第 3 章

汇编语言程序设计

一般来说，可以选择三种不同层次的计算机语言来编写程序，这就是机器语言、汇编语言和高级语言。

在机器语言(Machine Language)中，用二进制数表示指令和数据，它的缺点是不直观，不易理解和记忆，因此编写、阅读和修改机器语言程序都比较繁琐。但是，机器语言程序是计算机唯一能够直接理解和执行的程序，具有执行速度快，占用内存少等优点。

汇编语言(Assembly Language)弥补了机器语言的不足，它用助记符来书写指令，地址、数据也可用符号表示，与机器语言程序相比，编写、阅读和修改都比较方便，不易出错。它的执行速度和机器语言程序差不多。但计算机只能辨认和执行机器语言，因此，用汇编语言编写的源程序必须“翻译”成机器语言目标程序(或称目标代码)才能执行，这种翻译过程称为汇编(Assemble)。目前常常利用计算机配备的系统软件自动完成汇编工作，这种软件称为汇编程序(Assembler)。

一般来说，有两种汇编程序，一种通常称为小汇编(ASM)，另一种称为宏汇编(MASM)。后者的功能更强，但占用更多的内存容量。MASM 对 ASM 是兼容的。汇编语言源程序被汇编成为机器代码时，与 CPU 指令一一对应。另外，汇编语言和机器语言一样，都是面向具体机器的语言，也就是说，不同的 CPU 具有不同的汇编语言，互相之间不能通用。

高级语言(High Level Language)不针对某个具体的计算机，通用性强。用高级语言编程不需了解计算机内部的结构和原理，对于非计算机专业的人员比较容易掌握。高级语言程序易读、易编，相对比较简短，广泛应用于科学计算和事务处理中。常用的高级语言有 C 语言、BASIC、Fortran 等。但是高级语言编写的源程序同样必须“翻译”成为机器语言后，计算机才能执行，所用的系统软件称为编译程序或解释程序。通常，编译程序或解释程序比汇编程序复杂得多，需要占用更多的内存，编译或解释的过程也要花费更多的时间。

综上所述，三种语言各有利弊，究竟采用哪种，应视具体情况而定。随着计算机技术的发展，目前，直接使用机器语言编程的情况已不多见。在许多微型计算机系统中，尤其在一些对程序执行速度要求较高而内存容量又有限的场合，例如在某些实时系统中，常常采用汇编语言编程。虽然使用高级语言能够使软件开发的时间缩短，开发过程加快，但是高级语言不便于直接访问硬件，充分发挥硬件电路的性能；而且，高级语言通过编译以后得到的目标代码相对比较冗长。使用汇编语言在开始编程和调试阶段当然要花费较多的

时间,但是与等效的高级语言相比,执行速度快得多,占用内存也少得多。为了扬长避短,有时在一个程序中采用混合编程的方法,对执行速度或实时性要求较高的部分用汇编语言编写,而其余部分则可用高级语言编写。

3.1 汇编语言源程序的格式

在第2章介绍指令系统时,曾经列出若干程序举例,但是,这些只是局部的程序段,并不是完整的汇编语言源程序,下面举出一个比较简单,但比较完整的汇编语言源程序。例如,在例2.2中要求将两个5字节的十六进制数相加,可以编写出以下汇编语言源程序。

[**例3.1**] 汇编语言源程序的例子。

```
DATA     SEGMENT                          ;定义数据段
DATA1    DB 0F8H,60H,0ACH,74H,3BH         ;被加数
DATA2    DB 0C1H,36H,9EH,0D5H,20H         ;加数
DATA     ENDS                             ;数据段结束
CODE     SEGMENT                          ;定义代码段
         ASSUME CS: CODE,DS: DATA
START:   MOV AX,DATA
         MOV DS,AX                        ;初始化 DS
         MOV CX,5                         ;循环次数送 CX
         MOV SI,0                         ;置 SI 初值为零
         CLC                              ;清 CF 标志
LOOPER:  MOV AL,DATA2[SI]                 ;取一个字节加数
         ADC DATA1[SI],AL                 ;与被加数相加
         INC SI                           ;SI 加 1
         DEC CX                           ;CX 减 1
         JNZ LOOPER                       ;若不等于零,转 LOOPER
         MOV AH,4CH
         INT 21H                          ;返回 DOS
CODE     ENDS                             ;代码段结束
         END START                        ;源程序结束
```

在以上的汇编语言源程序举例中,在完成了5个字节的十六进制数加法之后,用以下两条指令结束:

```
MOV      AH,4CH
INT      21H
```

这两条指令表示一种功能号为4CH的DOS系统功能调用,它的作用是结束正在运行的程序,返回DOS。在汇编语言程序中,这是一种经常采用的结束程序的方式。关于DOS系统功能调用,将在本章3.4节进行介绍。

由上面的例子可以看出,汇编语言源程序的结构是分段结构的形式。一个汇编语言源程序由若干个段(Segment)组成,每个段以SEGMENT语句开始,以ENDS语句结束。

整个源程序的结尾是 END 语句。

这里所说的汇编语言源程序中的段与前面讨论过的 CPU 管理的存储器的段相比，既有联系，又在概念上有所区别。我们已经知道，微处理器对存储器的管理是分段的，因而，在汇编语言程序中也要求分段组织指令、数据和堆栈等，以便将源程序汇编成为目标程序后，可以分别装入存储器的相应段中。但是，以 8086 CPU 为例，它有 4 个段寄存器(DS、ES、SS 和 CS)，因此 CPU 对存储器按照 4 个物理段进行管理，即数据段、附加段、堆栈段和代码段。任何时候，8086 CPU 最多只能访问 4 个物理段。而在汇编语言源程序中，设置段的自由度比较大，例如一个源程序中可以有多个数据段或多个代码段等。一般来说，汇编语言源程序中段的数目可以根据实际需要而定。为了与 CPU 管理的存储器物理段相区别，我们将汇编语言源程序中的段称为逻辑段。在不致发生混淆的地方，有时简称为段。

在上面的简单源程序中只有两个逻辑段，一个逻辑段的名字是 DATA，其中存放着与程序有关的数据；另一个逻辑段的名字是 CODE，其中包含着程序的指令。每个段内均有若干行语句(Statement)，因此，总的来说，一个汇编语言源程序是由一行一行的语句组成的。下面先来讨论汇编语言语句的组成。

3.2　汇编语言语句的组成

汇编语言源程序中的语句主要有以下两种类型：

- 指令性语句。
- 指示性语句。

指令性语句主要由 CPU 指令组成；指示性语句又称伪操作语句，主要由伪操作指令组成。

一般情况下，汇编语言的语句可以有 1～4 个组成部分，如下所示：

```
[名字]  操作码/伪操作  [操作数]  [;注释]
```

其中带方括号的部分表示任选项，即可以有，也可以没有。例如，例 3.1 中有如下语句：

```
LOOPER:     MOV     AL,DATA2[SI]              ;取一个字节加数
DATA1       DB      0F8H,60H,0ACH,74H,3BH     ;被加数
```

以上第一条语句是指令性语句，其中的 MOV 是 CPU 指令的操作码助记符；第二条语句是指示性语句，其中的 DB 是伪操作。下面对汇编语言语句中的各个组成部分进行讨论。

3.2.1　名字

汇编语言语句的第一个组成部分是名字(Name)。在指令性语句中，这个名字可以是一个标号。指令性语句中的标号实质上是指令的符号地址。并非每条指令性语句都必须有标号，但如果一条指令前面有一个标号，则程序中其他地方就可以引用这个标号，例如可以转移到该标号处。在例 3.1 中，START、LOOPER 就是标号。标号后面通常有一个

冒号。

标号有三种属性：段、偏移量和类型。

标号的段属性是定义标号的程序段的段地址，当程序中引用一个标号时，该标号的段应在CS寄存器中。

标号的偏移量表示标号所在段的起始地址到定义该标号的地址之间的字节数。偏移量是一个16位的无符号数。

标号的类型有两种：NEAR和FAR。前一种标号可以在段内被引用，地址指针为两个字节；后一种标号可以在其他段中被引用，地址指针为4个字节。如果定义一个标号时后跟冒号，则汇编程序确认其类型为NEAR。

指示性语句中的名字可以是变量名、段名、过程名等。与指令性语句中的标号不同，这些指示性语句中的名字并不总是任选的，有些伪操作规定前面必须有名字，有些伪操作则不允许有名字，也有一些伪操作的名字是任选的。即不同的伪操作对于是否有名字有不同的规定。伪操作语句的名字后面通常不跟冒号，这是它和标号的一个明显区别。

很多情况下指示性语句中的名字是变量名，变量名代表存储器中一个数据区的名字，例如，例3.1中的DATA1、DATA2就是变量名。

变量也有三种属性：段、偏移量和类型。

变量的段属性是变量所代表的数据区所在段的段地址，由于数据区一般在存储器的数据段中，因此变量的段值常常放在DS或ES寄存器中。

变量的偏移量是该变量所在段的起始地址与变量的地址之间的字节数。

变量的类型有BYTE(字节)、WORD(字)、DWORD(双字)、QWORD(四字)和TBYTE(10个字节)等，表示数据区中存取操作对象的大小。

3.2.2 助记符和伪操作

汇编语言语句中的第二个组成部分是助记符(Memonic)或伪操作(Pseudo Operation)。

指令性语句中的第二部分是CPU指令的助记符，例如MOV，ADC等。CPU指令的助记符已在第2章进行了介绍。

例3.1的指示性语句中的DB、SEGMENT、ENDS、ASSUME、END等都是伪操作命令，而不是CPU指令的助记符。它们在程序中的作用是定义变量的类型、定义段以及命令汇编程序结束汇编等，这些操作都是由汇编程序完成的。关于伪操作的作用和使用方法，将在3.3节进行讨论。

3.2.3 操作数

汇编语言语句中的第三个组成部分是操作数(Operand)。对于CPU指令，可能有单操作数或双操作数，也可能无操作数，而伪操作可能有更多个操作数。当操作数不止一个时，互相之间应该用逗号隔开。

可以作为操作数的有常数、寄存器、标号、变量和表达式等。

1. 常数

汇编程序中的常数(Constant)可以采用不同的数制和不同的表示方法，为了避免混淆，在表示形式上应该互相有所区别。

- 十进制数。如 99D 或 99，后面加一个字母 D(Decimal)，或者什么也不加。
- 二进制。如 10101001B，后面加一个字母 B(Binary)。
- 十六进制数。如 64H，0F800H，后面加一个字母 H(Hexadecimal)，而且，当最高位十六进制数字不是 0～9 时，前面要再加一个数字 0。
- 八进制数。例如 174Q，后面加一个字母 O 或 Q(Octal)。
- ASCII 常数。例如'A'、'8'等。应将字符放在单引号中。
- 十进制科学表示法。例如 8.75E-4。
- 十六进制实数。例如 10A4FE87R，后面加一个字母 R，而且，其中十六进制数字的总位数必须等于 8、16 或 20。但若最高位不是 0～9，前面也要再加一个数字 0，此时总位数必须是 9、17 或 21。

2. 寄存器

8086 CPU 的寄存器(Register)可以作为指令的操作数。8086 CPU 的寄存器如下。

- 8 位寄存器：AH、AL、BH、BL、CH、CL、DH、DL。
- 16 位寄存器：AX、BX、CX、DX、SI、DI、BP、SP、DS、ES、SS、CS。

3. 标号

由于标号(Label)代表一条指令的符号地址，因此可以作为转移(无条件转移或条件转移)、过程调用 CALL 以及循环控制 LOOP 等指令的操作数。

4. 变量

因为变量(Variable)是存储器中某个数据区的名字，因此在指令中可以作为存储器操作数。

5. 表达式

汇编语言语句中的表达式(Expression)，按其性质可分为两种：数值表达式和地址表达式。数值表达式产生一个数值结果，只有大小，没有属性。地址表达式的结果不是一个单纯的数值，而是一个表示存储器地址的变量或标号，它有三种属性：段、偏移量和类型。

表达式中常用的运算符有以下几种：

1) 算术运算符

常用的算术运算符有＋(加)，－(减)，*(乘)，/(除)和 MOD(模除，即两个整数相除后取余数)等。

以上算术运算符可用于数值表达式，运算结果是一个数值。在地址表达式中通常只使用其中的＋和－(加和减)两种运算符。

2) 逻辑运算符

逻辑运算符有 AND(逻辑“与”)、OR(逻辑“或”)、XOR(逻辑“异或”)和 NOT(逻辑“非”)。

逻辑运算符只用于数值表达式中对数值进行按位逻辑运算中，并得到一个数值结果。对地址进行逻辑运算是没有意义的。

不要把逻辑运算符 AND、OR、XOR 和 NOT 等与同样名称的 CPU 指令相混淆。逻辑运算符可对整常数进行按位逻辑运算,这种运算由汇编程序在汇编时进行。而逻辑指令的操作数可以是寄存器、存储器或立即数(参阅第 2 章指令系统部分),指令的操作在程序运行时由 CPU 执行。汇编程序根据上下文能够将逻辑指令和逻辑运算符区分开来。例如:

```
AND     AL,01011010B
MOV     AL,01011010B AND 11110000B
```

上面第一条指令中的 AND 是指令助记符,而第二条指令的源操作数是一个表达式,其中的 AND 是逻辑运算符。

3) 关系运算符

关系运算符有 EQ(等于),NE(不等),LT(小于),GT(大于),LE(小于或等于),GE(大于或等于)等。

参与关系运算的必须是两个数值,或同一段中的两个存储单元地址,但运算结果只可能是两个特定的数值之一:当关系不成立(假)时,结果为 0;当关系成立(真)时,结果为 0FFFFH(即-1)。例如:

```
MOV    AX,4  EQ  3      ;关系不成立,故(AX)←0
MOV    AX,4  NE  3      ;关系成立,故(AX)←0FFFFH
```

4) 分析运算符和合成运算符

分析运算符用以分析一个存储器操作数的属性,如段、偏移量或类型等。合成运算符则可以规定存储器操作数的某个属性,例如类型。

分析运算符有 OFFSET、SEG、TYPE、SIZE 和 LENGTH 等;合成运算符有 PTR、THIS、SHORT 等。

(1) OFFSET

利用运算符 OFFSET 可以得到一个标号或变量的偏移地址,例如:

```
MOV    SI,OFFSET   DATA1
```

这条指令与下面的指令效果相同,均将变量 DATA1 的偏移地址送 SI 寄存器。

```
LEA    SI,DATA1
```

(2) SEG

利用运算符 SEG 可以得到一个标号或变量的段值,例如,下面两条指令的执行结果是将变量 ARRAY 的段地址送 DS 寄存器。

```
MOV    AX,SEG   ARRAY
MOV    DS,AX
```

(3) TYPE

运算符 TYPE 的运算结果是一个数值,这个数值与存储器操作数类型属性的对应关系如表 3.1 所示。

表 3.1　TYPE 返回值与类型的关系

TYPE 返回值	存储器操作数的类型	TYPE 返回值	存储器操作数的类型
1	BYTE	−1	NEAR
2	WORD	−2	FAR
4	DWORD		

下面是使用 TYPE 运算符的语句例子：

```
VAR     DW      ?                   ;变量 VAR 的类型为字
ARRAY   DD      10 DUP(?)           ;变量 ARRAY 的类型为双字
STR     DB      'This is a test'    ;变量 STR 的类型为字节
        ⋮
        MOV     AX,TYPE VAR         ;(AX)←2
        MOV     BX,TYPE ARRAY       ;(BX)←4
        MOV     CX,TYPE STR         ;(CX)←1
        ⋮
```

程序中的伪操作命令 DW、DD、DB 等将在本章 3.3 节介绍。

(4) LENGTH

如果一个变量已用重复操作符 DUP 说明其变量的个数，则利用 LENGTH 运算符可得到这个变量的个数。如果未用 DUP 说明，则得到的结果总是 1。

例如，在上面的例子中已经用“10 DUP(?)”说明变量 ARRAY 的个数，则 LENGTH ARRAY 的结果为 10。

(5) SIZE

如果一个变量已用重复操作符 DUP 说明，则利用 SIZE 运算符可得到分配给该变量的字节总数。如果未用 DUP 说明，则得到的结果是 TYPE 运算的结果。

例如，上面例子中变量 ARRAY 的个数为 10，类型为 DWORD(双字)，因此 SIZE ARRAY 的结果为 10×4＝40，由此可知，SIZE 的运算结果等于 LENGTH 的运算结果乘 TYPE 的运算结果。

(6) PTR

PTR 是一个合成运算符，可用以指定存储器操作数的类型，例如：

```
INC  BYTE  PTR[BX][SI]
```

指令中利用 PTR 运算符明确规定存储器操作数的类型是 BYTE(字节)，因此，本指令将一个 8 位存储器的内容加 1。

利用 PTR 运算符还可以建立一个新的存储器操作数，它与原来的同名操作数具有相同的段和偏移量，但可以有不同的类型。不过这个新类型只在当前语句中有效。例如：

```
STUFF   DD      ?                   ;STUFF 定义为双字类型变量
        ⋮
        MOV     BX,WORD PTR STUFF   ;从 STUFF 中取一个字到 BX
```

⋮

(7) THIS

运算符 THIS 也可以指定存储器操作数的类型。使用 THIS 运算符可以使标号或变量的类型具有灵活性。例如要求对同一个数据区,既可以字节作为单位,又可以字作为单位进行存取,则可用以下语句:

```
AREAW    EQU    THIS    WORD
AREAB    DB     100     DUP(?)
```

上面 AREAW 和 AREAB 实际上代表同一个数据区,其中共有 100 个字节,但 AREAW 的类型为 WORD,而 AREAB 的类型为 BYTE。

(8) SHORT

运算符 SHORT 指定一个标号的类型为 SHORT(短标号),即标号到引用标号的指令间的距离在-128～+127 个字节的范围内。短标号可以被用于无条件转移和条件转移指令中。使用短标号的指令比使用默认的近标号的指令少一个字节。

5) 其他运算符

(1) 方括号[ ]

例如,间址寻址指令的存储器操作数要在寄存器名 BX、BP、SI 或 DI 外面加上方括号,以表示存储器地址。又如,变址寻址指令的存储器操作数既要用算术运算符(加或减)将 SI 或 DI 与一个位移量(disp)作运算,又要在外面加上方括号来表示存储器地址。下面是间址寻址和变址寻址指令的例子:

```
MOV      CL,[BX]
MOV      AL,[SI+5]
```

(2) 段超越运算符":"

运算符":"(冒号)跟在段寄存器名(DS、ES、SS 或 CS)之后表示段超越,用以给一个存储器操作数指定一个段属性,而不管其原来隐含的段是什么。例如:

```
MOV    AX,ES:[DI]
```

(3) HIGH 和 LOW

运算符 HIGH 和 LOW 分别用来得到一个数值或地址表达式的高位和低位字节。例如:

```
STUFF    EQU    0ABCDH
         MOV    AH,HIGH  STUFF        ;(AH)←0ABH
         MOV    AL,LOW   STUFF        ;(AL)←0CDH
```

以上介绍了表达式中使用的各种运算符。如果一个表达式中同时具有多个运算符,则按照以下规则进行运算:

- 优先级高的先运算;优先级低的后运算。
- 优先级相同时按表达式中从左到右的顺序运算。

• 圆括号内的运算总是在其任何相邻的运算之前进行。

各种运算符的优先级顺序如表 3.2 所示。表中同一行中的运算符具有同等的优先级。

表 3.2　运算符的优先级

优先级	运　算　符	优先级	运　算　符
（高）		8	+，-（二元运算符）
1	LENGTH,SIZE,WIDTH,MASK,( ),[],< >	9	EQ,NE,LT,LE,GT,GE
2	.（结构变量名后面的运算符）	10	NOT
3	:（段超越运算符）	11	AND
4	PTR,OFFSET,SEG,TYPE,THIS	12	OR,XOR
5	HIGH,LOW	13	SHORT
6	+，-（一元运算符）	（低）	
7	* ,/,MOD,SHL,SHR		

3.2.4　注释

汇编语言语句的最后一个组成部分是注释(Comment)。对于一个汇编语言语句来说，注释部分并不是必要的，但是加上适当的注释以后，可以增加源程序的可读性。一个较长的实用程序，如果从头至尾没有任何注释，可能很难读懂。因此最好在重要的程序段前面以及关键的语句处加上简明扼要的注释。

注释前面要求加上分号(;)。如果注释的内容较多，超过一行，则换行以后前面还要加上分号。注释也可以从一行的最前面开始。

汇编程序对于注释不予理会，即注释对汇编后产生的目标程序没有任何影响。

3.3　伪操作命令

指示性语句中的伪操作命令，无论从表示形式或其在语句中所处的位置来看，都与 CPU 指令相似，因此也称为伪指令。但两者之间又有着重要的区别。首先，CPU 指令是给 CPU 的命令，在程序运行时由 CPU 执行，每条指令对应 CPU 的一种特定的操作，例如传送、进行加法运算等；而伪操作命令是给汇编程序的命令，在汇编过程中由汇编程序进行处理，例如定义数据、分配存储区、定义段以及定义过程等。其次，汇编以后，每条 CPU 指令产生一一对应的目标代码；而伪操作则不产生与之相应的目标代码。

宏汇编程序 MASM 提供了大约几十种伪操作，其中有一些伪操作命令小汇编 ASM 不能支持，例如宏处理伪操作等。根据伪操作的功能，大致可以分为以下几类：

- 处理器方式伪操作。
- 数据定义伪操作。
- 符号定义伪操作。

- 段定义伪操作。
- 过程定义伪操作。
- 模块定义与连接伪操作。
- 宏处理伪操作。
- 条件伪操作。

此外还有列表等其他伪操作。在本教材中,不可能对所有的伪操作逐一进行详细的说明,本节主要介绍一些基本的、常用的伪操作命令。

3.3.1 处理器方式伪操作

对于80X86系列的CPU来说,其中高一级CPU的指令系统总是兼容了下一级CPU的全部指令,并在此基础上又扩充和增加了一些指令。因此,编写源程序时应该通知汇编程序,所编写的汇编语言源程序使用的是何种CPU的指令系统。处理器方式伪操作用以设置CPU的方式。

处理器方式伪操作的格式是在处理器名称前面加一个点。常用的有以下几种:

.8086:通知汇编程序只汇编8086 CPU的指令系统。此时若在源程序中出现80286及以上CPU的指令,则将提示出错。

.286和.286C:告诉汇编程序汇编80286非保护方式(即实地址方式)的指令。

.286P:设置80286的保护方式,此时可以接收所有80286的指令,包括保护方式和非保护方式的指令。设置保护方式的伪操作通常由系统程序员在初始化和管理保护方式时使用。

.386和.386C:告诉汇编程序汇编非保护方式的80386指令。

.386P:设置80386的保护方式,即汇编所有80386的指令。

.486和.486C:告诉汇编程序汇编非保护方式的80486指令。

.486P:设置80486的保护方式,即汇编所有80486的指令。

.586和.586C:告诉汇编程序汇编非保护方式的Pentium指令。

.586P:设置Pentium的保护方式。

这些伪操作都支持同级协处理器的指令。例如.8086支持8087的指令:.486支持80487的指令等。

一般情况下,处理器方式伪操作置于整个汇编语言源程序的开头,用以设定源程序的指令系统。但是,也可以在源程序的各个部分使用不同的处理器方式伪操作。但是,处理器方式伪操作应该放在源程序的各个逻辑段之外,而不可在一个逻辑段内部改用其他处理器方式。如果不写任何处理器方式伪操作,则汇编程序默认为是.8086方式。

3.3.2 数据定义伪操作

数据定义伪操作的用途是定义一个变量的类型,给存储器赋初值,或者仅仅给变量分配存储单元,而不赋予特定的值。

常用的数据定义伪操作有DB、DW、DD、DQ和DT等。

数据定义伪操作的一般格式为: