDA   60H
MOV   A, B
LDA   61H
ADD   B
STA   62H
```

不可移植，不仅意味着：我们的汇编语言程序，在不同的微计算机上是不能使用的，而且，任何其他程序，只要它们不是特地为我们所使用的微计算机而编写的，我们也不能使用这些程序。对微计算机来说，这是一个特殊的缺陷，因为这些微处理机是新的，为它们而编制的汇编语言程序并不多。结果，我们经常要编写出自己的程序。如果需要一个程序去执行特定的任务，在大多数厂家所提供的小程序库中，很可能找不到这样的程序。在挡案、杂志等文献里，或在某一个旧的程序文件中，也很可能找不到这种程序。多半得自己把它编写出来。

高　级　语　言

解决汇编语言程序有关的很多困难，是使用“高级”语言，或“面向过程”的语言。这种语言，允许我们以面向问题的，而不是面向计算机的形式，来叙述任务。在高级语言里的每一个语句，执行一个可辨认的功能；通常，它相当于很多汇编语言指令。一个被称为编译程序的程序，把高级语言源程序，翻译成目标代码，即机器语言指令。

高级语言FORTRAN

对不同类型的任务，存在着很多不同的高级语言。例如，如果希望计算机去做的一些事情，是可以用代数记号来表达的，就可以用FORTRAN(Formula Translation Language)语言来编写程序，它是最老的，使用最广泛的高级语言。现在，如果希望二个数相加，只要告诉计算机：

$$SUM=NUMB_1+NUMB_2$$

这就比等价的机器语言程序，或等价的汇编语言程序简单得多（也简短得多）。还有其他高级语言，包括COBOL（商业应用），PASCAL（结构程序设计语言），PL/1(FORTRAN和COBOL的结合），APL和ABSIC（分时系统用的），以及C（贝尔电话实验室研制的系统程序设计语言）。

高级语言的优点

很清楚，用高级语言编写程序，既容易又迅速。一般的评价是：程序设计者用高级语言编写一个程序，比用汇编语言编写，快十倍左右[2-4]。这只是程序的编写而已；它还不包括

问题定义、程序设计、调整、测试、或归挡，所有这些，都变得更简单和迅速。例如，高级语言程序，本身就是部份归挡的。至使不了解FORTRAN，也有可能讲出，上面所叙述的语句是做什么的。

机器独立性

高级语言解决了很多与汇编语言程序设计有关的问题。高级语言有它自己的语法（通常是由国家标准、或国际标准所定义的）。高级语言没有牵涉到指令集、寄存器、或特定计算机的其他特性。编译程序对所有这些细节进行加工。程序设计者可以把精力集中于他自己的任务，他们不需要详细地了解CPU的基本结构——就此而言，他们不需要了解计算机的任何有关事情。

可 移 植

用高级语言编写的程序，至少，在理论上是可移植的。它们可以在任何计算机上运行，只要这计算机有这个语言的标准的编译程序。

同时，当我们在新的计算机上进行程序设计时，以前用高级语言为早先的计算机编写的程序，现在都可以为我们所用。在类似于FORTRAN 或 BASIC 语言的情况下，这就可能意味着，我们可以用到数千个这样的程序。

高级语言的缺点

如果我们所讲的有关高级语言，所有的事情都这么好——如果可以迅速地编写程序，此外，这些程序又是可移植的——为什么又要操心汇编语言呢？谁希望为着寄存器、指令代码、记忆符、以及所有这些杂乱的东西而烦恼呢？正如通常那样，优点总被缺点所抵销。

语　　法

和汇编语言一样，一个明显的问题是：我们希望使用的任何高级语言，我们都应该遵照它的“法则”或“语法”。高级语言有一套相当复杂的法则。我们将发现，要花费很多时间，才能得到一个语法上正确的程序。（甚至在这种情况下，它也多半不能完成我们所希望的事情）高级计算机语言类似于一门外语。如果能够的话，就可以使用这些法则，并产生出编译程序可以接受的程序。可是，遵照这些法则，力图取得编译程序可以接受的程序，对完成我们的工作，并不能直接起到作用。

例如，下面是FORTRAN的一些法则：

- 标号必须是在卡片上前五列的数。
- 语句必须在第七列开始。
- 整数变量必须以字母I、J、K、L、M或N开始。

编译程序的成本

另一个明显的问题是：需要一个编译程序，它把用高级语言编写的程序，翻译成机器语言。翻译程序是很昂贵的，而且，要使用大量的存贮器。大部份汇编程序，只占据 2K 到16 K字节的存贮器（1K＝1024），而编译程序，却占据4K到64K。在使用编译程序时，所牵涉到的这种数量的开支，是相当大的。

与语言相适应的任务

此外，只有一些编译程序，能够较简单地实现我们的任务。例如，FORTRAN，很适合于可以表达成代数公式的课题。可是，如果我们的课题是控制打印机、编辑字符串、或监控模拟系统，就不能容易地用代数记号来表达我们的课题。在实际上，用代数记号来系统地叙述我们的这种问题的解，比用汇编语言更笨拙，更困难。当然，答案是使用更合适的高级语言。上面所提到的这样一些任务，专门为这些任务所设计的语言是存在的——它们叫做系统执行语言。可是，这些语言，使用范围比FORTRAN窄，标准化程度比FORTRAN差。

效率差

高级语言不能产生很有效的机器语言程序。其基本原因是：编译是自动处理的，它充满着折衷的解决办法，要考虑到范围很广的可能性。编译程序的工作，很类似于计算机化的语言翻译器——有时，翻译出来的单词，是正确而合理的，而句子构结却是笨拙的。例如，尽管并不需要长期地使用一个变量，且可以取消它，尽管可以使用寄存器，而不必使用存贮单元，或者，尽管变量之间具有简单的关系，而简单的编译程序是不知道的。有经验的程序设计者，可以方便地利用捷径，以缩短执行时间，或减少所使用的存贮器。少数编译程序（如优化的编译程序）也可以做到这一点，但是，这样的编译程序，比正规的编译程序大些。

优点和缺点概述

高级语言的优点：

- 较易于学会（以及教给别人）。
- 叙述任务较方便。
- 编写程序花费的时间较少。
- 较易于归档。
- 标准的语法。
- 与特定的计算机结构无关。
- 可移植。
- 库和其他程序可通用。

高级语言的缺点：

- 法则特殊。
- 要求有更多的硬件和软件的支持。
- 一般语言倾向于代数问题，或商业问题。
- 程序的效率低些。
- 优化的代码难于满足时间和存贮器的要求。
- 不可能方便地使用计算机的独特的特性。

微处理机的高级语言

在使用高级语言时，微处理机用户将遇到一系列特殊困难。其中有：

- 微处理机很少具备有高级语言。特别是对那些新的，比较不流行的处理机，或者是那些预期作为简单控制用的处理机，更是这样。

· 没有几个标准语言可以广泛使用。
· 编译程序通常需要大量的存贮单元，或者，甚至是需要一个完全不同的计算机。
· 大部份微处理机的应用，都不适宜于使用高级语言。
· 很多微处理机语言，并不产生目标程序。即，它们把程序翻译出来，并一行接着一行地运行——这是作为解析性处理，而不是作为编译性处理的——或者，它们产生的目标程序，要求有特殊的系统软件（运行时程序包）来执行。在执行这个程序时，要得到结果，可能是很慢的，并且要使用大量的存贮器。最通用的高级语言，如ABSIC，PASCAL，通常就是使用这些方法之一。
· 在应用微处理机时，存贮器的费用，经常是很关键的。

相对来说，微处理机只有少数几个高级语言，这是由于微处理机的历史还很短，并且，它们是来源于半导体工业，而不是计算机工业的。最经常使用的高级语言中，有BASIC[6]，PASCAL[6,7]，FORTRAN，C[8]，以及如PL/M那样的PL/1语言。

现存的很多高级语言，和已经得到认可的标准都不一致。以致微处理机用户，不可能期望得到可移植的程序，或者是利用程序库，或使用以前的经验或程序。剩下的主要优点，就只有减轻程序设计的工作，易于归挡，以及不需要那么详细地了解计算机结构。

高级语言的费用

在使用微处理机高级语言时，所牵涉到的费用是相当大的。直到最近，微处理机都是比较适合于控制，以及慢速的交互应用，而不适合于编译时所牵涉到的字符处理和语言分析。因此，一些微处理机的编译程序，将不能在以微处理机为基础的系统上运行。换句话说，它们需要比较大的计算机，即它们是交叉编译程序，而不是自编译程序。用户不仅必须承担大计算机的费用，而且也必须把程序由大计算机的，翻译成微型机的。

可以用到一些自编译程序。在微计算机上运行这些编译程序，为它们产生目标代码，遗憾的是，它们需要大量的存贮器（16K以上）加上专用的硬件或软件。

高级语言的不足

通常，高级语言不能很好地适合于微处理机的应用。最普遍的语言都设计成既有助于解决科学上的问题，又能处理大规模的商业数据。没有几个微处理机的运用适合于这二个领域。大多数微处理机的应用，都牵涉到对输出设备发送数据和控制信息，由输入设备接收数据和状态信息。控制信息和状态信息，经常由几位二进制数字组成，这些数字，具有很明确的有关硬件的意义。如果试图用高级语言，编写一个典型的控制程序，就可能感觉到，好像是用筷子喝汤一样。对于诸如测试装置、终端、导航系统、信号处理、以及商业装置，这样一些领域里的任务，使用高级语言处理，就比在仪器、通讯、外部设备、以及机动的应用中，使用高级语言要好些。

语言级的应用领域

需要大量存贮器的那一些应用，是比较适合于高级语言的。如果是像在阀门控制、电子游戏、器械控制、或小型的仪器中，单存贮片的成本很重要，那么，使用高级语言，存贮器效率很差，就无法容忍了。在另一方面，如果是像在终端、或测试装置中，无论如何，系统有千万个字节的存贮器，效率差就不是那么重要了。很清楚，程序的大小和应用的数量，也