第3章 指令系统

CHAPTER 3

指令是让计算机完成某种操作的“命令”，某处理器可以执行的全部指令的集合称为该处理器的指令系统。不同系列的CPU有不同的指令系统。指令系统确定了CPU所能完成的功能，是用汇编语言进行程序设计的最基本部分。

IA32系列微处理器的指令集是在8086/8088CPU的指令系统上发展起来的。8086/8088CPU的指令系统是基本指令集，IA32系列微处理器的指令系统在基本指令集上进行了扩充，增加了多媒体指令(MMX指令、SSE指令等)，以及专用于保护模式下的一些指令。本章将介绍IA32系列微处理器的寻址方式及其基本指令集。

3.1 简介

CPU只能识别由二进制代码组成的机器指令。一般来说，不同系列CPU的机器指令是不同的。机器指令无论在书写、阅读和记忆方面都十分困难。为了便于使用而采用指令助记符来编写程序，从而构成汇编语言。任何一种汇编语言的指令语句都是与机器指令一一对应的，它采用规定的助记符和规定的书写格式来书写，通过汇编程序将其翻译成机器指令代码(目标代码)，让CPU执行。

IA32系列汇编语言指令语句格式如图3-1所示。其中由前向后的箭头表示是可选项，由后向前的箭头表示是重复项，圆头方框表示是语句中的关键字。

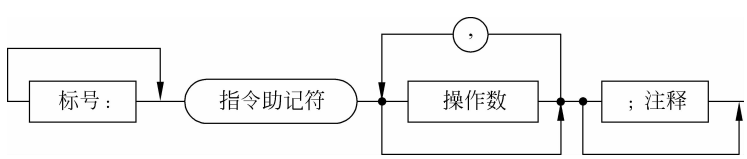


图3-1 汇编语言指令语句格式

1) 标号

标号是给该指令所在地址取的名字，可以缺省。它代表指令代码存放在存储单元的第一个字节地址，通常在代码段中定义，后面必须跟冒号“：”。通常在一个程序段或子程序的入口指令语句选用标号，当程序需要转入这个程序段或调用子程序时，就可以直接引用这个标号。它经常在转移指令或CALL指令中引用，用以表示转向地址，引用时标号后不跟冒号。

标号的命名规则是：由字母(a～z、A～Z)、数字(0～9)或某些特殊字符(@、－、?)组成。

第一个字符必须是字母(a～z、A～Z)或某些特殊符号(@、－、?),但“?”不能单独作标识符。其长度不超过 31 个字符,若超过 31 个字符,则只保留前面的 31 个字符为有效标识符。

2) 指令助记符

指令助记符是指令语句的核心,是不可缺省的组成部分,它用来说明要求 CPU 完成的具体操作,如传送、运算、移位、跳转等操作。

3) 操作数

操作数是参加本指令运算的数据,有些指令不需要操作数,可以缺省;有些指令需要两个操作数,也有个别指令需要三个甚至四个操作数,这时必须用逗号(,)将多个操作数隔开。操作数可以是一个具体的数值,也可以是存放数据的寄存器名称,或指明数据在主存位置的存储器地址。

4) 注释部分

用来说明一段程序、一条或多条指令的功能,是可选项,注释必须以分号“;”开头,可以占一行或多行。注释是语句的非执行部分,不出现在机器目标代码中,汇编程序不对它做任何处理。

对于一条汇编语言指令来说,主要有两个问题要解决:第一,要指出该指令要实现什么操作,这由指令助记符来表明;第二,要指出指令涉及的操作数存放在何处,即指出操作数的来源,这就是操作数的寻址方式。另外,指令系统中还有转移指令和调用指令这两类指令会涉及转移地址或调用地址的提供方式,一般称为指令地址的寻址方式。

因此,在两种情况下涉及寻址方式:对操作数的寻址和对指令地址的寻址。3.2 节所讨论的寻址方式都是针对操作数的,关于指令地址的寻址将在介绍控制转移指令时做具体说明。

3.2 寻址方式

本节所介绍的寻址方式是针对操作数的寻址,用来确定操作数地址从而找到操作数。操作数是指令或程序的主要处理对象。在 CPU 的指令系统中,除 NOP(空操作指令)、HLT(停机指令)等少数指令之外,大量的指令在执行过程中都会涉及操作数。所以,在指令中如何表达操作数或操作数所在位置就是正确运用汇编指令的一个重要因素。

指令系统设计了多种操作数的来源,操作数可以定位在指令、寄存器、存储单元或 I/O 端口中。寻找操作数的过程就是操作数的寻址方式。操作数采取不同的寻址方式,会对机器运行的速度和效率产生不同的影响,掌握操作数的寻址方式是用汇编语言进行程序设计的基础。

在 x86 系列 CPU 中,有七种基本的寻址方式:立即数寻址方式、寄存器寻址方式、直接寻址方式、寄存器间接寻址方式、寄存器相对寻址方式、基址变址寻址方式和相对基址变址寻址方式。其中,后五种寻址方式下操作数都存在于内存单元中,用这五种寻址方式可方便地实现对数组元素、字符串或表格的访问。另外,在 IA32 系列微处理器中还增加了与比例因子有关的三种寻址方式。

8086 和 80286 的字长是 16 位,一般只处理 8 位和 16 位数,只是在乘、除指令中才会有 32 位数;IA32 系列微处理器的字长为 32 位,因此它除了可处理 8 位和 16 位操作数外,还

可处理 32 位操作数，在乘除法情况下可产生 64 位数。本节下面所述例子，若处理的是 32 位数，则适用于 IA32 系列微处理器。

1）立即数寻址方式

操作数直接存放在指令中，紧跟在操作码之后，它作为指令的一部分存放在代码段里，这种操作数称为立即数。这种寻址方式称为立即数寻址方式。立即数可以是 8 位或 16 位，对于 IA32 系列微处理器则可以是 8 位或 32 位。如果是 16 位数，则高位字节存放在高地址中，低位字节存放在低地址中；如果是 32 位数，则高位字节存放在高地址中，低位字节存放在低地址中。

【例 3-1】

```
MOV AL,10H          ;将十六进制数 10H 送入 AL,指令执行后,(AL) = 10H
MOV BX,1234H        ; 将 1234H 送 BX,指令执行后,(BX) = 1234H
MOV EAX,12003400H ; 将 12003400H 送 EAX,指令执行后,(EAX) = 12003400H
```

因为操作数可以从指令中直接取得，不需要运行总线周期，所以，立即数寻址方式的显著特点就是速度快。立即数寻址方式常用来给寄存器或内存单元赋值，并且规定立即数只能是整数，不能是小数、变量或者其他类型的数据。立即数只能作为源操作数，不能作为目的操作数，而且源操作数长度应与目的操作数长度一致。

2）寄存器寻址方式

如果操作数存放在 CPU 的内部寄存器中，那么寄存器名可在指令中指出。这种寻址方式就叫寄存器寻址方式。

对 16 位操作数，寄存器可以为 AX、BX、CX、DX、SI、DI、SP、BP；对 8 位操作数，寄存器可为 AH、AL、BH、BL、CH、CL、DH、DL；对 32 位操作数，寄存器可以为 EAX、EBX、ECX、EDX、ESI、EDI、ESP、EBP。

【例 3-2】

```
MOV CL,AL           ; 将 AL 的内容送 CL,(AL)保持不变
MOV AX,BX           ; 将 BX 的内容送 AX,(BX)保持不变
INC SI              ; 将 SI 的内容加 1
ADD EAX,EBX         ; EAX 和 EBX 的内容相加,结果送 EAX
```

采用寄存器寻址方式的指令在执行时，由于操作数在寄存器中，操作就在 CPU 内部进行，不需要使用总线周期，因此可以得到较高的执行速度。

除了上述两种寻址方式外，以下介绍的各种寻址方式的操作数都在除代码段以外的存储区中。通过不同方式求得操作数所在存储单元的地址，从而取得操作数。操作数地址是由段基地址和偏移地址相加而取得的。段基地址在实模式和保护模式下可从不同途径取得。本节要解决的问题是如何取得操作数的偏移地址。在 IA32 系列微处理器中，把操作数的偏移地址称为有效地址（Effective Address，EA），所以下述各种寻址方式即为求得有效地址（EA）的不同途径。

3）直接寻址方式

使用直接寻址方式时，操作数总是存放在存储器中，存储单元的有效地址 EA 由指令直接指出，这种寻址方式称为直接寻址方式。直接寻址是对存储器进行访问时可采用的最简单的方式。

【例 3-3】

```
MOV AX,[1200H]  ; 将数据段中有效地址为 1200H 和 1201H 两单元的内容取到 AX 中
```

采用直接寻址方式时，如果指令中没有使用段跨越前缀指明操作数在哪一段，则默认段寄存器是数据段寄存器 DS，即默认操作数在数据段中。例如，上一条指令执行时，若(DS)＝2000H，则执行过程是将物理地址为 21200H 和 21201H 两单元的内容取出送到 AX 中。

如果操作数存在于其他段中，则在对其进行直接寻址时，要用段跨越前缀指出段寄存器名。

【例 3-4】

```
MOV BX,ES:[2000H] ; 将附加段中有效地址为 2000H 和 2001H 两单元的内容取到 BX 中
```

若(ES)＝3000H，则本指令在执行时，将物理地址为 32000H 和 32001H 两单元的内容取出送到 BX。

4) 寄存器间接寻址方式

操作数在存储器中，操作数的有效地址由寄存器指出，这种寻址方式称为寄存器间接寻址方式。在 16 位寻址时可用的寄存器是 BX、BP、SI 和 DI；在 32 位寻址时可用 EAX、EBX、ECX、EDX、ESI、EDI、ESP 和 EBP 这 8 个通用寄存器。即有效地址等于其中某一个寄存器的值：

$$EA=\begin{cases}[BX]\\[BP]\\[SI]\\[DI]\end{cases}\quad 或\quad \begin{cases}[EAX]\\[EBX]\\[ECX]\\[EDX]\\[ESI]\\[EDI]\\[ESP]\\[EBP]\end{cases}$$

如果没有用段跨越前缀指明具体的段寄存器，则在以 BP、ESP、EBP 寄存器进行间接寻址时，默认的段寄存器为 SS，以其他寄存器进行间接寻址时，默认的段寄存器为 DS。

【例 3-5】

```
MOV  AX,[BX]
```

设(DS)＝2000H，(BX)＝4000H，则本指令在执行时，将物理地址为 24000H 和 24001H 两个单元的内容送 AX。

```
MOV  BX,[BP]
```

设(SS)＝5000H，(BP)＝4000H，则本指令在执行时，将物理地址为 54000H 和 54001H 两个单元的内容送 BX。

```
MOV  ECX,[EDX]
```

把数据段中有效地址为 EDX 寄存器内容的 4 字节的存储单元数据传送到 ECX 寄存器中。

若要对其他段寄存器所指的区域进行寻址，则必须用段跨越前缀指出段寄存器名。

【例 3-6】

```
MOV  CX,ES:[SI]
```

设(ES)＝3000H，(SI)＝2100H，则本指令在执行时，将物理地址为 32100H 和 32001H 两个单元的内容送 CX。

这种寻址方式可以用于对数组、字符串、表格的处理，执行完一条指令后，只需修改寄存器内容就可以取出下一个元素。

5）寄存器相对寻址方式（或称直接变址寻址方式）

操作数在存储器中，操作数的有效地址 EA 为基址或变址寄存器的内容与指令中指定的位移量之和，所以有效地址由两种成分组成。这种寻址方式下允许使用的寄存器及其对应的默认段情况与寄存器间接寻址方式相同。EA 计算方法如下：

$$EA=\begin{cases}[BX]\\[BP]\\[SI]\\[DI]\end{cases}\quad 或\quad \begin{cases}[EAX]\\[EBX]\\[ECX]\\[EDX]\\[ESI]\\[EDI]\\[ESP]\\[EBP]\end{cases}\quad +位移量$$

【例 3-7】

```
MOV BX, 100H [SI](也可表示为 MOV BX, [SI + 100H])
```

设(DS)＝1000H，(SI)＝2340H，则 EA＝(SI)＋100H＝2440H，本指令在执行时，将物理地址为 12440H 和 12441H 两个单元的内容送 BX。

```
MOV  AX,COUNT[BP](也可表示为 MOV AX,[COUNT + BP])
```

设(SS)＝3000H，(BP)＝2000H，COUNT＝3000H，则 EA＝(BP)＋COUNT＝5000H，本指令在执行时，将物理地址为 35000H 和 35001H 两个单元的内容送 AX。

```
MOV  EAX,TABLE[ESI]
```

该指令执行时，将从数据段中有效地址为(ESI)＋TABLE 的连续 4 个存储单元中取数送给 EAX。

寄存器相对寻址方式也可以使用段跨越前缀。如：

```
MOV DL,ES:BUF[SI]
```

该寻址方式通常也用于对数组元素、字符串、表格等进行操作，操作过程中，指令会自动修改变址寄存器中的地址，以指向下一个操作数。

6）基址变址寻址方式

用这种寻址方式时，操作数的有效地址等于基址寄存器的内容加上一个变址寄存器的内容，有效地址由两种成分组成。在 16 位寻址时，基址寄存器可用 BX 和 BP，变址寄存器

可用 SI 和 DI；在 32 位寻址时，基址寄存器可用任何 32 位通用寄存器，变址寄存器可用除 ESP 以外的 32 位通用寄存器。若没有用段跨越前缀指明具体的段寄存器，对段寄存器的约定规则如下：如果基址寄存器用 BP 或 32 位的 ESP、EBP，则默认的段寄存器为 SS；其他情况下，默认的段寄存器为 DS。例如：

```
MOV  AX,[BX][SI](也可表示为 MOV  AX,[BX+SI])
```

设(DS)＝1000H，(BX)＝0156H，(SI)＝20C2H，则 EA＝(BX)＋(SI)＝2218H，指令执行时，将物理地址为 12218H 和 12219H 两个单元的内容取到 AX 中。

类似地，对于 32 位寻址方式可有：

```
MOV EDX,[EBX][ESI]
```

该寻址方式使用段跨越前缀时的格式为：

```
MOV AX, ES:[BX][SI]
```

基址变址寻址方式同样适用于数组、字符串或表格处理，首地址可存放在基址寄存器中，而用变址寄存器来访问数组的各个元素。因为允许两个地址分量分别改变，所以该寻址方式使用起来更加灵活。

7）相对基址变址寻址方式

相对基址变址寻址方式中，操作数在存储器中，其有效地址 EA 是基址寄存器内容、变址寄存器内容和指令中指定的位移量三项之和，有效地址由三种成分组成。这种寻址方式下允许使用的寄存器及其对应的默认段情况与基址变址寻址方式相同。

【例 3-8】

```
MOV  AX,MASK[BX][SI]
(也可写成  MOV  AX,MASK[BX+SI]或 MOV  AX,[MASK+BX+SI])
```

设(DS)＝3000H，(BX)＝2000H，(SI)＝1000H，MASK＝0250H，则 EA＝(BX)＋(SI)＋MASK＝3250H，指令执行时，将物理地址为 33250H 和 33251H 两个单元的内容取到 AX 中。

类似地，对于 32 位寻址方式可有：

```
MOV EAX,ARRAY[EBX][ECX]
```

从相对基址变址这种寻址方式来看，由于它的可变因素较多，看起来就显得复杂些，但正因为其可变因素多，它的灵活性也就很高。这种寻址方式通常用于对二维数组的寻址。例如，存储器中存放着由多个记录组成的文件，则位移量可指向文件之首，基址寄存器指向某个记录，变址寄存器则指向该记录中的一个元素。这种寻址方式也为堆栈处理提供了方便，一般可用 BP 指向栈顶，从栈顶到数组的首址可用位移量表示，变址寄存器可用来访问数组中的某个元素。

相对基址变址寻址方式有多种等价的书写方式，下面的书写格式都是正确的，并且其寻址含义也是一致的。

```
MOV  AX, [BX+SI+1000H]        MOV  AX, 1000H[BX+SI]
MOV  AX, 1000H[BX][SI]        MOV  AX, 1000H[SI][BX]
```

以下将要介绍的三种寻址方式只能用在 IA32 系列微处理器及其后继机型中，8086/80286 不支持这几种寻址方式，它们均与比例因子有关。比例因子是 386 及其后继机型新增加的寻址方式中的一个术语，其值可为 1、2、4 或 8，比例因子只能与变址寄存器同时使用。在寻址中，可用变址寄存器的内容乘以比例因子来取得变址值。这类寻址方式对访问元素长度为 2、4、8 字节的数组特别有用。

8）比例变址寻址方式

操作数的有效地址是变址寄存器的内容乘以指令中指定的比例因子再加上位移量之和，有效地址由三种成分组成。

寻址时可用的变址寄存器可以是 EAX、EBX、ECX、EDX、ESI、EDI 或者 EBP 这 7 个 32 位通用寄存器。如果没有用段跨越前缀指明具体的段寄存器，则在以 EBP 寄存器进行变址寻址时，默认的段寄存器为 SS，以其他寄存器进行变址寻址时，默认的段寄存器为 DS。

这种寻址方式与相对寄存器寻址相比，增加了比例因子。其优点在于：对于元素大都为 2、4、8 字节的数组，可以在变址寄存器中给出数组元素下标，而由寻址方式控制直接用比例因子把下标转换为变址值。

【例 3-9】

```
MOV  EAX, COUNT [ESI * 4]
```

如要求把双字数组 COUNT 中的元素 3 送到 EAX 中，用这种寻址方式可直接在 ESI 中放入 3，选择比例因子 4(数组元素为 4 字节长)就可以方便地达到目的，而不必像在相对寄存器寻址方式中要把变址值直接装入寄存器中。该指令的执行情况如图 3-2 所示。

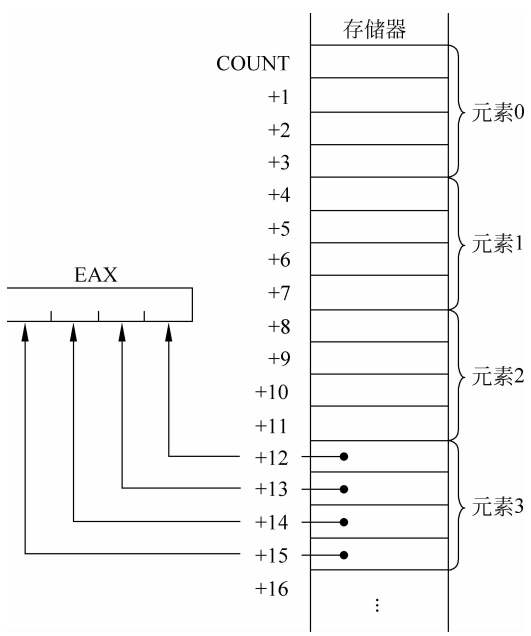


图 3-2　指令 MOV EAX,COUNT [ESI * 4]的执行情况

9）基址比例变址寻址方式

操作数的有效地址是变址寄存器的内容乘以比例因子再加上基址寄存器的内容之和，

有效地址由三种成分组成。这种寻址方式下，可以使用 EAX、EBX、ECX、EDX、ESI、EDI、EBP、ESP 这 8 个 32 位通用寄存器做基址寄存器，使用 EAX、EBX、ECX、EDX、ESI、EDI 或者 EBP 这 7 个 32 位通用寄存器做变址寄存器。若没有用段跨越前缀指明具体的段寄存器，对段寄存器的约定规则如下：如果基址寄存器用 ESP、EBP，则默认的段寄存器为 SS；其他情况下，默认的段寄存器为 DS。

这种寻址方式与基址变址寻址相比，增加了比例因子，其优点也是非常明显的。

【例 3-10】

```
MOV  ECX,[EAX][EDX * 8]
```

10）相对基址比例变址寻址方式

操作数的有效地址是变址寄存器的内容乘以比例因子，加上基址寄存器的内容，再加上位移量之和，有效地址由四种成分组成。这种寻址方式允许使用的寄存器及其对应的默认段情况与基址比例变址寻址方式相同。

这种寻址方式与相对基址变址方式相比，增加了比例因子，便于对元素为 2、4、8 字节的二维数组的处理，如：

```
MOV  EAX,TABLE[EBP][EDI * 4]
```

对于寻址方式的几点说明。

① 在一条指令中，若有两个操作数，可以采用不同的寻址方式，也可以采用相同的寻址方式。

【例 3-11】

```
ADD  AX,3064H      ; 源操作数采用立即寻址,目的操作数采用寄存器寻址
MOV  AX,CX         ; 源操作数和目的操作数均采用寄存器寻址
```

② 汇编语言中，常将“[ ]”方括号中的内容作为存储单元的地址，如[BX]、[DI]、[2056H]。

③ 除立即数寻址和寄存器寻址外，其他寻址方式都有隐含的段寄存器，操作数实际的物理地址由段寄存器的内容与指令码中求得的有效地址相加求得。如果要访问默认段之外的其他逻辑段，那么指令中必须明确写出相关逻辑段的段跨越前缀。

④ 在带有位移量的寻址方式中，若是 16 位寻址，则位移量可以是 8 位或 16 位的无符号整数；若是 32 位寻址，则位移量可以是 8 位或 32 位的无符号整数。

⑤ 在用 16 位寄存器来访问存储单元时，只能使用基地址寄存器（BX 和 BP）和变址寄存器（SI 和 DI）来作为地址偏移量的一部分，但在用 32 位寄存器寻址时，不存在上述限制，所有 32 位寄存器（EAX、EBX、ECX、EDX、ESI、EDI、EBP 和 ESP）都可以是地址偏移量的一个组成部分。其中，32 位基址寄存器是 EAX、EBX、ECX、EDX、ESI、EDI、EBP 和 ESP；32 位变址寄存器是 EAX、EBX、ECX、EDX、ESI、EDI 和 EBP（除 ESP 之外）。32 位寻址时的有效地址 EA 计算公式如下。

EA＝（基址寄存器）＋（变址寄存器）×比例因子＋位移量

由于 32 位寻址方式能使用所有的通用寄存器，所以，和该有效地址相组合的段寄存器也就有新的规定。具体规定如下。

地址中寄存器的书写顺序决定该寄存器是基址寄存器还是变址寄存器。例如，[EBX＋EBP]中的 EBX 是基址寄存器，EBP 是变址寄存器，而[EBP＋EBX]中的 EBP 是基址寄存器，EBX 是变址寄存器；默认段寄存器的选用取决于基址寄存器：基址寄存器是 EBP 或 ESP 时，默认的段寄存器是 SS，否则，默认的段寄存器是 DS。

⑥ 在除直接寻址方式之外的存储器寻址方式中，可以使用 16 位的寄存器寻址，也可以使用 32 位的寄存器寻址，但要注意一个问题：当 CPU 工作在实地址模式的时候，段长度最大为 64KB，无论采用 16 位寄存器寻址还是 32 位寄存器寻址都必须保证 CPU 最终算出的偏移地址不超过 FFFFH，而且操作数最高字节单元的偏移地址也不能超过 FFFFH，否则执行寻址操作时系统会瘫痪。

【例 3-12】

```
MOV EBX,10000H      ; EBX 中偏移地址大于 FFFFH
MOV AL,[EBX]        ; 执行该指令，系统瘫痪
MOV SI,0FFFFH       ; 虽然 SI 中偏移地址不大于 FFFFH
MOV AX,[SI]         ; 但[SI]寻址的是双字节数，高字节偏移地址为 10000H，超出了 FFFFH，执行该指
                    ; 令系统死机
```

3.3　IA32 微处理器的基本指令集

IA32 微处理器的基本指令集可以分为以下 6 组：数据传送指令、算术指令、逻辑指令、串处理指令、控制转移指令和处理机控制指令。下面将分别对这 6 组指令加以说明。

3.3.1　数据传送指令

数据传送是计算机中最基本、最重要的一种操作，可以实现数据、地址、标志的传送，执行寄存器和寄存器之间、主存和寄存器之间、AL/AX 与外设端口之间的多种传送操作。这类指令又可以分为通用数据传送指令、累加器专用传送指令、地址传送指令、标志寄存器传送指令和类型转换指令。该类指令中除了目标地址为标志寄存器的传送指令外，其他指令不影响标志。

1. 通用数据传送指令

1）传送指令

① 最基本的传送指令 MOV

MOV 指令是形式最简单、用得最多的指令。它可以实现 CPU 内部寄存器之间、寄存器和内存之间的数据传送，还可以把一个立即数送给 CPU 的内部寄存器或内存单元。

语句格式：`MOV  DST,SRC`

其功能是将源操作数传送给目的操作数，指令执行后，目的操作数的值被改变，而源操作数的值不变。其中，DST 表示目的操作数，SRC 表示源操作数，以下相同。该指令对标志位没有影响。

【例 3-13】

```
MOV  AH, 'H'              ; 把立即数(字符 H 的 ASCII 码)送到 AH 寄存器
MOV  AX,BUF               ; BUF 是变量，将 BUF 的内容送到 AX 寄存器
MOV  BX,[BP + SI]         ; 将堆栈段中有效地址为(BP) + (SI)的存储单元的内容送到 BX 寄存器
```

```
MOV  [DI],AX              ; 将累加器 AX 的内容送到有效地址为(DI)的存储单元
MOV  EAX,[EBX+ECX*4]      ; 将 DS 段中有效地址为(EBX)+(ECX)*4 的存储单元的 32 位内容送给
                          ; EAX 寄存器
```

说明：

源操作数可以是 8 位、16 位或 32 位的立即数、寄存器、段寄存器或内存操作数。目的操作数是与源操作数等长的寄存器、段寄存器(CS 除外)或内存单元。

源、目的操作数类型必须匹配，如果目的操作数为内存单元，而源操作数为立即数，则必须用 PTR 运算符说明目的操作数的属性。例如：

```
MOV BYTE PTR [SI],24H             ; BYTE PTR 说明是字节操作
MOV WORD PTR BUF[BX][SI],1234H    ; WORD PTR 说明是字操作
```

不能向段寄存器写入立即数，对段寄存器初始化应借用一个通用寄存器过渡。例如：

```
MOV AX,DATA                       ; DATA 为数据段段名，汇编后即为 DATA 段的段基址
MOV DS,AX
```

MOV 指令的目的操作数不允许用立即数方式，也不允许用 CS 寄存器，而且除源操作数为立即数的情况外，两个操作数中必须有一个是寄存器。也就是说，不允许用 MOV 指令在两个存储单元之间直接传送数据。此外，也不允许在两个段寄存器间直接传送信息。

② 带符号扩展传送指令 MOVSX

语句格式：MOVSX DST,SRC

其功能是将源操作数的符号位向高位扩展使其与目标字长相同，然后再传送到目的寄存器，而源操作数不变。该指令的源操作数可以是 8 位或 16 位的寄存器或存储单元的内容，而目的操作数必须是 16 位或 32 位的寄存器。可以是 8 位符号扩展到 16 位或 32 位，也可以是 16 位符号扩展到 32 位。该指令不影响标志位。

【例 3-14】

```
MOVSX BX,DL               ; 将 DL 中的 8 位数符号扩展为 16 位数，送到 BX 中
MOVSX EAX,CL              ; 将 CL 中的 8 位数符号扩展为 32 位数，送到 EAX 中
MOVSX EDX,[EDI]           ; 将 DS 段中由 EDI 内容指定地址的 16 位数符号扩展为 32 位
                          ; 数，送到 EDX 中
```

③ 带零扩展传送指令 MOVZX

语句格式：MOVZX DST,SRC

其功能是将源操作数高位用零补足 16 位或 32 位，然后再传送到目的寄存器，源操作数不变。关于源操作数和目的操作数以及对标志位的影响均与 MOVSX 相同。

【例 3-15】

```
MOVZX BX,CL               ; 将 CL 寄存器中的 8 位数零扩展成 16 位数，送到 BX 中
MOVZX EAX,DX              ; 将 DX 寄存器中的 16 位数零扩展成 32 位数，送到 EAX 中
```

MOVSX 和 MOVZX 的差别是 MOVSX 的源操作数是带符号数，作符号扩展；而 MOVZX 的源操作数是无符号数，作零扩展(无论源操作数的符号位是否为 1，高位均扩展为 0)。MOVSX 和 MOVZX 指令与一般双操作数指令的差别是：一般双操作数指令的源操作数和目的操作数的长度是一致的，但 MOVSX 和 MOVZX 的源操作数长度一定要小于

目的操作数长度。

2）堆栈操作指令

在介绍堆栈操作指令之前，先介绍一下什么是堆栈，以及为什么需要堆栈。

在子程序调用和中断处理过程时，分别要保存返回地址和断点地址，在进入子程序和中断处理后，还需要保留通用寄存器的值；子程序返回和中断处理返回时，则要恢复通用寄存器的值，并分别将返回地址或断点地址恢复到指令指针寄存器中。这些功能都要通过堆栈来实现，其中寄存器的保存和恢复需要由堆栈指令来完成。

堆栈是人为定义的一块连续的内存空间，用来暂存数据。这些数据按照先进后出的规律存取。在IA32系列中，栈底为堆栈空间的高地址单元，栈顶为低地址单元。数据进栈后，栈顶向低地址方向浮动；数据出栈后，栈顶向高地址方向调整。一个16位或32位的数据进栈的规律是：高位字节存入高地址单元，低位字节存入低地址单元。一个16位或32位数据的出栈规律是：低位字节弹出到目标操作数低位，高位字节弹出到目标操作数高位。为了指示栈顶的当前位置，用栈顶指针寄存器SP或ESP存放栈顶的有效地址。堆栈的段地址存放在SS中，栈顶指针在任何时候都指向栈顶，进出栈后自动修改栈顶指针寄存器的值。

堆栈操作指令可分为两类：把信息压入堆栈的指令和把信息弹出堆栈的指令。

① 进栈指令PUSH

语句格式：PUSH　SRC

该指令执行时，首先调整堆栈指针，然后把源操作数压入堆栈，即执行的操作为：

16位指令：(SP)←(SP)－2

((SP)＋1,(SP))←(SRC)

32位指令：(ESP)←(ESP)－4

((ESP)＋3,(ESP)＋2,(ESP)＋1,(ESP))←(SRC)

如果源操作数是SP或ESP，则CPU将调整前的SP或ESP压栈。

进栈数据可以是段寄存器内容，也可以是16位或32位的立即数、通用寄存器或内存操作数。如果内存操作数不是采用直接寻址，则必须用PTR运算符说明其属性。

【例 3-16】

```
PUSH 1200H                    ; (SP)←(SP) - 2,((SP) + 1,(SP))←立即数 1200H
PUSH AX                       ; (SP)←(SP) - 2,((SP) + 1,(SP))←(AX)
PUSH EBX                      ; (ESP)←(ESP) - 4,(ESP)起始的 4 个单元←(EBX)
PUSH WORD PTR[SI]             ; (SP)←(SP) - 2,((SP) + 1,(SP))←数据段中有效地址为(SI)
                              ; 的 2 个存储单元的内容
PUSH DWORD PTR BUF[SI]        ; (ESP)←(ESP) - 4,(ESP)起始的 4 个单元←数据段中有效地
                              ; 址为(BUF + SI)的 4 个存储单元的内容
```

② 出栈指令POP

语句格式：POP　DST

其功能是先从栈顶弹出2个或4个字节送目的操作数，然后调整堆栈指针，即执行的操作为：

16位指令：(DST)←((SP)＋1,(SP))

(SP)←(SP)＋2

32 位指令：(DST)←((ESP)+3,(ESP)+2,(ESP)+1,(ESP))
(ESP)←(ESP)+4

目的操作数可以是除 CS 之外的段寄存器，也可以是 16 位或 32 位的通用寄存器或内存单元。在 IA32 系列指令系统中，不允许 CS 寄存器作为目的操作数使用，因为一旦改变了代码段寄存器 CS 的内容，使程序有了新的当前代码段，就会导致 CPU 从新的 CS 和 IP 给出的毫无意义的地址中去取下一条指令，使程序错误运行。如果内存操作数不是采用直接寻址，则必须用 PTR 运算符说明其属性。例如：

```
POP AX                          ; (AX)←((SP)+1,(SP)),(SP)←(SP)+2
POP DWORD PTR[SI]               ; 数据段中有效地址为(SI)的 4 个存储单元←(ESP)起始的
                                ; 4 个单元的内容,(ESP)←(ESP)+4
```

【例 3-17】

假设(SS)=0400H,(SP)=0056H,(AX)=102AH,(BX)=3CF0H,执行以下三条指令后，堆栈空间的数据变化如图 3-3 所示。

```
PUSH AX
PUSH BX
POP  CX
```

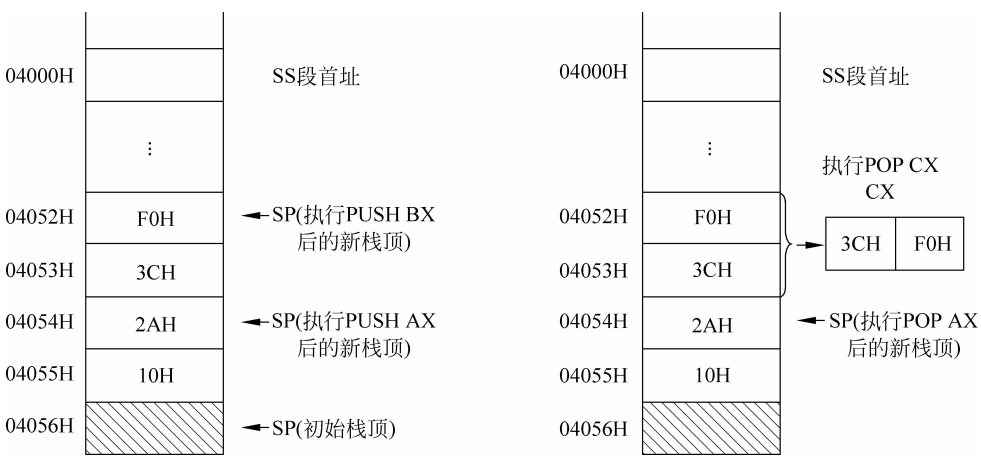


图 3-3　堆栈空间示意

堆栈的存取在 16 位指令中必须以字为单位(不允许字节堆栈)，在 32 位指令中必须以双字为单位，所以 PUSH 和 POP 指令只能做字或双字操作。

PUSH 和 POP 指令均不影响标志位。

③ 所有寄存器进栈指令 PUSHA/PUSHAD

语句格式：PUSHA
PUSHAD

执行的操作如下。

PUSHA：16 位通用寄存器依次进栈。执行 PUSHA 时，首先(SP)←(SP)－16，然后将 16 位通用寄存器依次压入堆栈，进栈次序为 AX、CX、DX、BX，指令执行前的 SP、BP、SI、DI。

PUSHAD：32 位通用寄存器依次进栈。执行 PUSHAD 时，首先(ESP)←(ESP)－32，然后将 32 位通用寄存器依次压入堆栈，进栈次序为 EAX、ECX、EDX、EBX，指令执行前的 ESP、EBP、ESI、EDI。

④ 所有寄存器出栈指令 POPA/POPAD

语句格式：POPA
　　POPAD

执行的操作如下。

POPA：16 位通用寄存器依次出栈。执行 POPA 时，先从栈顶弹出 16 个字节，并依次装入 DI、SI、BP、SP、BX、DX、CX、AX。

POPAD：32 位通用寄存器依次出栈。执行 POPAD 时，先从栈顶弹出 32 个字节(8 个双字)，并依次装入 EDI、ESI、EBP、ESP、EBX、EDX、ECX、EAX。

堆栈在计算机工作中起着重要的作用，如果在程序中要用到某些寄存器，但它们的内容却在将来还有用，这时就可以用 PUSHA/PUSHAD 指令把它们保存在堆栈中，然后在需要时再用 POPA/POPAD 指令恢复其原始内容。

PUSHA/PUSHAD 和 POPA/POPAD 指令均不影响标志位。

3) 数据交换指令

① 交换指令 XCHG

该指令可以实现字节、字或双字交换。

语句格式：XCHG　DST,SRC

其功能是将源操作数与目的操作数的内容互换。

该指令的两个操作数中必须有一个在寄存器中，因此它可以在寄存器之间或者在寄存器与存储器之间交换信息，但不允许使用段寄存器，不能在两个存储单元之间进行数据交换。该指令可用除立即数之外的任何寻址方式，并且不影响标志位。

【例 3-18】

```
XCHG  AL,BL       ; AL 和 BL 之间进行字节交换
XCHG  BX,CX       ; BX 和 CX 之间进行字交换
XCHG  [2530],CX   ; CX 中的内容和数据段中有效地址为 2530、2531 两单元的内容交换
XCHG  EAX,EBX     ; EAX 和 EBX 寄存器的内容互换
```

② 字节交换指令 BSWAP

语句格式：BSWAP　DST

该指令的目的操作数为 32 位的通用寄存器。其功能是将 32 位通用寄存器的 31～24 位与 7～0 位交换，23～16 与 15～8 位交换。

【例 3-19】

```
BSWAP EAX
```

若指令执行前(EAX)＝12345678H，则指令执行后(EAX)＝78563412H，字节次序变反。

2. 累加器专用传送指令

这组指令只限于使用累加器 EAX、AX 或 AL 传送信息，包括输入/输出指令和换码

指令。

由于I/O端口地址和内存单元地址是相互独立的，这些端口地址不能用普通的访问内存指令来访问其信息，所以，在指令系统中就专门设置了输入/输出指令来存取I/O端口的信息。输入/输出指令用来完成累加器EAX、AX或AL与I/O端口之间的数据传送功能。如果I/O端口地址为一个字节，即在0～255范围之内，那么可以采用直接寻址方式，即长格式，端口地址可在输入/输出中直接给出，最多可访问256个端口；如果I/O端口地址为两个字节，即端口地址≥256，则必须用间接寻址方式，即短格式，要先把该端口地址放到寄存器DX中，然后在输入/输出指令中由DX给出其端口地址。端口地址在0～255范围之内时，也可使用间接寻址方式。间接寻址方式最多可寻址2^{16}个端口。

1）输入指令IN

输入指令用来从指定的I/O端口取信息送入累加器。

长格式：IN AL,PORT(字节)
IN AX,PORT(字)
IN EAX,PORT(双字)

执行的操作：(AL)←(PORT)(字节)
(AX)←(PORT+1,PORT)(字)
(EAX)←(PORT+3,PORT+2,PORT+1,PORT)(双字)

短格式：IN AL,DX(字节)
IN AX,DX(字)
IN EAX,DX(双字)

执行的操作：(AL)←((DX))(字节)
(AX)←((DX)+1,(DX))(字)
(EAX)←((DX)+3,(DX)+2,(DX)+1,(DX))(双字)

2）输出指令OUT

输出指令用来把累加器的内容送往指定的I/O端口。

长格式：OUT PORT,AL(字节)
OUT PORT,AX(字)
OUT PORT,EAX(双字)

执行的操作：(PORT)←(AL)(字节)
(PORT+1,PORT)←(AX)(字)
(PORT+3,PORT+2,PORT+1,PORT)←(EAX)(双字)

短格式：OUT DX,AL(字节)
OUT DX,AX(字)
OUT DX,EAX(双字)

执行的操作：((DX))←(AL)(字节)
((DX)+1,(DX))←(AX)(字)
((DX)+3,(DX)+2,(DX)+1,(DX))←(EAX)(双字)

IN和OUT指令提供了字节、字和双字三种使用方式，选用哪一种，则取决于外设端口宽度。若端口宽度只有8位，则只能用字节指令传送信息。IN和OUT指令均不影响标志位。

需要注意的一点：这里的端口地址或DX的内容均为地址，而传送的是端口中的信息，而且在用短格式时DX内容就是端口地址本身，不需要由任何段寄存器来修改它的值。

【例 3-20】

```
IN    AX, 20H     ; 将端口 20H 和 21H 的内容送入 AX
MOV   [20H], AX   ; 将 AX 内容传送到内存单元 20H 和 21H 中
MOV   DX, 180H
IN    EAX, DX     ; 从端口 180H 送一个双字到 EAX 寄存器
OUT   70H, AX     ; 从 AX 寄存器输出一个字到端口 70H 和 71H
MOV   DX,1A8H     ;
MOV   AL, 40H     ;
OUT   DX, AL      ; 这三条指令将 40H 输出到端口 1A8H
```

【例 3-21】 测试某状态寄存器(端口号 27H)的第 2 位是否为 1。

```
IN    AL, 27H
TEST  AL, 00000100B
JNZ   ERROR       ; 若第 2 位为 1,则转 ERROR 处理
```

3) 换码指令 XLAT

语句格式：XLAT OPR

或 XLAT

执行的操作：

16 位指令：(AL)←((BX)+(AL))

32 位指令：(AL)←((EBX)+(AL))

XLAT 是一条完成字节翻译功能的指令,它可以使累加器中的一个值变换为内存表格中的某一个值,一般用来实现编码制的转换。例如,把字符的扫描码转换成 ASCII 码,或者把数字 0～9 转换成 7 段数码管所需要的相应代码等。在使用换码指令之前,要先建立一个字节表格(该表的最大容量为 256 字节),表格的首地址提前存于 BX 或 EBX 寄存器,需要转换的代码应该是相对于表格首地址的位移量也提前存于 AL 寄存器中,表格的内容则是要换取的代码。指令执行时,将 BX 或 EBX 与 AL 中的值相加,把得到的值作为地址,然后将此地址所对应的内存表格中的值取到 AL 中去。该指令执行后即可在 AL 中得到转换后的代码。该指令可用 XLAT 或 XLAT OPR 两种格式中的任一种,使用 XLAT OPR 时,OPR 为表格的首地址(一般为符号地址),但这里的 OPR 只是为了提高程序的可读性而设置的,指令执行时只使用预先已存入 BX 或 EBX 中的表格首地址,而并不用汇编格式中指定的值。该指令不影响标志位。

【例 3-22】 将 NUM 单元中的十进制数字翻译成 7 段数码管所需要的相应代码。

十进制数对应的 7 段显示码如表 3-1 所示。

表 3-1　十进制数的 7 段显示码

十进制数字	0	1	2	3	4	5	6	7	8	9
7 段显示码(H)	3F	06	5B	4F	66	6D	7D	07	7F	6F

首先应在数据段按 0～9 的顺序设置一个 7 段显示码表：

```
TABLE DB 3FH, 06H,5BH,4FH,66H,6DH,7DH,07H,7FH,6FH
NUM   DB× ×   ; 0～9 中的任一数
```

代码段编写如下指令，即可得到 NUM 单元中数字对应的 7 段显示码：

```
…
MOV    BX,OFFSET TABLE
MOV    AL,NUM
XLAT                ; AL 中的内容即为相应的 7 段显示码
```

3. 地址传送指令

地址传送指令主要用于将存储器操作数地址(段地址、偏移地址)传送给指定的寄存器。它包括 6 条指令：有效地址送寄存器指令 LEA；指针送寄存器和段寄存器指令 LDS、LES、LFS、LGS 和 LSS。

1) 有效地址送寄存器指令 LEA

语句格式：LEA REG,SRC

其功能是将源操作数的有效地址送到指定的目的寄存器中。LEA 指令格式中，要求源操作数必须为内存操作数；目的操作数为 16 位或 32 位的寄存器，但不能使用段寄存器。这条指令常用来使一个寄存器作为地址指针。例如：

```
LEA    AX,LIST        ; 将 LIST 单元的偏移地址送 AX
LEA    BX,[BP + SI]   ; 指令执行后,BX 中的内容为 BP + SI 的值
LEA    EAX,[SI + 2]   ; 将数据段采用 SI + 2 寄存器相对寻址的那个存储单元的偏移地址送 EAX
```

偏移地址传送还可以用 MOV 指令完成。例如：

```
MOV    AX,OFFSET LIST     ; 将 LIST 单元的偏移地址送 AX
```

OFFSET 是汇编语言提供的运算符，在源程序汇编时完成偏移地址计算，执行该指令时完成赋值。MOV AX,OFFSET LIST 与 LEA AX,LIST 在功能上是相同的，并且此时 MOV 指令的执行速度比 LEA 指令更快。但是，OFFSET 只能与简单的符号地址相连，而不能和诸如 LIST[SI]或[SI]等复杂操作数相连。

2) 指针送寄存器和段寄存器指令

语句格式(以 LDS 为例)：LDS REG,SRC

其他 4 条指令 LES、LFS、LGS、LSS 格式与 LDS 相同，仅指定的段寄存器不同。

执行的操作：(REG)←(SRC)
(SREG)←(SRC + 2)或(SREG)←(SRC + 4)

指令助记符的后两位字母代表段寄存器，它们是隐含的目的寄存器。

该组指令的源操作数只能用存储器寻址方式，根据任一种存储器寻址方式找到一个存储单元。

如果目标是 16 位通用寄存器，则源操作数应是 32 位内存操作数，指令执行后，内存操作数高 16 位送到隐含的段寄存器，低 16 位送到目标指定的通用寄存器。

如果目标是 32 位通用寄存器，则源操作数应是 48 位内存操作数，指令执行后，内存操作数高 16 位送到隐含的段寄存器，低 32 位送到目标指定的通用寄存器。

本组指令的目的寄存器不允许使用段寄存器，不影响标志位。

【例 3-23】 设(DS)＝C000H，(C2480H)＝1357H，(C2482H)＝2468H

则执行指令LDS SI,[2480]后，(DS) = 2468H,(SI) = 1357H

【例 3-24】 设数据段：ADDR1　DF　1234567890ABH

则执行指令　LES　EBX,ADDR1 后,(ES) = 1234H,(EBX) = 567890ABH

4. 标志寄存器传送指令

标志寄存器传送指令用来传送标志寄存器 FLAGS 的内容,方便进行对各个标志位的直接操作。标志位传送指令有 4 条指令,即 LAHF、SAHF、PUSHF 和 POPF。

1) 读取标志指令 LAHF

语句格式：LAHF

其功能是将标志寄存器的低 8 位送入 AH 寄存器,即(AH)←$(FLAGS)_{7\text{-}0}$。该指令的执行对标志位无影响。

2) 设置标志指令 SAHF

语句格式：SAHF

其功能是将 AH 的内容送入标志寄存器的低 8 位,高 8 位不变,即$(FLAGS)_{7\text{-}0}$←(AH)。该指令用于设置或恢复 SF、ZF、AF、PF、CF 五个标志位,该指令的执行只影响标志寄存器的低 8 位,对高 8 位(即 OF、DF、IF、TF)标志位无影响。

从该指令的功能可看出,SAHF 为 LAHF 的逆过程。

3) 标志寄存器进栈指令 PUSHF/PUSHFD

格式：PUSHF
　　PUSHFD

其功能是将标志寄存器的内容压入堆栈。

执行的操作：

PUSHF：(SP)←(SP)－2
　　((SP)＋1,(SP))←(FLAGS)

PUSHFD：(ESP)←(ESP)－4

((ESP)＋3,(ESP)＋2,(ESP)＋1,(ESP))←(EFLAGS AND 0FCFFFFH)(清除 VM 和 RF 位)

4) 标志寄存器出栈指令 POPF/POPFD

语句格式：POPF
　　POPFD

其功能是将栈顶字单元内容弹出到标志寄存器中。该指令的执行影响标志位。

执行的操作：

POPF：(FLAGS)←((SP)＋1,(SP))
　　(SP)←(SP)＋2

POPFD：(EFLAGS)←((ESP)＋3,(ESP)＋2,(ESP)＋1,(ESP))
　　(ESP)←(ESP)＋4

PUSHF/PUSHFD 和 POPF/POPFD 互为逆过程。

注意：这组指令中的 LAHF 和 PUSHF/PUSHFD 不影响标志位。SAHF 和 POPF/POPFD 则由装入的值来确定标志位的值,将直接影响标志寄存器的内容,但 POPFD 不影响 VM、RF、IOPL、VIF 和 VIP 的值。利用这一特性,可以方便地改变标志寄存器中指定位的状态。

5. 类型转换指令

这组指令用来将字节转换为字,将字转换为双字,将双字转换为4字。

1) 字节扩展到字 CBW

语句格式:CBW

执行的操作:AL中的内容符号扩展到AH,形成AX中的字。即如果(AL)的最高有效位为0,则(AH)=0;如果(AL)的最高有效位为1,则(AH)=0FFH。

2) 字扩展到双字 CWD

语句格式:CWD

执行的操作:AX中的内容符号扩展到DX,形成DX:AX中的双字。即如果(AX)的最高有效位为0,则(DX)=0;如果(AX)的最高有效位为1,则(DX)=0FFFFH。

3) 字扩展到双字 CWDE

语句格式:CWDE

执行的操作:AX的内容符号扩展到EAX,形成EAX中的双字。

4) 双字扩展到4字 CDQ

语句格式:CDQ

执行的操作:EAX中的内容符号扩展到EDX,形成EDX:EAX中的4字。

上述指令的操作数总是在累加器中,通常用于除法之前产生双倍长的被除数。本组指令均不影响标志位。

3.3.2 算术运算指令

算术运算指令用来执行加、减、乘、除四则运算,其运算对象可以是8位、16位、32位的有符号或无符号数,另外还提供了十进制算术运算调整指令,使得运算对象可以是组合的BCD码数(即压缩的BCD码数)或非组合的BCD码数(即非压缩的BCD码数)。算术运算指令中有双操作数指令,也有单操作数指令。双操作数指令的两个操作数中除源操作数为立即数的情况外,必须有一个操作数在寄存器中,单操作数指令不允许使用立即数方式。这类指令执行后会影响标志寄存器中的状态标志。

1. 加法指令

加法指令包括ADD、ADC、INC和XADD四条指令,执行字节、字或双字的加法运算。

1) 加指令 ADD

语句格式:ADD DST,SRC

其功能是将目的操作数与源操作数相加,结果送到目的操作数,源操作数不变,即(DST)←(SRC)+(DST)。

【例 3-25】

```
ADD  AL,20H              ; AL的内容和20H相加,结果送到AL中
ADD  DI,SI               ; DI和SI的内容相加,结果送到DI中
ADD[BX+DI],AX            ; 数据段中有效地址为(BX+DI)的存储单元中的字和AX的内容相加,结
                         ; 果再送回到该存储单元
ADD  BYTE PTR [BX],12H   ; 数据段中有效地址为(BX)的存储单元中的字节和12H相加,结果送回
                         ; 该单元
ADD  EAX,ECX             ; EAX和ECX的内容相加,结果送到EAX中
```

ADD 指令按照状态标志的定义相应设置这些标志位。

如：ADD　DX,0F0F0H

若指令执行前，(DX) = 4652H

则指令执行后，(DX) = 3742H,ZF = 0,SF = 0,CF = 1,OF = 0,AF = 0,PF = 1

2）带进位加指令 ADC

语句格式：ADC　DST, SRC

其功能是将目的操作数加源操作数再加进位标志 CF，结果送目的操作数，即(DST)←(SRC)+(DST)+ CF。ADC 指令对状态标志的影响与 ADD 指令相同。

ADC 指令主要用于与 ADD 指令相结合实现多字节数相加。

【例 3-26】 实现两个无符号双字加法运算。设目的操作数存放在 DX 和 AX 寄存器中，其中 DX 存放高位字。源操作数存放在 BX、CX 中，其中 BX 存放高位字。

实现两个无符号双字相加的指令序列为：

```
ADD　AX,CX
ADC　DX,BX
```

若指令执行前(DX) = 0002H,(AX) = 0F365H,(BX) = 0005H,(CX) = 0E024H

则第一条指令执行后，(AX) = 0D389H,SF = 1,ZF = 0,CF = 1,OF = 0

第二条指令执行后，(DX) = 0008H,SF = 0,ZF = 0,CF = 0,OF = 0

3）加 1 指令 INC

语句格式：INC　OPR

其功能是将目的操作数加 1，结果送目的操作数，即(OPR)←(OPR)+1。

INC 指令是一个单操作数指令，操作数可以是寄存器或存储器操作数。

【例 3-27】

```
INC BX                        ; 将 BX 中的内容加 1
INC BYTE PTR [BX + DI + 500] ; 将数据段中有效地址为(BX + DI + 500)的存储单元中的字节内容加 1
```

设计加 1 指令和后面将要介绍的减 1 指令的目的，主要是用于对计数器和地址指针的调整，所以它们不影响进位标志 CF，对其他状态标志位的影响与 ADD 指令相同。

4）交换并相加指令 XADD

语句格式：XADD　DST, SRC

其功能是将源操作数和目的操作数互换，然后把源操作数和目的操作数之和送目的地址，即

```
TEMP←(SRC) + (DST)
(SRC)←(DST)
(DST)←TEMP
```

源操作数只能用寄存器寻址方式，目的操作数可用寄存器或任一种存储器寻址方式。该指令可做双字、字或字节运算。它对标志位的影响与 ADD 指令相同。

【例 3-28】

```
XADD　BL,CL
若指令执行前,(BL) = 12H,(CL) = 02H
```

则指令执行后,(BL) = 14H,(CL) = 12H

2. 减法指令

减法指令包括 SUB、SBB、DEC、NEG 和 CMP 五条指令,执行字节、字或双字的减法运算,除 DEC 不影响 CF 标志外,其他减法指令按定义影响全部状态标志位。

1) 减指令 SUB

语句格式:SUB DST,SRC

其功能是将目的操作数减源操作数,结果送到目的操作数,源操作数不变,即(DST)←(DST)－(SRC)。

【例 3-29】

```
SUB  BX,3340H              ; 将 BX 中的内容减去 3340H,结果送到 BX 中
SUB  BX,CX                 ; 将 BX 中的内容减去 CX 中的内容,结果送到 BX 中
SUB  [BP + 2],CL           ; 将堆栈段中(BP + 2)所指的单元中的值减去 CL 中的内容,结果再送回
                           ; 该单元
SUB  WORD PTR[DI],1000H    ; 将数据段中有效地址为(DI)的存储单元中的字减去 1000H,结果送回
                           ; 该单元
SUB  EAX,EBX               ; 将 EAX 中的内容减去 EBX 中的内容,结果送到 EAX 中
```

【例 3-30】

```
SUB DH,[BP + 4]
若指令执行前,(DH) = 41H,(SS) = 0000H,(BP) = 00E4H,(000E8H) = 5AH
则指令执行后,(DH) = 0E7H,SF = 1,ZF = 0,CF = 1,OF = 0
```

2) 带借位减指令 SBB

语句格式:SBB DST,SRC

其功能是将目的操作数减源操作数再减借位标志 CF,结果送目的地址,即(DST)←(DST)－(SRC)－CF。该指令主要用于与 SUB 指令相结合实现多字节数相减。例如:

```
SBB  AX,2030H              ; 将 AX 的内容减去立即数 2030H,并减去进位标志 CF 的值,结果送
                           ; 给 AX
SBB  WORD PTR [DI + 2],1000H  ; 将数据段中有效地址为(DI + 2)的存储单元中的字减去 1000H,并
                           ; 减去 CF,结果送回该单元
```

【例 3-31】 实现两个无符号双字减法运算。设目的操作数存放在 DX 和 AX 寄存器中,其中 DX 存放高位字。源操作数存放在 BX、CX 中,其中 BX 存放高位字。

实现两个无符号双字相减的指令序列为:

```
SUB  AX,CX
SBB  DX,BX

若指令执行前(DX) = 0234H,(AX) = 4652H,(BX) = 0F0F0H,(CX) = 0F0F0H
则第一条指令执行后,(AX) = 5562H,SF = 0,ZF = 0,CF = 1,OF = 0
第二条指令执行后,(DX) = 1143H,SF = 0,ZF = 0,CF = 1,OF = 0
```

3) 减 1 指令 DEC

语句格式:DEC OPR

其功能是将目的操作数减 1,结果送目的地址,即(OPR)←(OPR)－1。

DEC 指令是一个单操作数指令，操作数可以是寄存器或存储器操作数。与 INC 指令一样，DEC 指令不影响 CF，但影响其他状态标志。例如：

```
DEC  CX                          ; 将 CX 的内容减 1,再送回 CX 中
DEC  BYTE PTR [DI + 2]           ; 将数据段中有效地址为(DI + 2)的存储单元中的字节; 数据减 1,
                                 ; 结果送回此单元
```

DEC 指令一般用于对计数器和地址指针的调整。

4）求补指令 NEG

语句格式：NEG　OPR

其功能是对目的操作数执行求补运算，即用零减去目的操作数，然后将结果返回目的操作数，即(OPR)←0－(OPR)。求补运算也可以表达为：将目的操作数按位取反后加 1。NEG 指令对标志位的影响与用零做减法的 SUB 指令一样。例如，NEG 指令对 CF 和 OF 的影响：只有当操作数为 0 时，求补运算的结果使 CF＝0，其他情况均为 1；只有当字节运算时对－128 求补，以及字运算时对－32 768 求补和双字运算时对-2^{31}求补的情况下 OF＝1，其他则均为 0。

【例 3-32】 求补运算。

```
MOV AX,0FF64H
NEG AL                    ; (AX) = 0FF9CH,OF = 0,CF = 1,SF = 1,ZF = 0
SUB AL,9DH                ; (AX) = 0FFFFH,OF = 0,CF = 1,SF = 1,ZF = 0
NEG AX                    ; (AX) = 0001H,OF = 0,CF = 1,SF = 0,ZF = 0
DEC AL                    ; (AX) = 0000H,OF = 0,CF = 1,SF = 0,ZF = 1
NEG AX                    ; (AX) = 0000H,OF = 0,CF = 0,SF = 0,ZF = 1
```

5）比较指令 CMP

语句格式：CMP　OPR1,OPR2

该指令将目的操作数减去源操作数，结果只影响标志位，但不回送目的操作数，即(OPR1)－(OPR2)。

CMP 指令用于比较两个操作数的大小关系。执行比较指令之后，可以根据标志判断两个数是否相等、大小关系等。所以，CMP 指令后面常跟条件转移指令，根据比较结果不同产生不同的分支。另外，CMP 指令的操作数与 ADD/ADC、SUB/SBB 指令都一样。

【例 3-33】 比较 AL 的内容数值大小。

```
      CMP  AL,100        ; (AL) - 100
      JB   BELOW         ; 若(AL)＜100,则转到 BELOW 处执行
      SUB  AL,100        ; 若(AL)≥100,则(AL)←(AL) - 100
      INC  AH            ; (AH)←(AH) + 1
BELOW: …
```

3. 乘法指令

乘法指令用来实现两个二进制操作数的相乘运算，包括两条指令：无符号数乘法指令 MUL 和有符号数乘法指令 IMUL。

1）无符号数乘法指令 MUL

语句格式：MUL　SRC

执行的操作：

字节乘法：(AX)←(AL)＊(SRC)

字乘法：(DX,AX)←(AX)＊(SRC)

双字乘法：(EDX,EAX)←(EAX)＊(SRC)

2) 有符号数乘法指令 IMUL

语句格式：IMUL SRC

执行的操作：与 MUL 相同，但必须是带符号数，而 MUL 是无符号数。

在乘法指令中，目的操作数必须是累加器，字节运算为 AL，字运算为 AX，双字运算为 EAX。若是字节数据相乘，(AL)与 SRC 相乘得到字数据存入 AX 中；若是字数据相乘，则(AX)与 SRC 相乘得到双字数据，高字存入 DX、低字存入 AX 中；若是双字相乘，则(EAX)与 SRC 相乘得到 4 字数据，高位存入 EDX、低位存入 EAX 中。指令中的源操作数可以使用除立即数方式以外的任何一种寻址方式。

MUL 指令和 IMUL 指令的使用条件是由数的格式决定的。(11111111b)＊(11111111b)当把它看作无符号数时应为 255d×255d=65025d；而看作带符号数时则为(－1)×(－1)=1。所以，必须根据所要相乘数的格式来决定选用哪一种指令。

乘法指令对除 CF 位和 OF 位以外的状态标志无定义，即成为任意，不可预测。注意，无定义和对标志位没有影响是不同的，没有影响是指不改变原来的状态。

对于 MUL 指令，若乘积高一半为 0，即字节操作的(AH)或字操作的(DX)或双字操作的(EDX)为 0，则 CF 位和 OF 位均为 0；否则，CF 位和 OF 位均为 1。这样可以用来检查字节相乘的结果是字节还是字，或者可以检查字相乘的结果是字还是双字，双字相乘的结果是双字还是 4 字。对于 IMUL 指令，若乘积高一半是低一半的符号扩展，则 CF 位和 OF 位均为 0；否则，CF 位和 OF 位均为 1。

【例 3-34】 若(AL)＝0B4H，(BL)＝11H，求执行指令 IMUL BL 和 MUL BL 后的乘积值。

(AL)＝0B4H 为无符号数的 180D，带符号数的－76D；(BL)＝11H 为无符号数的 17D，带符号数的 17D。

则执行 IMUL BL 的结果为：(AX)＝0FAF4H＝－1292D，CF＝OF＝1

执行 MUL BL 的结果为：(AX)＝0BF4H＝3060D，CF＝OF＝1

另外，IMUL 指令除上述的单操作数指令(累加器是隐含的)外，还有双操作数和三操作数指令格式：

格式：IMUL　REG,SRC

执行的操作：

字操作数：(REG16)←(REG16)＊(SRC)

双字操作数：(REG32)←(REG32)＊(SRC)

其中，目的操作数必须是 16 位或 32 位寄存器，而源操作数则可用任一种寻址方式取得和目的操作数长度相同的数；但当源操作数为立即数时，除相应地用 16 位或 32 位立即数外，指令中也可指定 8 位立即数，在运算时机器会自动把概数符号扩展成与目的操作数长度相同的数。

格式：IMUL　REG,SRC,IMM

执行的操作：

字操作数：(REG16)←(SRC) * IMM

双字操作数：(REG32)←(SRC) * IMM

其中，目的操作数必须是 16 位或 32 位寄存器，源操作数可用除立即数以外的任一种寻址方式取得和目的操作数长度相同的数；IMM 表示立即数，它可以是 8、16 或 32 位数，但其长度必须和目的操作数一致，当长度为 8 位时，运算时将符号扩展成与目的操作数长度一致的数。

IMUL 的双操作数和三操作数指令格式与单操作数指令有一个很大的区别：后者的乘积字长是源操作数和目的操作数字长的两倍，因而即使 OF 位为 1，乘积结果也是正确的，不存在溢出问题。但前两种格式乘积的字长和源操作数与目的操作数的字长是一致的，即字操作数相乘得到的乘积也是字，双字操作数相乘得到的乘积也是双字，这样就可能产生溢出。机器规定：对 IMUL 的双操作数和三操作数指令，16 位操作数相乘得到的乘积在 16 位之内或 32 位操作数相乘得到的乘积在 32 位之内时，CF 位和 OF 位均置 0，否则置 1，这时的 OF 位为 1 说明是溢出的。其他标志位也是无定义的。

例如：

```
IMUL   DX,WORD                  ; DX 中的内容乘以 WORD 单元中的 16 位字,结果送到 DX 中
IMUL   EAX,ARRAY[ESI * 4],8     ; 根据存储器寻址方式从 ARRAY 中取出相应单元的 32 位字乘以 8,
                                ; 结果送到 EAX 中
```

4. 除法指令

除法指令执行两个二进制数的除法运算，包括无符号除法指令 DIV 和有符号除法指令 IDIV 两条指令。

1) 无符号除法指令 DIV

语句格式：`DIV  SRC`

执行的操作：

字节除法：16 位被除数在 AX 中，8 位除数为源操作数，结果的 8 位商在 AL 中，8 位余数在 AH 中。表示为：

(AL)←(AX)/(SRC)的商

(AH)←(AX)/(SRC)的余数

字除法：32 位被除数在 DX、AX 中。其中，DX 为高位字；16 位除数为源操作数，结果的 16 位商在 AX 中，16 位余数在 DX 中。表示为：

(AX)←(DX,AX)/(SRC)的商

(DX)←(DX,AX)/(SRC)的余数

双字除法：64 位被除数在 EDX、EAX 中。其中，EDX 为高位双字；32 位除数为源操作数，结果的 32 位商在 EAX 中，32 位余数在 EDX 中。表示为：

(EAX)←(EDX,EAX)/(SRC)的商

(EDX)←(EDX,EAX)/(SRC)的余数

商和余数均为无符号数。

2) 有符号除法指令 IDIV

语句格式：`IDIV  SRC`

执行的操作：与 DIV 相同，但操作数必须是带符号数，商和余数也都是带符号数，并且余数的符号和被除数的符号相同。

除法指令的寻址方式和乘法指令相同。其目的操作数必须存放在 AX 或(DX,AX)或(EDX,EAX)中；而其源操作数可以用除立即数以外的任一种寻址方式。

除法指令对所有标志位均无定义。

由于除法指令的字节操作要求被除数为 16 位，字操作要求被除数为 32 位，双字操作要求被除数为 64 位，因此往往需要用符号扩展的方法取得除法指令所需要的被除数格式。在使用除法指令时，还要注意一个问题，除法指令要求字节操作时商为 8 位，字操作时商为 16 位，双字操作时商为 32 位。如果字节操作时，被除数的高 8 位绝对值≥除数的绝对值；或者字操作时，被除数的高 16 位绝对值≥除数的绝对值；或者双字操作时，被除数的高 32 位绝对值≥除数的绝对值，则商就会产生溢出。在 IA32 中这种溢出是由系统直接转入 0 号中断处理的。为避免出现这种情况，必要时程序应进行溢出判断及处理。

【例 3-35】 设(AX)＝0400H，(BL)＝0B4H，求执行指令 DIV BL 和 IDIV BL 后的结果。

(AX)为无符号数的 1024D，带符号数的＋1024D；(BL)为无符号数的 180D，带符号数的－76D。

执行 DIV BL 的结果是：(AH)＝7CH＝124D　余数

(AL)＝05H＝5D　商

执行 IDIV BL 的结果是：(AH)＝24H＝36D　余数

(AL)＝0F3H＝－13D　商

【例 3-36】 算术运算综合举例。编写程序段，计算：$(C-120+A*B)/C$，并把所得的商和余数分别存入 X 和 Y 中(其中，A、B、C、X 和 Y 都是有符号的自变量)。

```
…
A DW ?
B DW ?
C DW ?
X DW ?
Y DW ?
…
MOV  AX, C
SUB  AX, 120
CWD
MOV  CX, DX
MOV  BX, AX    ; (CX, BX)←(DX, AX),调度寄存器,为做乘法准备必要的寄存器
MOV  AX, A
IMUL B         ; (DX, AX)←A * B
ADD  AX, BX    ; 计算 32 位二进制之和,为做除法准备
ADC  DX, CX
IDIV C         ; AX 是商,DX 是余数
MOV  X, AX     ; 分别保存商和余数到指定的字变量单元里
MOV  Y, DX
…
```

5. 十进制调整指令

前面介绍的算术运算指令都是针对二进制数的，然而，十进制是人们日常使用的进制。计算机进行十进制算术运算时，先将十进制数转换为二进制数，然后做二进制数算术运算，再将结果转换为十进制数输出。为了便于十进制数的计算，计算机提供了一组十进制算术运算调整指令，用于将运算后的二进制数调整为 BCD 码。

为了理解调整指令的工作原理，表 3-2 列出了用二进制加法指令完成十进制数运算的调整原则。

表 3-2　十进制数加法调整

笔　　算	CPU 运算	加 法 调 整
43 ＋ 55 98	0100 0011 ADD) 0101 0101 1001 1000	CF＝0，AF＝0，高低 4 位均没有出现非法 BCD 码，结果正确，不需要修正
39 ＋ 49 88	0011 1001 ADD) 0100 1001 1000 0000 ＋) 0110 1000 1000	低 4 位有进位，即 AF＝1，对运算结果加 06H 修正
63 ＋ 54 117	0110 0011 ADD) 0101 0100 1011 0111 ＋) 0110 1 0001 0111	高 4 位出现非法 BCD 码数，对运算结果加 60H 修正
87 ＋ 86 173	1000 0111 ADD) 1000 0110 1 0000 1101 ＋) 0110 0110 1 0111 0011	因为 CF＝1，低 4 位出现非法 BCD 码，对运算结果加 66H 修正

从表中列举的部分例题可以看出：用二进制加法指令完成十进制数运算，结果往往是错误的，因为十进制数(BCD 码数)加法法则是“逢十进一”，即低 4 位大于 1001 时，就应当向第 4 位(设最低位为第 0 位)“进一”。但是二进制数加法指令遵循“逢二进一”的法则，只有当低 4 位大于 16 时，才向第 4 位“进一”，两种数制的进位值相差为 6，因此造成错误。另外，在二进制数数列中出现 1010～1111 是有意义的，而在 BCD 码数列中出现 1010～1111 是非法 BCD 码。

所以，当使用二进制运算指令进行十进制数运算时，必须对结果进行有条件的修正，这种修正不需要程序员去做，IA32 专门设置了一套调整指令，调整指令会自动对运算结果进行修正。

注意：十进制调整指令并非用来进行运算，它不能单独使用，必须与加、减、乘、除二进制指令配合使用才能进行十进制调整，十进制调整指令形式上均为无操作数指令，其操作对象隐含在 AX 中。该类指令分为压缩 BCD 码调整指令和非压缩 BCD 码调整指令。

1) 压缩 BCD 码调整指令

压缩 BCD 码是通常的 8421 码,它用 4 个二进制位表示一个十进制位,1 个字节可以表示两个十进制位,即 00～99。该组指令包括加法和减法的十进制调整指令 DAA 和 DAS,它们用来对二进制加、减法指令的执行结果进行调整,得到十进制结果。必须注意的是,在使用 DAA 或 DAS 指令之前,应先执行加法或减法指令。

① 加法的十进制调整指令 DAA

语句格式:DAA

其功能是将 AL 中的和调整为压缩 BCD 码并送回 AL。本指令执行之前必须先执行 ADD 或 ADC 指令,把两个压缩 BCD 码相加,并把和存放在 AL 寄存器中。具体调整方法如下:

若(AL)的低 4 位是十六进制的 A～F 或 AF=1,则:AL←(AL)+06H,并使 AF=1;

若(AL)的高 4 位是十六进制的 A～F 或 CF=1,则:AL←(AL)+60H,并使 CF=1。

其余情况 AL 内容不变。

说明:DAA 指令对 OF 无定义,但影响所有其他状态标志位。

【例 3-37】 压缩 BCD 码的加法运算。

```
MOV  AL,68H          ; (AL) = 68H,表示压缩 BCD 码 68
MOV  BL,28H          ; (BL) = 28H,表示压缩 BCD 码 28
ADD  AL,BL           ; 二进制加法: (AL) = 68H + 28H = 90H
DAA                  ; 十进制调整: 因 AF = 1 需做 AL←(AL) + 06H,
                     ; 得(AL) = 96H,CF = 0,AF = 1
```

② 减法的十进制调整指令 DAS

语句格式:DAS

其功能是将 AL 中的差调整为压缩 BCD 码并送回 AL。本指令执行之前必须先执行 SUB 或 SBB 指令,把两个压缩 BCD 码相减,并把差存放在 AL 寄存器中。具体调整方法如下:

若(AL)的低 4 位是十六进制的 A～F 或 AF=1,则:AL←(AL)－06H,并使 AF=1;

若(AL)的高 4 位是十六进制的 A～F 或 CF=1,则:AL←(AL)－60H,并使 CF=1。

其余情况 AL 内容不变。

说明:DAS 指令对 OF 无定义,但影响所有其他状态标志位。

【例 3-38】 压缩 BCD 码的减法运算。

```
MOV  AL,86H          ; (AL) = 86H,表示压缩 BCD 码 86
MOV  BL,07H          ; (BL) = 07H,表示压缩 BCD 码 07
SUB  AL,BL           ; 二进制减法: (AL) = 86H - 07H = 7FH
DAS                  ; 十进制调整: 因 AF = 1 需做 AL←(AL) - 06H,
                     ; 得(AL) = 79H,CF = 0,AF = 1
```

2) 非压缩 BCD 码调整指令

非压缩 BCD 码用 8 个二进制位表示一个十进制位,实际上只是用低 4 个二进制位表示一个十进制位 0～9,高 4 位任意,但通常默认为 0。ASCII 码中 0～9 的编码是 30H～39H,所以 0～9 的 ASCII 码(高 4 位变为 0)就可以认为是非压缩 BCD 码。

对非压缩 BCD 码，有 AAA、AAS、AAM 和 AAD 四条指令，分别用于对二进制加、减、乘、除指令的结果进行调整，以得到非压缩 BCD 码表示的十进制数结果。由于只要在调整后的结果中加上 30H 就成为 ASCII 码，所以这组指令实际上也是针对 ASCII 码的调整指令的。

① 加法的 ASCII 调整指令 AAA

语句格式：AAA

其功能是将 AL 中的和调整为非压缩 BCD 码并送回 AL。在使用 AAA 指令前，必须执行 ADD 或 ADC 指令把非压缩 BCD 相加，且把和存放在 AL 中。具体调整方法如下：

若二进制相加后(AL)的低 4 位是十六进制的 A～F 或 AF＝1，则：(AL)←(AL)＋6；(AH)←(AH)＋1；使 AF＝CF＝1 且 AL 高 4 位清零。

否则使 AF＝CF＝0 且 AL 高 4 位清零。

AAA 指令除影响 AF 和 CF 标志外，对其他标志位均无定义。

【例 3-39】 非压缩 BCD 码的加法运算。

```
MOV  AX,0535H        ; (AX) = 0535H,表示非压缩 BCD 码 55
MOV  BL,39H          ; (BL) = 39H,表示非压缩 BCD 码 9
ADD  AL,BL           ; 二进制加法：(AL) = 35H + 39H = 6EH
AAA                  ; 因(AL)的低 4 位> 9,需做 ASCII 调整,调整的结果使(AX) = 0604H,CF = AF = 1
```

② 减法的 ASCII 调整指令 AAS

语句格式：AAS

其功能是将 AL 中的差调整为非压缩 BCD 码并送回 AL，向高位的借位在 AH 和 CF 中。在使用 AAS 指令之前，必须执行 SUB 或 SBB 指令把两个非压缩的 BCD 码相减，并把差存放在 AL 中。具体调整方法如下：

若二进制相减后(AL)的低 4 位是十六进制的 A～F 或 AF＝1，则：(AL)←(AL)－6；(AH)←(AH)－1；使 AF＝CF＝1 且 AL 高 4 位清零。

否则使 AF＝CF＝0 且 AL 高 4 位清零。

AAS 指令除影响 AF 和 CF 标志外，对其他标志位均无定义。

【例 3-40】 非压缩 BCD 码的减法运算。

```
MOV  AX,0608H        ; (AX) = 0608H,表示非压缩 BCD 码 68
MOV  BL,09H          ; (BL) = 09H,表示非压缩 BCD 码 9
SUB  AL,BL           ; 二进制减法：(AL) = 08H - 09H = FFH
AAS                  ; 因(AL)的低 4 位> 9,需做 ASCII 调整,调整的结果使(AX) = 0509H,CF = AF = 1
```

③ 乘法的 ASCII 调整指令 AAM

语句格式：AAM

其功能是将 AL 中的积调整为非压缩 BCD 码并送回 AX。使用 AAM 指令之前必须执行 MUL 把两个非压缩的 BCD 码相乘(此时要求两个非压缩的 BCD 码的高 4 位必须为 0)，乘积放在 AL 中。具体调整方法如下：

把 AL 寄存器的内容除以 0AH，并把商放在 AH 寄存器中，余数放在 AL 寄存器中。

本指令根据 AL 寄存器的内容设置 SF、ZF 和 PF，对 OF、CF 和 AF 无定义。

【例 3-41】 非压缩 BCD 码的乘法运算。

```
MOV  AX,0608H                    ; (AX) = 0608H,表示非压缩 BCD 码 68
MOV  BL,09H                      ; (BL) = 09H,表示非压缩 BCD 码 9
MUL  BL                          ; 二进制乘法: (AX) = 08H × 09H = 0048H
AAM                              ; ASCII 调整的结果使(AX) = 0702H
```

④ 除法的 ASCII 调整指令 AAD

语句格式：AAD

其功能是在除法运算前，先调整被除数 AX 内容，将 AX 中的非压缩 BCD 码扩展成二进制数，即(AL)←(AH)×0AH＋(AL)，(AH)← 0。

AAD 调整指令与其他的调整指令应用情况不同。它是先将存放在 AX 寄存器中的两位非压缩 BCD 码数进行调整，然后再用 DIV 指令除以一个非压缩 BCD 码数，这样得到非压缩 BCD 码数的除法结果。其中，要求 AL、AH 和除数的高 4 位为 0。AAD 指令根据结果设置 SF、ZF 和 PF，对 OF、CF 和 AF 无定义。

【例 3-42】 非压缩 BCD 码的除法运算。

```
MOV  AX,0608H                    ; (AX) = 0608H,表示非压缩 BCD 码 68
MOV  BL,09H                      ; (BL) = 09H,表示非压缩 BCD 码 9
AAD                              ; 二进制扩展: (AX) = 68 = 0044H
DIV  BL                          ; 除法运算: 商(AL) = 07H,余数(AH) = 05H
```

3.3.3 逻辑指令

逻辑指令用来对二进制数的各个位进行操作，包括逻辑运算指令、位测试并修改指令、位扫描指令和移位指令。

1. 逻辑运算指令

逻辑运算指令用来对字节、字或双字按位进行逻辑运算，包括逻辑与 AND、逻辑或 OR、逻辑非 NOT、异或 XOR 和测试 TEST 五条指令。

1) 逻辑与指令 AND

语句格式：AND DST,SRC

其功能是将目的操作数和源操作数进行按位的逻辑与运算，结果送到目的操作数，即(DST)←(DST)∧(SRC)。

AND 指令常用于以下场合：

① 使一个操作数中的某些位保持不变，而某些位清 0。

② 某一操作数，自己和自己相“与”，操作数不变，但可以使进位标志 CF 清 0。

【例 3-43】 逻辑与运算。

```
AND  AL,77H                      ; 将 AL 中第 3 位和第 7 位清 0
AND  AX,BX                       ; 两个寄存器逻辑与
AND  AL,1111 0000B               ; 屏蔽 AL 寄存器低 4 位
AND  MEM - BYTE,AL               ; 存储单元和寄存器逻辑与
```

2) 逻辑或指令 OR

语句格式：OR DST,SRC

其功能是将目的操作数和源操作数进行按位的逻辑或运算，结果送到目的操作数，即(DST)←(DST)∨(SRC)。

OR 指令常用于以下场合：

① 使一个操作数中的若干位保持不变，而另外若干位置 1 的场合。这时，要保持不变的这些位与 0 相或；而要置 1 的这些位与 1 相或。

② 某一操作数，自己和自己相“或”，操作数不变，但可以使进位标志 CF 清 0。

【例 3-44】 逻辑或运算。

```
OR  AL,88H                    ; 将 AL 寄存器中第 3 位和第 7 位置 1
OR  BX,0C000H                 ; 将 BX 中第 15 位和第 14 位置 1
```

3）逻辑非指令 NOT

语句格式：NOT OPR

其功能是将目的操作数按位取反，即(OPR)←～(OPR)(符号～表示逻辑反)。

NOT 指令改变寄存器或存储单元的每一位状态，原来为 0 变为 1，原来为 1 变为 0。

【例 3-45】 逻辑非运算。

```
MOV  AX,878AH                 ; (AX) = 878AH
NOT  AX                       ; (AX) = 7875H,标志不变
```

4）异或指令 XOR

语句格式：XOR DST,SRC

其功能是将目的操作数与源操作数做按位的异或运算，结果送到目的操作数，即(DST)←(DST)＋(SRC)。

XOR 指令常用于以下场合：

① 使一个操作数中的若干位保持不变，而另外若干位取反。

② 使某一操作数清 0。

③ 测试某一操作数是否与另一确定的操作数相等。这种操作在检查地址是否匹配时是常用的。

【例 3-46】 异或运算。

```
MOV  AL,45H                   ; (AL) = 45H
XOR  AL,31H                   ; (AL) = 74H
XOR  BX,0C000H                ; 将 BX 的第 12 位和 13 位取反,其余位不变
XOR  AX,AX                    ; 将 AX 清 0
```

5）测试指令 TEST

语句格式：TEST OPR1,OPR2

其功能是将两个操作数执行按位的逻辑与运算，但结果不送入目的操作数，即(OPR1)∧(OPR2)。

TEST 指令常用于在不希望改变原有操作数的情况下，用来检测某一位或某几位的条件是否满足。编程时常与条件转移指令一起使用，可在 TEST 指令后面加上条件转移指令，来测试操作数某位是否为 1，或者是否为 0。

【例 3-47】 测试 AX 中的第 12 位是否为 0，不为 0 则转 L。

```
TEST  AX,1000H
JNE  L
```

说明：在以上五种指令中，NOT 指令不允许使用立即数，其他 4 条指令除非源操作数是立即数，否则至少有一个操作数必须在寄存器中，另一个则可以使用任意寻址方式。它们对标志位的影响：NOT 指令不影响标志位，其他 4 种指令将使 CF 和 OF 为 0，AF 无定义，SF、ZF 和 PF 则根据运算结果设置。

2. 位测试并修改指令

本组指令包括位测试指令 BT、位测试并置 1 指令 BTS、位测试并置 0 指令 BTR 和位测试并变反指令 BTC。

1）位测试指令 BT

语句格式：`BT  DST,SRC`

2）位测试并置 1 指令 BTS

语句格式：`BTS  DST,SRC`

3）位测试并置 0 指令 BTR

语句格式：`BTR  DST,SRC`

4）位测试并变反指令 BTC

语句格式：`BTC  DST,SRC`

这 4 条指令相同的功能是：测试目的操作数中由源操作数指定的那一位，将测试位的值送 CF。如果源操作数大于等于目的操作数字长，则源操作数除以目的操作数字长之后，其余数才是测试位。指令执行后，源操作数不变。

这 4 条指令不同之处在于：BT 执行之后，目的操作数不变，而 BTS、BTR、BTC 执行后，测试位分别被置 1、置 0、取反。

【例 3-48】 位测试并修改。

```
BT    AX,0     ; AX 的第 0 位送 CF
BT    AX,16    ; AX 的第 0 位送 CF
BTR   EAX,31   ; EAX 的第 31 位送 CF,然后将 EAX 的第 31 位置 0
MOV   EAX,48
BTS   EBX,EAX  ; EBX 的第 16 位送 CF,然后将 EAX 的第 16 位置 1,EAX 不变
```

3. 位扫描指令

1）正向位扫描 BSF

语句格式：`BSF  REG,SRC`

操作：从源操作数的最低位开始向高位搜索，将遇到的第一个 1 所在的位序号存入目的寄存器中，并将 ZF 置 0；若源操作数为 0，则将 ZF 置 1，目的寄存器内容不变。

2）反向位扫描 BSR

语句格式：`BSR  REG,SRC`

操作：从源操作数的最高位开始向低位搜索，将遇到的第一个 1 所在的位序号存入目的寄存器中，并将 ZF 置 0；若源操作数为 0，则将 ZF 置 1，目的寄存器内容不变。

说明：本组指令其源操作数只能是 16 位、32 位的寄存器操作数或内存操作数，目的寄

存器必须与源操作数等长。指令执行后，源操作数不变。

【例 3-49】 位扫描操作。

```
MOV    EAX,23456780H
BSF    EBX,EAX              ; (EBX) = 7,EAX 不变,ZF = 0
BSR    BX,AX                ; (BX) = 14,AX 不变,ZF = 0
```

4. 移位指令

移位指令包括 10 条指令，可分成 3 组：非循环移位指令、循环移位指令和双精度移位指令。其功能为将目的操作数的所有位按操作符规定的方式移动 1 位，或按寄存器 CL 规定的次数，结果送入目的地址。

1）非循环移位指令

逻辑左移 SHL：`SHL  OPR,CNT`

逻辑右移 SHR：`SHR  OPR,CNT`

算术左移 SAL：`SAL  OPR,CNT`

算术右移 SAR：`SAR  OPR,CNT`

2）循环移位指令

循环左移 ROL：`ROL  OPR,CNT`

循环右移 ROR：`ROR  OPR,CNT`

带进位循环左移 RCL：`RCL  OPR,CNT`

带进位循环右移 RCR：`RCR  OPR,CNT`

其中，CNT 为移位次数，它可以是立即数或者是预先存放在 CL 寄存器中的移位次数。以上 8 条移位指令，其运算对象可以为 8 位、16 位或 32 位的寄存器操作数或内存操作数。如果操作数是内存操作数，则必须用 PTR 运算符说明它的属性是字节型、字型还是双字型。这 8 条指令执行的操作如图 3-4 所示。

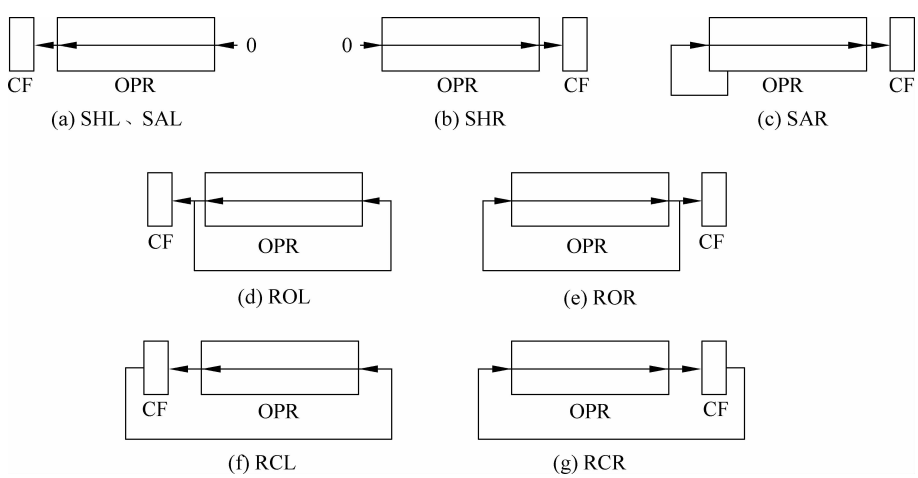


图 3-4　移位指令的操作

逻辑左移 SHL 和算术左移 SAL 实际上是同一条指令的两种助记符形式，两者完全相同，建议采用 SHL。在指令系统中还有类似的情况，采用多个助记符只是为了方便使用，增加可读性。

以上两组指令对标志位的影响是：CF 位根据各条指令的规定设置。OF 位只有当 CNT＝1 时才是有效的，否则该位无定义。当 CNT＝1 时，在移位后最高有效位的值发生变化时(即原来为 0，移位后为 1；或原来为 1，移位后为 0)OF 位置 1，否则置 0。循环移位指令不影响除 CF 和 OF 以外的其他状态标志位。而非循环移位指令则根据移位后的结果设置 SF、ZF 和 PF 位，AF 位则没有定义。

循环移位指令可以改变操作数中所有位的位置，在程序中很有用。非循环移位指令则常用来做乘以 2 或除以 2 的操作。其中，逻辑移位指令适用于无符号数运算，SHL 执行一次移位，相当于无符号数乘 2；SHR 执行一次移位，相当于无符号数除以 2，商在目的操作数中，余数由 CF 标志位反映。而算术移位指令适用于有符号数运算，SAR 执行一次移位，相当于有符号数除以 2，但是当操作数为负(最高位为 1)，并且最低位有 1 移出时，SAR 指令产生的结果与等效的 IDIV 指令的结果不同。例如，－5 经 SAR 右移一位等于－3，而 IDIV 指令执行(－5)÷2 的结果为－2。

【例 3-50】 若(DS)＝0F800H，(DI)＝1800H，(BX)＝5451H，(0F9800H)＝0064H

```
MOV   CL, 5
SAR   WORD PTR [DI], CL     ; (OF9800H) = 0003H,CF = 0
MOV  CL, 2
SHL   BX, CL                ; (BX) = 5144H,CF = 1
```

【例 3-51】 已知(BX)＝84F0H

(1) 若(BX)为无符号数，求(BX)÷2。

```
SHR   BX, 1    ; (BX) = 4278H
```

(2) 若(BX)为有符号数，求(BX)×2。

```
SAL   BX, 1    ; (BX) = 09E0H, OF = 1
```

(3) 若(BX)为有符号数，求(BX)÷4。

```
MOV   CL, 2
SAR   BX, CL   ; (BX) = 0E13CH
```

【例 3-52】 若(AX)＝0012H，(BX)＝0034H，把它们装配成(AX)＝1234H。

```
MOV   CL, 8
ROL   AX, CL
ADD   AX, BX
```

【例 3-53】 将两个非压缩 BCD 码(高位在 BL，低位在 AL)合并成压缩 BCD 码送 AL。

```
MOV   CL,4     ; 将计数值送 CL
SHL   BL,CL    ; 将高位移到 BL 的高 4 位
AND   AL,0FH   ; 清零 AL 高 4 位
OR    AL,BL    ; 合并 AL 和 BL 形成压缩 BCD 码
```

【例 3-54】 (BX)＝84F0H，把(BX)中的 16 位数每 4 位压入堆栈。

```
    MOV    CH, 4     ; 循环次数
    MOV    CL, 4     ; 移位次数
```

```
NEXT:
    ROL     BX, CL
    MOV     AX, BX
    AND     AX, 0FH
    PUSH    AX
    DEC     CH
    JNZ     NEXT
```

3）双精度移位指令

① 双精度左移指令 SHLD

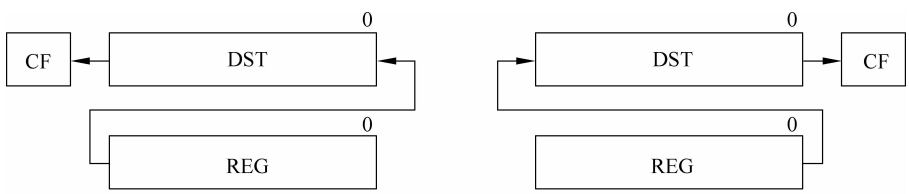
语句格式：SHLD　DST,REG,CNT

② 双精度右移指令 SHRD

语句格式：SHRD　DST,REG,CNT

这是一组三操作数指令，其中 DST 可用除立即数以外的任一种寻址方式指定字或双字操作数，源操作数为与目的操作数等长的寄存器寻址方式，CNT 为移位次数，可以是一个 8 位的立即数或 CL。移位计数值的范围为 1～31，若大于 31 则取模 32 的值来取代。

执行的操作如图 3-5 所示。



(a) 双精度左移SHLD　　(b) 双精度右移SHRD

图 3-5　双精度移位指令的操作

这组指令可以取两个字做移位操作而得到的是一个字的结果，也可以取两个双字做移位操作而得到的是一个双字的结果。在移位中，作为源操作数的寄存器提供移位值，以补目的操作数因移位引起的空缺，而指令执行完后，只取目的操作数作为移位的结果，源操作数寄存器则保持指令执行前的值不变。当移位次数为 0 时，不影响标志位；否则，根据移位后的结果设置 SF、ZF、PF 和 CF 值。OF 的设置情况是：当移位次数为 1 时，如移位后引起符号位改变则 OF 位为 1，否则为 0；当移位次数大于 1 时 OF 位无定义。AF 位除移位次数为 0 外均无定义。

【例 3-55】　SHLD　EBX,ECX,16

若指令执行前，(EBX)＝12345678H，(ECX)＝13572468H；

则指令执行后，(EBX)＝56781357H，(ECX)＝13572468H，CF＝0。

3.3.4　串处理指令

计算机中的大部分数据是存放在内存中的，前面介绍的数据传送类指令每次只能传送一个数据，若要传送大批量数据就需要重复编程，这样就浪费了大量的时间和空间。为此，IA32 提供了一组处理内存中连续存放的数据串指令，这就是串处理指令。串处理指令就是用一条指令实现对一串字符或数据的操作，如传送、比较、搜索及赋值等。

串处理指令中，源串和目的串的存储及寻址方式都有隐含的规定：源串默认在数据段DS中，但允许段超越；目的串默认在附加段ES中，不允许段超越。16位寻址时，源串首址存放于SI，目的串首址存放于DI，用CX作为串计数器记录串的长度；32位寻址时，源串首址存放于ESI，目的串首址存放于EDI，用ECX作为串计数器。由于实地址模式下逻辑段的最大容量为64KB，没有必要使用32位寻址，为了描述方便，在介绍指令功能时，均以16位寻址为基础。

串处理时，地址的修改往往与方向标志DF有关，当DF＝1时，SI和DI做自动减量修改；当DF＝0时，SI和DI做自动增量修改。在同一个段内实现串传送时，应将数据段基址和附加段基址设置成同一数值，即(DS)＝(ES)，此时，仍由SI和DI分别指出源串和目的串的首地址。串处理指令是唯一的一组源操作数和目的操作数都在存储单元的指令。

串处理指令包括6条指令，分别是串传送MOVS、存入串STOS、从串取LODS、串比较CMPS、串扫描SCAS、串输入INS/串输出OUTS。

串处理指令执行一次，仅对数据串中的一个字节、字或双字进行操作。但在串处理指令之前，都可以加一个重复前缀，实现串操作的重复执行。重复次数隐含在CX寄存器中。

重复前缀指令用来控制紧跟其后的字符串指令是否重复，为单字节指令，不能单独使用。在介绍串处理指令之前，先对重复前缀做一下说明。

1. 重复前缀

重复前缀可分为两类，包括三条指令：一类是配合不影响标志的MOVS、STOS、LODS及INS/OUT指令的REP前缀；另一类是配合影响标志的CMPS、SCAS指令的REPZ和REPNZ前缀。

1) REP重复前缀

语句格式：`REP String Primitive`

其中，String Primitive可以是MOVS、STOS、LODS或INS/OUT指令。

功能：每执行一次串指令，CX中的值减1，直到(CX)＝0，重复执行结束，它的意义相当于“数据串未处理完((CX)≠0)，则继续传送”。

2) REPZ/REPE重复前缀

语句格式：`REPZ/REPE  String Primitive`

其中，String Primitive可以是CMPS或SCAS指令。

功能：每执行一次串指令，CX中的值减1，并判断ZF标志是否为0，只要(CX)＝0或ZF＝0，则重复执行结束。

本指令只有在同时满足如下两个条件时，才重复进行扫描和比较：

① (CX)≠0，表示重复次数还未满。

② ZF＝1，表示目的操作数等于源操作数(CMPS指令)或等于扫描值(SCAS指令)。

反过来也就是说，两个条件中只要有一个不满足(或者重复次数已满，或者是目的操作数不等于源操作数或扫描值)，则重复停止，串指令结束。它的意义相当于“数据串未处理完((CX)≠0)，并且串相等(ZF＝1)，则继续执行”。

3) REPNE/REPNZ前缀

语句格式：`REPNE/REPNZ  String Primitive`

其中，String Primitive可以是CMPS或SCAS指令。

功能：每执行一次串指令，CX 中的值减 1，并判断 ZF 标志是否为 0，只要(CX)＝0 或 ZF＝1，则重复执行结束。

如果在 CMPS 和 SCAS 指令前使用 REPNE/REPNZ 前缀，则只有在同时满足如下两个条件时，才重复进行扫描或比较：CX≠0；ZF＝0。它的意义相当于“数据串未处理完((CX)≠0)，并且串不相等(ZF＝0)，则继续执行”。

2. 串传送指令 MOVS

语句格式：

字节传送：MOVSB

字传送：MOVSW

双字传送：MOVSD

该指令把数据段的 1 个字节或字或双字传送至附加段的内存单元中，即执行的操作为：

① ((DI))←((SI))

② 字节传送：(SI)←(SI)±1，(DI)←(DI)±1

字传送：(SI)←(SI)±2，(DI)←(DI)±2

双字传送：(SI)←(SI)±4，(DI)←(DI)±4

在上述操作中，当方向标志 DF＝0 时用＋，DF＝1 时用－。

除了这三种格式外，MOVS 指令还有另外一种格式：

```
MOVS DST,SRC
```

这种格式增加了可读性，但要求两个串名类型一致，应在操作数中表明是字节、字还是双字操作。例如：

```
MOVS ES:BYTE PTR[DI], DS:[SI]
```

实际上 MOVS 的寻址方式是隐含的，所以这种格式中的 DST 及 SRC 只提供给汇编程序作类型检查用，并且不允许用其他寻址方式来确定操作数。其他串处理指令同样具有这种格式，后面不再介绍。MOVS 指令不影响标志位。

MOVS 指令本身只能传送一个字节或一个字或一个双字，要传送整个数据串，还需要重复执行这个指令。

【例 3-56】 把数据段 source 字符串中的 200 个字节传送到附加段 destination 字符串，用 MOVSW 指令实现。

```
    mov si,offset source          ; 源串首地址送 SI
    mov di,offset destination     ; 目的串首地址送 DI
    mov cx,100                    ; cx←传送次数
    cld                           ; 置 DF = 0,地址递增
again:
    movsw                         ; 传送一个字
    dec cx                        ; 传送次数减 1
    jnz again                     ; 判断传送次数 cx 是否为 0.不为 0,则转到 again 位置执行指
                                  ; 令; 否则,结束
```

在使用 MOVS 指令进行串传送时，要注意传送方向，如图 3-6 所示。如果源串与目的串不重叠[如图 3-6(a)所示]，则传送方向没有任何影响。如果源串与目的串部分重叠(如

图 3-6(b)和(c)所示),则要注意传送方向:若源串的首址低于目的串的首址,则应自动减量(DF=1);若源串的首址高于目的串的首址,则应自动增量(DF=0)。

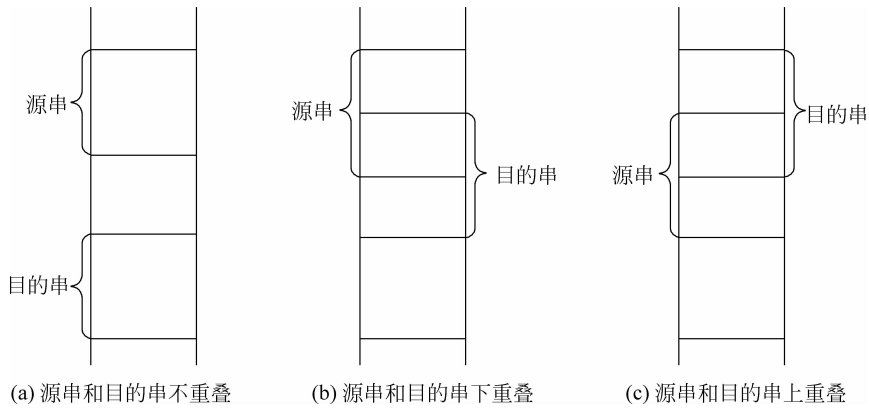


图 3-6 字符串传送方向示意图

前面提到过,MOVS 指令执行一次,仅传送数据串中的一个字节、字或双字,当该指令与前缀 REP 联用时,则可将数据段中的整串数据传送到附加段中去。这里,源串必须在数据段中,目的串必须在附加段中,但源串允许使用段跨越前缀来修改。在与 REP 联用时还必须先把数据串的长度值送到计数寄存器中,以便控制指令结束。因此在执行该指令前,应该先做好以下准备工作:

① 把存放在数据段中的源串首地址(若反向传送则应是末地址)放入源变址寄存器中。

② 把将要存放数据串的附加段中的目的串首地址(或反向传送时的末地址)放入目的变址寄存器中。

③ 把数据串长度放入计数寄存器。

④ 建立方向标志。

完成这些准备工作后,就可以使用串指令传送信息。

为了建立方向标志,需要使用两条指令:

① CLD:该指令使 DF=0,在执行串处理指令时可使地址自动增量。

② STD:该指令使 DF=1,在执行串处理指令时可使地址自动减量。

例如,MOVS 指令与 REP 联用实现例 3-56 的程序段如下:

```
mov si,offset source          ; 源串首地址送 SI
mov di,offset destination     ; 目的串首地址送 DI
mov cx,100                    ; cx←传送次数
cld                           ; 置 DF = 0,地址递增
rep movsw                     ; 重复传送
```

如果使(CX)=200,就可以采用 MOVSB 实现同样的功能:

```
mov si,offset source          ; 源串首地址送 SI
mov di,offset destination     ; 目的串首地址送 DI
mov cx,200                    ; cx←传送次数
cld                           ; 置 DF = 0,地址递增
rep movsb                     ; 重复传送
```

3. 存入串指令 STOS

语句格式：

字节操作：STOSB

字操作：STOSW

双字操作：STOSD

该指令把 AL、AX 或 EAX 寄存器的内容存入由 DI 指定的附加段内存单元中，并根据 DF 和传送单位修改 DI 寄存器，即执行的操作为：

字节操作：((DI))←(AL)，(DI)←(DI)±1

字操作：((DI))←(AX)，　(DI)←(DI)±2

双字操作：((DI))←(EAX)，　(DI)←(DI)±4

STOS 指令不影响标志。当它与 REP 联用时，可把 AL、AX 或 EAX 寄存器的内容存入一个长度为(CX)的缓冲区中。该指令在初始化某一缓冲区时很有用。

【例 3-57】 把附加段中首址为 mess 的 20 个字节缓冲区置为 10H。

方法一：用 STOS 指令实现。

```
        lea  di, mess       ; 缓冲区首址送 DI
        mov  al, 10H        ; 初始化数据 10H 送入 AL
        mov  cx, 20         ; 传送次数送入 CX
        cld                 ; 设置 DF = 0,实现地址递增
again:  stosb               ; 传送一个字节
        dec  cx
        jnz  again          ; 判断传送次数(CX)是否为 0
```

方法二：用 STOS 指令与 REP 联用实现。

```
Lea  di, mess               ; 缓冲区首址送 DI
mov  ax, 1010H              ; 初始化数据 1010H 送入 AX
mov  cx, 10                 ; 传送次数送入 CX
cld                         ; 设置 DF = 0,实现地址递增
rep  stosw                  ; 重复传送 AX 中的内容至缓冲区
```

4. 从串取指令 LODS

语句格式：

字节操作：LODSB

字操作：LODSW

双字操作：LODSD

LODS 指令和 STOS 指令功能互逆，它将 SI 寄存器指向的内存单元内容送至 AL、AX 或 EAX 中，并根据 DF 和数据类型修改 SI 的内容使其指向下一个元素，即执行的操作为：

字节操作：(AL)←((SI))，　(SI)←(SI)±1

字操作：(AX)←((SI))，　(SI)←(SI)±2

双字操作：(EAX)←((SI))，　(SI)←(SI)±4

该指令允许使用段跨越前缀来指定非数据段的存储区，该指令也不影响条件标志位。该指令一般不和 REP 联用。有时缓冲区中的一串字符需要逐次取出来测试时，可使用本指令。

【例 3-58】 设 BLOCK 数据块中存储有正数和负数，其长度为 count 字节，试编写程序将正负数分开，分别存放在 dplus 和 dminus 开始的存储区域。

```
        mov     si,offset block
        mov     di,offset dplus
        mov     bx,offset dminus
        mov     ax,ds
        mov     es,ax                   ; 数据都在一个段中,所以设置 ES = DS
        mov     cx,count                ; 数据块长度送 CX
        cld
go_on:  lodsb                           ; 从 block 取出一个数据
        test    al,80h                  ; 检测符号位,判断是正是负
        jnz     minus                   ; 符号位为 1,是负数,转向 minus
        stosb                           ; 符号位为 0,是正数,存入 dplus
        jmp     again                   ; 程序转移到 again 处继续执行
minus:  xchg    bx,di
        stosb                           ; 把负数存入 dminus
        xchg    bx,di
again:  loop    go_on                   ; 字节数减 1,重复执行
```

5. 串输入/串输出指令 INS/OUTS

1）串输入指令 INS

语句格式：

字节操作：INSB

字操作：INSW

双字操作：INSD

该指令把端口号在 DX 寄存器中的 I/O 空间的字节、字或双字传送到附加段中的由目的变址寄存器 DI 所指向的存储单元中，并根据 DF 的值和数据类型修改 DI 的内容，即执行的操作为：

字节操作：((DI))←((DX))
(DI)←(DI)±1

字操作：((DI))←((DX))
(DI)←(DI)±2

双字操作：((DI))←((DX))
(DI)←(DI)±4

该指令不影响标志位，当它与 REP 联用时，可以把成组的字节、字或双字输入到长度为 CX 的缓冲区中。

2）串输出指令 OUTS

语句格式：

字节操作：OUTSB

字操作：OUTSW

双字操作：OUTSD

该指令把由源变址寄存器 SI 所指向的存储器中的字节、字或双字传送到端口号在 DX 寄存器中的 I/O 端口中去，并根据 DF 的值及数据类型修改源变址寄存器 SI 的内容，即执

行的操作为：

字节操作：((DX))←((SI))
　　(SI)←(SI)±1

字操作：((DX))←((SI))
　　(SI)←(SI)±2

双字操作：((DX))←((SI))
　　(SI)←(SI)±4

该指令不影响标志位，当它与 REP 联用时，可以把存储器中长度为 CX 的字节、字或双字成组地传送到 I/O 空间。

6. 串比较指令 CMPS

语句格式：

字节操作：CMPSB

字操作：CMPSW

双字操作：CMPSD

该指令将源串中的一个字节、字或双字与目的串中的一个字节、字或双字相减，但不保存结果，只根据其减法结果设置标志位，指令在每次比较之后修改 SI 和 DI 的值，使之指向下一元素，即执行的操作为：

① ((SI))－((DI))根据比较结果置条件标志位：相等则 ZF＝1，不等则 ZF＝0。

② 字节操作：(SI)←(SI)±1,(DI)←(DI)±1

字操作：(SI)←(SI)±2,(DI)←(DI)±2

双字操作：(SI)←(SI)±4,(DI)←(DI)±4

通常在 CMPS 指令前加重复前缀 REPE/REPZ，用来寻找两个串中的第一个不相同的数据；在 CMPS 指令前加重复前缀 REPNE/REPNZ，用来寻找两个串中的第一个相同的数据。

【例 3-59】 编写程序判断数据段中字符串 str1 和附加段中字符串 str2 是否相同，设 str1 和 str2 的长度相同，都为 count 个字节。比较的结果存入 result 单元，为 0 表示相等；为－1(即 FFH)表示不等。

```
            mov     si, offset str1
            mov     di, offset str2
            mov     cx, count               ; 字符串长度 count 送 CX
            cld                             ; 清方向标志 DF
            repe    cmpsb                   ; 两串相等并且未比较完时继续比较
            jne     notequal                ; ZF = 0,两串不相同,转移到 notequal
            mov     al, 0                   ; 两串相同,设置 00h 标记
            jmp     output                  ; 转向 output
notequal:   mov     al, offh
output:     mov     result, al              ; 输出结果标记
```

7. 串扫描指令 SCAS

语句格式：

字节操作：SCASB

字操作：SCASW

双字操作：SCASD

SCAS指令是用来从目标串中查找某个关键字，要求查找的关键字应事先置入AL、AX或EAX寄存器中。将AL/AX/EAX寄存器中的关键字减去由DI所指向的目标串中一个元素，不传送结果，只根据结果置标志位，修改DI寄存器内容指向下一元素，即执行的操作为：

字节操作：(AL)－((DI))，(DI)←(DI)±1

字操作：(AX)－((DI))，(DI)←(DI)±2

双字操作：(EAX)－((DI))，(DI)←(DI)±4

SCAS指令和REPE/REPZ或REPNE/REPNZ相结合，可以用来从一个字符串中查找一个指定的字符。

【例3-60】 在字符串string中查找是否存在“$”字符。若存在，则将“$”字符所在地址送入BX寄存器中，否则将BX寄存器清0。程序如下：

```
        cld                         ; 清方向标志DF
        lea    di,string            ; 送目的串首地址
        mov    al,'$'               ; 关键字"$"送AL
        repne  scasb                ; 找关键字
        and    cx,0ffh
        jz     zer                  ; 没找到,转到zer
        dec    di
        mov    bx,di                ; 找到,将关键字所在地址送BX
        jmp    st0
zer:    mov    bx,0                 ; 将0送BX
st0:    …
```

3.3.5 控制转移指令

程序的执行序列是由代码段寄存器CS和指令指针寄存器IP或EIP确定的，CS包含当前指令所在代码段的段地址，IP或EIP则存放下一条要执行指令的偏移地址。一般情况下指令是顺序逐条执行的，但实际上程序不可能全部顺序执行而经常需要改变程序的执行流程。控制转移指令就是通过修改CS和IP或EIP的值来控制程序的执行流程的。控制转移指令包括5类指令，即无条件转移指令、条件转移指令、循环指令、子程序调用及返回指令、中断及中断返回指令。

1. 与转移地址有关的寻址方式

在讲述控制转移指令之前，先把寻址方式问题中遗留的一部分即与转移地址有关的寻址方式做一个介绍。

转移可以分成两类：段内转移和段间转移。段内转移是指在同一段的范围内进行转移，此时只需改变IP和EIP寄存器的内容，即用新的转移目标地址代替原有的IP或EIP的值就可达到转移的目的。段间转移则是要转到另一个段去执行程序，此时不仅要修改IP或EIP寄存器的内容，还需要修改CS寄存器的内容才能达到目的。无论段内转移还是段间转移，都有直接和间接转移之分。所谓直接转移，就是转移的目标地址信息直接出现在指令的机器码中；所谓间接转移，就是转移的目标地址信息间接存储于某一个寄存器中或某一个内存变量中。所以，与转移地址有关的寻址方式有4种：段内直接寻址、段内间接寻址、段间直接寻址、段间间接寻址，如图3-7所示。

为了叙述方便，下面以转移指令为对象来分析各种转移地址的寻址方式。这些寻址方式也适用于子程序调用指令中对调用地址的寻址。

1）段内直接寻址

转向的有效地址是当前 IP 寄存器的内容和指令中指定的 8 位或 16 位位移量之和，如图 3-7(a)所示。

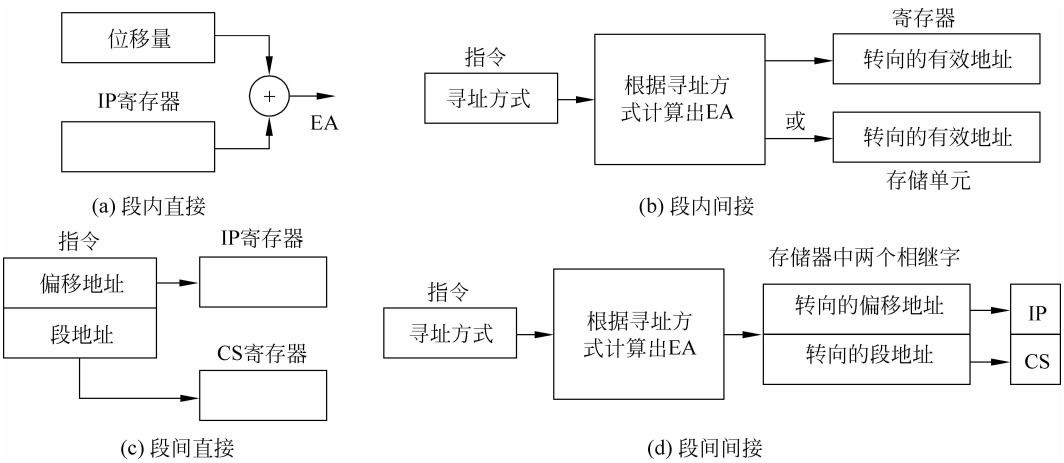


图 3-7　与转移地址有关的寻址方式

这种方式的有效地址用相对于当前 IP 值的位移量来表示，所以它是一种相对寻址方式。指令中的位移量是转向的有效地址与当前 IP 值之差，所以当这一程序段在内存中的不同区域运行时，这种寻址方式的转移指令本身不会发生变化，这是符合程序的再定位要求的。这种寻址方式适用于条件转移及无条件转移指令，但是当它用于条件转移指令时，位移量只允许 8 位(386 及其后继机型条件转移指令的位移量可为 8 位或 32 位)。无条件转移指令在位移量为 8 位时称为短跳转，位移量为 16 位时则称为近跳转。

例如：

```
JMP   NEAR PTR LOOP1
JMP   SHORT LOOP2
```

其中，LOOP1 和 LOOP2 均为转向的符号地址，在机器指令中，用位移量表示。在汇编指令中，如果位移量为 16 位，则在符号地址前加操作符 NEAR PTR；如果位移量为 8 位，则在符号地址前加操作符 SHORT。

对于 386 及其后继机型，代码段的偏移地址存放在 EIP 中，同样用相对寻址的段内直接方式，只是其位移量为 8 位或 32 位。8 位对应于短跳转；32 位对应于近跳转。因为位移量本身是一个有符号数，所以 8 位位移量的跳转范围在－128～＋127 的范围内；16 位位移量的跳转范围为±32K，32 位位移量的跳转范围为±2G。

2）段内间接寻址

转向有效地址是一个寄存器或一个存储单元的内容。这个寄存器或存储单元的内容可以用数据寻址方式中除立即数以外的任何一种寻址方式取得，所得到的转向有效地址用来取代 IP 的内容，如图 3-7(b)所示。

这种寻址方式以及以下的两种寻址方式都不能用于条件转移指令。也就是说，条件转移指令只能使用段内直接寻址的 8 位或 32 位位移量，而 JMP 和 CALL 指令则可用四种寻址方式中的任何一种。

例如：

```
JMP   BX
JMP   WORD PTR[SI + 2000H]
```

其中，WORD PTR 是一个操作符，说明其后的寻址方式所取得的转向地址是一个字类型的有效地址，从而表明这种寻址方式是一种段内转移。

以上两种寻址方式均为段内转移，所以直接把求得的转向有效地址送到 IP 寄存器就可以了。

3）段间直接寻址

在指令中直接给出转向的段地址和偏移地址，所以只要用指令中指定的偏移地址取代 IP 的内容，用指令中指定的段地址取代 CS 的内容就完成了从一个段到另一个段的转移操作，如图 3-7(c)所示。

例如：

```
JMP   FAR PTR LOOP3
```

其中，LOOP3 为转向的符号地址，FAR PTR 则是表示段间转移的操作符。

对于 386 及其后继机型，段间转移应修改 CS 和 EIP 的内容，方法与 16 位寻址时一致。

4）段间间接寻址

由指令寻址方式确定的连续两个字的内容来取代 IP 和 CS 寄存器中的原有内容，低位字单元中的 16 位数据作为转向的偏移地址用以取代 IP 的内容，高位字单元中的 16 位数据作为段基址用以取代 CS，从而实现段间程序转移。这里，存储单元的地址是由指令指定除立即数方式和寄存器以外的任何一种数据寻址方式取得，如图 3-7(d)所示。

例如：

```
JMP   DWORD PTR[BX]
```

其中，DWORD PTR 为双字操作符，是一个双字的有效地址，说明后面紧跟的存储器操作数所取得的转向地址。32 位寻址方式与 16 位寻址相同。

2. 无条件转移指令

无条件转移就是无任何先决条件就能使程序改变执行顺序，它可以转移到内存中存放的任何程序段。处理器只要执行无条件转移指令 JMP，就使程序转到指定的目标地址处，从目标地址处开始执行那里的指令。下面给出 JMP 指令的各种格式及其执行的操作。

1）段内直接短转移

语句格式：JMP SHORT OPR

执行的操作：(IP)←(IP)＋8 位位移量　或(EIP)←(EIP)＋8 位位移量

若操作数长度为 16 位，则还需(EIP)←(EIP)AND 0000FFFFH

其中，8 位位移量由目标地址 OPR 确定。转移的目标地址在汇编格式中可直接用符号

地址，而在机器执行时则是当前的 IP 或 EIP 的值与指令中指定的 8 位位移量之和。位移量是一个有符号数，以满足向前或向后转移的需要，这种转移格式只允许在－128～＋127 字节的范围内转移。

2）段内直接近转移

语句格式：JMP NEAR PTR OPR

执行的操作：(IP)←(IP)＋16 位位移量　或(EIP)←(EIP)＋32 位位移量

若操作数长度为 16 位，则(EIP)←(EIP)AND 0000FFFFH

也采用相对寻址方式，位移量为 16 位或 32 位，在汇编格式中 OPR 也是只需要使用符号地址。该转移格式可以转移到段内的任何位置。

3）段内间接近转移

语句格式：JMP WORD PTR OPR

执行的操作：(IP)←(EA)或(EIP)←(EA)

若操作数长度为 16 位，则(EIP)←(EIP)AND 0000FFFFH

其中，有效地址 EA 的值由 OPR 的寻址方式确定。它可以使用除立即数方式以外的任一种寻址方式。如果指定的是寄存器，则把寄存器的内容送到 IP 或 EIP 中；如果指定的是存储器中的一个字或双字，则把该存储单元的内容送到 IP 或 EIP 中。

4）段间直接远转移

语句格式：JMP FAR PTR OPR

执行的操作：(IP)←OPR 的段内偏移地址

(CS)←OPR 所在段的段地址

或者

(EIP)←OPR 的段内偏移地址

(CS)←OPR 所在段的段地址

若操作数长度为 16 位，则(EIP)←(EIP)AND 0000FFFFH

这里使用的是直接寻址方式。在汇编格式中 OPR 可使用符号地址，而机器语言中则要指定转向的偏移地址和段地址。

5）段间间接远转移

语句格式：JMP DWORD PTR OPR

执行的操作：(IP)←(EA)，(CS)←(EA＋2)

或者

(EIP)←(EA)，(CS)←(EA＋4)

若操作数长度为 16 位，则(EIP)←(EIP)AND 0000FFFFH

其中，有效地址 EA 的值由 OPR 的寻址方式确定，它可以使用除立即数和寄存器方式以外的任何存储器寻址方式，根据寻址方式求出 EA 后，把指定存储单元的字内容送到 IP 或 EIP 寄存器，并把下一个字的内容送到 CS 寄存器，这样就实现了段间转移。

JMP 指令不影响标志位。

3. 条件转移指令

条件转移指令根据指定的条件确定程序是否发生转移。如果满足条件，则程序转移到目标地址去执行程序；如果不满足条件，则程序将顺序执行下一条指令。

条件转移指令使用了相对寻址方式，它只可用 JMP 中的短转移和近转移两种格式。在 8086 和 80286 中只提供短转移格式，目标地址应在本条转移指令下一条指令地址的－128～＋127 个字节的范围之内。在 386 及其后继机型中，除短转移格式外，还提供了近转移格式，这样它就可以转移到段内的任何位置。但条件转移不提供段间远转移格式，若有需要，则可采用转换为 JMP 指令的办法来解决。所有的条件转移指令都不影响标志位。

条件转移指令的通用格式为：Jcc　OPR

其中，OPR 表示目标地址，Jcc 中的 cc 表示利用标志判断的条件。根据判定的标志位的不同可将条件转移指令分为 4 组。

1）根据单个条件标志的设置情况转移

该组指令的格式、功能及测试条件如表 3-3 所示。

表 3-3　根据单个条件标志位转移的条件转移指令

格　　式	指 令 功 能	测 试 条 件
JZ/JE　　OPR	结果为 0(或相等)则转移	ZF ＝ 1
JNZ/JNE　　OPR	结果不为 0(或不相等)则转移	ZF ＝ 0
JS　　OPR	结果为负则转移	SF ＝ 1
JNS　　OPR	结果为正则转移	SF ＝ 0
JO　　OPR	溢出则转移	OF ＝ 1
JNO　　OPR	不溢出则转移	OF ＝ 0
JP/JPE　　OPR	奇偶位为 1 则转移	PF ＝ 1
JNP/JPO　　OPR	奇偶位为 0 则转移	PF ＝ 0
JB/JC/JNAE　　OPR	低于/进位为 1/不高于或等于则转移	CF ＝ 1
JNB/JNC/JAE　　OPR	不低于/进位为 0/高于或等于则转移	CF ＝ 0

【例 3-61】 若 AL 最高位为 0，则设置(AH)＝0；若 AL 最高位为 1，则设置(AH)＝FFH。即编程实现符号扩展指令 CBW 的功能。

方法一：用 JZ 指令实现。

```
        test    al, 80h             ; 测试最高位
        jz      set0                ; 最高位为 0 则转到 set0
        mov     ah, 0ffh            ; 最高位为 1,则将 AH 置 0FFH
        jmp     next                ; 无条件转向 next
set0:   MOV     ah, 0
next:   …
```

方法二：用 JNZ 指令实现。

```
        test    al, 80h             ; 测试最高位
        jnz     set1                ; 最高位为 1 则转到 set1
        mov     ah, 0h              ; 最高位为 0,则将 AH 置 0
        jmp     next                ; 无条件转向 next
set1:   mov     ah, 0ff h
next:   …
```

【例 3-62】 计算|$X-Y$|，X 和 Y 是存放于 X 单元和 Y 单元的 16 位操作数，结果存入 result。

```
          mov    ax, X
          sub    ax, Y                ; X－Y 送 AX,下面求绝对值
          jns    next                 ; 绝对值为正,不需处理,转向 next 保存结果
          neg    ax                   ; 绝对值为负,进行求补得到绝对值
next:     mov    result,ax            ; 保存结果
```

【例 3-63】 编写程序段判断 DX 中 1 的个数。

```
          xor    al, al
again:    test   dx, offffh           ; 等价于 cmp bx,0
          je     next
          shl    dx, 1
          jnc    again
          inc    al
          jmp    again
next:     …                           ; AL 保存 1 的个数
```

2）比较两个无符号数，并根据比较结果转移

用来比较两个无符号数的大小，适用于地址或双精度数低位字的比较。无符号数的大小用高(above)、低(below)表示。它利用 CF 确定高低、利用 ZF 确定相等。该组指令的格式、功能及测试条件如表 3-4 所示。

表 3-4　比较两个无符号数的条件转移指令

格　　式	指 令 功 能	测 试 条 件
JB/JNAE/JC　　OPR	低于/不高于或等于/进位为 1 则转移	CF＝1
JNB/JAE/JNC　　OPR	不低于/高于或等于/进位为 0 则转移	CF＝0
JBE/JNA　　OPR	低于等于/不高于则转移	CF＝1 或 ZF＝1
JNBE/JA　　OPR	不低于等于/高于则转移	CF＝0 且 ZF＝0

【例 3-64】 比较 AX 和 BX 中两个无符号数的大小，将较小的存放到 AX 中。

```
          cmp    ax, bx               ; 比较 ax 和 bx
          jb     next                 ; 若 ax＜bx,则转移到 next
          xchg   ax, bx               ; 若 ax≥bx,则两者交换
next:     …
```

3）比较两个有符号数，并根据比较结果转移

用于比较两个有符号数的大小，有符号数的大小用大(Greater)、小(Less)表示，比较时需要组合 OF、SF 标志，并利用 ZF 标志确定相等与否。该组指令的格式、功能及测试条件如表 3-5 所示。

表 3-5　比较两个有符号数的条件转移指令

格　　式	指 令 功 能	测 试 条 件
JL/JNGE　　OPR	小于/不大于等于则转移	SF⊕OF＝1
JNL/JGE　　OPR	不小于/大于等于则转移	SF⊕OF＝0

续表

格　式	指令功能	测试条件
JLE/JNG　OPR	小于等于/不大于则转移	SF⊕OF=1 或 ZF=1
JNLE/JG　OPR	不小于等于/大于则转移	SF⊕OF=0 且 ZF=0

【例 3-65】 比较 AX 和 BX 中两个有符号数的大小，将较小的存放到 AX 中。

```
        cmp     ax, bx              ; 比较 ax 和 bx
        jl      next                ; 若 ax<bx,则转移到 next
        xchg    ax, bx              ; 若 ax≥bx,则两者交换
next:   …
```

【例 3-66】 a、b 是双精度数，分别存于 DX，AX 及 BX，CX 中，$a>b$ 时转 L1，否则转 L2。

```
        cmp     dx, bx              ; 比较 dx 和 bx
        jg      l1                  ; 若 dx>bx,则转移到 L1
        jl      l2                  ; 若 dx<bx,则转移到 L2
        cmp     ax, cx              ; 若 dx = bx,则比较 ax 和 cx
        ja      l1                  ; 若 ax>cx,则转移到 L1
l2:     …
l1:     …
```

4）测试 CX 或 ECX 的值为 0 则转移

这两条指令与前面所有条件转移指令不同，不是根据标志位的测试而是根据对 CX 或 ECX 的内容是否为零的测试来确定是否转移，CX 或 ECX 经常用来设置计数值，所以这两条指令是根据计数值的变化情况来产生两个不同的程序分支，并且这组指令只能提供 8 位位移量，即只能做短转移，而且也不影响标志位。该组指令的格式、功能及测试条件如表 3-6 所示。

表 3-6　测试 CX 或 ECX 的条件转移指令

格　式	指令功能	测试条件
JCXZ　OPR	CX 的内容为零则转移	(CX)=0
JECXZ　OPR	ECX 的内容为零则转移	(ECX)=0

4. 循环指令

循环结构是程序的三大结构之一。为了方便构成循环结构，汇编语言提供了多种循环指令，这些循环指令的循环次数都是保存在计数器 CX 或 ECX 中。除了 CX 或 ECX 可以决定循环是否结束外，有的循环指令还可由标志位 ZF 来决定是否结束循环。

汇编语言的循环指令都是放在循环体的下面，在循环时，首先执行一次循环体，然后把循环计数器 CX 或 ECX 减 1。当循环终止条件达到满足时，该循环指令下面的指令将是下一条被执行的指令，否则，程序将向上转到循环体的第一条指令。

循环指令包括三条指令：LOOP、LOOPZ / LOOPE、LOOPNZ / LOOPNE。它们的格式、功能及测试条件如表 3-7 所示。

表 3-7　循环指令

格　　式	指 令 功 能	测 试 条 件
LOOP　　OPR	(E)CX←(E)CX－1,若(E)CX 不为 0 则转移	(E)CX≠0
LOOPZ/LOOPE　　OPR	(E)CX←(E)CX－1,若(E)CX 不为 0 且 ZF＝1 则转移	(E)CX≠0 且 ZF＝1
LOOPNZ/LOOPNE　　OPR	(E)CX←(E)CX－1,若(E)CX 不为 0 且 ZF＝0 则转移	(E)CX≠0 且 ZF＝0

LOOP 指令首先将计数器 ECX 或 CX 中的值减 1,然后判断计数值是否为 0。若不为 0 则继续执行循环体内的指令,若为 0 则表示循环结束,程序退出循环,顺序执行后面的指令。LOOPZ/LOOPE 和 LOOPNZ/LOOPNE 指令中又要求同时 ZF 为 1 或 0 才进行循环,用于判断结果是否为零或相等,以便提前结束循环。

在循环未终止而向上转移时,规定该转移只能是短转移,所以转向地址必须在该循环指令的下一条指令地址的－128～＋127 字节的范围之内。如果循环体过大,则可以用前面介绍的转移指令 JMP 来构造循环结构。循环指令本身的执行不影响任何标志位。

【例 3-67】 求首地址为 array 的 n 个字之和,结果存入 sum 单元。

```
        mov   cx, n                ; 将计数值 n 送 CX
        xor   ax, ax               ; 累加器 AX 清 0
        xor   dx, dx               ; DX 清 0
        xor   si, si               ; SI 清 0
again:  add   ax, array[si]        ; 将下一个数据累加到 AX
        adc   dx, 0                ; 将产生的进位累加到 DX
        add   si, 2                ; SI 加 2,指向下一个数据
        loop  again                ; 若 CX 不为 0,继续循环
        mov   sum, ax              ; 循环结束,结果的低 16 位送 sum 单元
        mov   sum + 2,dx           ; 结果的高 16 位送 sum + 2 单元
```

【例 3-68】 在长度为 m 个字符的字符串 string 中查找“空格”字符(其 ASCII 码为 20H),找到则继续执行,未找到则转到 not_found 执行。编程实现如下。

```
        mov   cx, m                ; 将字符串长度送 CX
        mov   si, -1               ; 初始化 SI 为 -1
        mov   al, 20h              ; 将"空格"符的 ASCII 码送 AL
next:   inc   si                   ; SI 加 1
        cmp   al, string[si]       ; 比较是否为空格
        loopne next                ; 如果不相等且 CX 不为零则继续循环
        jnz   not_found            ; 没找到空格符,转向 not_found
        …
not_found: …
```

5. 子程序调用及返回指令

程序中有些部分可能要实现相同的功能,只是参数不一样,而且这些功能需要经常用到,这时可以用子程序实现这个功能。子程序又称为过程,采用子程序可以提高编程效率,使程序结构更为清楚,便于维护,也有利于大程序开发时多个程序员分工合作。

子程序是完成特定功能的、相对独立的程序,调用它的程序称为主程序或调用程序。当

主程序(调用程序)需要执行这个功能时,采用 CALL 调用指令转移到该子程序的起始处执行,当运行完子程序功能后,采用 RET 返回指令回到主程序继续执行。CALL 和 RET 指令均不影响标志位。

1) 子程序调用指令 CALL

如果子程序和调用它的主程序同属一个代码段,则该子程序具有 NEAR(近)属性,否则子程序应设置 FAR(远)属性。因而子程序调用分段内调用和段间调用;调用时可以采用直接寻址,也可以采用间接寻址,故调用指令有四种格式:段内直接调用、段内间接调用、段间直接调用、段间间接调用。指令中 near ptr 表示段内调用,far ptr 表示段间调用,但是由于汇编程序可自动识别"段内"和"段间",故可省略。

CALL 指令的格式为:

```
CALL DST
```

其中,DST 为子程序的入口地址。其功能是把子程序的返回地址(即调用程序中 CALL 指令的下一条指令地址)压入堆栈保存,以便子程序返回主程序时使用,然后转移到子程序的入口地址去继续执行。

① 段内直接调用

执行的操作:

当操作数长度为 16 位时,Push(IP)

(IP)←(IP)+D16

或(EIP)←((EIP)+D16)AND 0000FFFFH

当操作数长度为 32 位时,Push(EIP)

(EIP)←(EIP)+D32

指令中 DST 给出转向地址(即子程序的入口地址,亦即子程序的第一条指令的地址),D16 或 D32 为机器指令中的位移量,它是转向地址和返回地址之间的差值。首先将指令指针入栈保存,然后把从指令中得到的距目标子程序的相对位移量加到指令指针上,得到子程序的入口地址,实现子程序调用。

例如:

```
CALL  SUB_PRO
```

② 段内间接调用

执行的操作:

当操作数长度为 16 位时,Push(IP)

(IP)←(EA)

或(EIP)←(EA)AND 0000FFFFH

当操作数长度为 32 位时,Push(EIP)

(EIP)←(EA)

指令中的 DST 可使用寄存器寻址方式或任一种存储器寻址方式,由指定的寄存器或存储单元的内容给出转向地址。EA 是由 DST 的寻址方式所确定的有效地址。

例如：

```
CALL  BX              ; 通过寄存器 BX 间接给出子程序偏移地址
CALL  WORD PTR [BX]   ; 通过存储器的字单元间接给出子程序偏移地址,字单元为寄存器间接寻址
```

③ 段间直接调用

执行的操作：

当操作数长度为 16 位时，Push(CS)
Push(IP)
(IP)←DST 指定的偏移地址
(CS)←DST 指定的段地址

当操作数长度为 32 位时，Push(CS)
Push(EIP)
(EIP)←DST 指定的偏移地址
(CS)←DST 指定的段地址

首先保存返回地址，即把现行的 CS 的内容和 IP 或 EIP 的值入栈保存，然后把指令中由 DST 指定的偏移地址和段地址送入 IP 或 EIP 和 CS，转向目标地址去执行。因为调用程序和子程序不在同一段中，所以返回地址的保存和转向地址的设置都要考虑到段地址。

例如：

```
CALL FAR PTR SUB_PRO
```

或

```
CALL SUB_PRO
```

④ 段间间接调用

执行的操作：

当操作数长度为 16 位时，Push(CS)
Push(IP)
(IP)←(EA)
(CS)←(EA+2)

当操作数长度为 32 位时，Push(CS)
Push(EIP)
(EIP)←(EA)
(CS)←(EA+4)

首先把现行的 CS 的内容和 IP 或 EIP 的值入栈保存，然后把指令中的偏移地址和段地址送入 IP 或 EIP 和 CS。其中，EA 是由 DST 的寻址方式所确定的有效地址，可使用任一种存储器寻址方式来取得 EA 值。

例如：

```
CALL DWORD PTR[BX]  ; 通过存储器的双字单元间接给出子程序地址,双字单元为寄存器间接寻址,低
                    ; 位字为子程序偏移地址,高位字为子程序段地址
```

2）子程序返回指令 RET

返回指令是子程序最后执行的指令，它使子程序在功能完成后返回调用程序继续执行，而返回地址是主程序调用子程序时存放在堆栈中的，所以，RET 指令的作用是将返回地址出栈送 IP 或 EIP(段内或段间)和 CS 寄存器(段间)。根据子程序与主程序是否同处于一个代码段内，返回指令分为段内返回和段间返回，两类指令的助记符相同，由汇编程序加以区分，并产生不同的机器指令。

① 段内近返回 RET

执行的操作：当操作数长度为 16 位时，(IP)←Pop()

或(EIP)←(EIP)AND 0000FFFFH

当操作数长度为 32 位时，(EIP)←Pop()

② 段内带立即数近返回 RET EXP

执行的操作：当操作数长度为 16 位时，(IP)←Pop()

或(EIP)←(EIP)AND 0000FFFFH

当操作数长度为 32 位时，(EIP)←Pop()

然后，需要修改堆栈指针：(SP 或 ESP)←(SP 或 ESP)＋D16

其中，EXP 是一个表达式，根据它的值计算出来的常数成为机器指令中的位移量 D16。这种指令允许返回地址出栈后修改堆栈的指针，这就便于调用程序在用 CALL 指令调用子程序以前把子程序所需要的参数入栈，以便子程序运行时使用这些参数。当从子程序返回主程序后，这些参数已不再有用，(SP 或 ESP)＋D16 用以释放参数所占的堆栈单元，修改指针使其指向参数入栈以前的值。

③ 段间远返回 RET

执行的操作：当操作数长度为 16 位时，(IP)←Pop()

或(EIP)←(EIP)AND 0000FFFFH

(CS)←Pop()

当操作数长度为 32 位时，(EIP)←Pop()

(CS)←Pop()(32 位数出栈，高 16 位废除)

④ 段间带立即数远返回 RET EXP

执行的操作：当操作数长度为 16 位时，(IP)←Pop()

或(EIP)←(EIP)AND 0000FFFFH

(CS)←Pop()

当操作数长度为 32 位时，(EIP)←Pop()

(CS)←Pop()(32 位数出栈，高 16 位废除)

然后需要修改堆栈指针：(SP 或 ESP)←(SP 或 ESP)＋D16

这里 EXP 的含义及使用情况与段内带立即数近返回的指令相同。

【例 3-69】 子程序举例：利用子程序将 AL 低 4 位的十六进制数转换成 ASCII 码。

4 位二进制数表示成十六进制数为 0～9、A～F，0～9 的 ASCII 码为 30H～39H，对 0～9 直接加 30H，即得到其 ASCII 码；对 A～F，加上 30H 后还要再加上 7 才得其 ASCII 码。如 4 位二进制数为 1010(0AH)，则其 ASCII 码为 0AH＋30H＋07H。

```
; 主程序:
MOV  AL,9DH              ; 提供参数 AL
CALL H2ASC               ; 调用子程序 H2ASC
      …
; 子程序:
H2ASC    PROC            ; 子程序定义伪指令
         AND  AL,0FH     ; 取 AL 的低 4 位
         ADD  AL,30H     ; 加 30H
         CMP  AL,39H     ; 是 0～9 还是 A～F
         JBE  H2END      ; 是 0～9 转向 H2END
         ADD  AL,07H     ; 是 A～F,再加 7
H2END:   RET             ; 子程序返回
H2ASC    ENDP            ; 子程序定义结束
```

6. 中断及中断返回指令

这里介绍有关中断的三条指令：软中断指令 INT、中断返回指令 IRET/IRETD、若溢出则中断指令 INTO。

1) 软中断指令 INT

语句格式：INT　TYPE

执行的操作：

① 标志寄存器入栈，即 Push(FLAGS)。

② 禁止响应可屏蔽中断和单步中断，将对准检查标志 AC 置 0，即 IF←0，TF←0，AC←0。

③ 断点的段地址 CS 入栈，偏移地址 IP 入栈，即 Push(CS)，Push(IP)。

④ 从中断向量表取入口地址，进入中断服务程序，即

(IP)←(TYPE * 4)，(CS)←(TYPE * 4＋2)

该指令的功能是调用类型号为 TYPE 的中断处理程序。其中，TYPE 为类型号，可以是常数或常数表达式，其值必须在 0～255 之间。在程序中使用 INT 指令时，预定义的类型号(0、1、2、3、4)和可屏蔽中断已使用的类型号一般不出现在 INT 指令中。INT 指令和后面的 INTO 指令不影响除 IF、TF 和 AC 以外的标志位。

2) 中断返回指令 IRET/IRETD

语句格式：IRET/IRETD

其功能是将 IP、CS、标志寄存器依次出栈，IRET 适用于操作数长度为 16 位的情况，IRETD 适用于操作数长度为 32 位的情况，即执行的操作为：

IRET：(IP)←Pop()
　　(CS)←Pop()
　　(FLAGS)←Pop()

IRETD：(EIP)←Pop()
　　(CS)←Pop()
　　(EFLAGS)←Pop()

这两条指令的标志位由堆栈中取出的值来设置。

3) 若溢出则中断指令 INTO

语句格式：INTO

若溢出标志 OF=1,产生 4 号中断,否则顺序执行,即执行的操作为:

① 标志寄存器入栈,即 Push(FLAGS)。

② 禁止响应可屏蔽中断和单步中断,将对准检查标志 AC 置 0,即 IF←0,TF←0,AC←0。

③ 断点的段地址 CS 入栈,偏移地址 IP 入栈,即 Push(CS),Push(IP)。

④ 从中断向量表取 4 号中断的入口地址,进入中断服务程序,即

(IP)←(10H),(CS)←(12H)

3.3.6 处理机控制指令

处理机控制指令用来控制各种 CPU 的操作,指令中不需要设置地址码,所以又称为无地址指令。

1) 标志处理指令

除了有些指令影响标志位外,IA32 还提供了一组设置或清除标志位指令,它们只影响本指令指定的标志,而不影响其他标志位。这些指令是:

进位位置 0 指令 CLC	CF←0
进位位置反指令 CMC	CF←～CF
进位位置 1 指令 STC	CF←1
方向标志位置 0 指令 CLD	DF←0
方向标志位置 1 指令 STD	DF←1
中断标志置 0 指令 CLI	IF←0
中断标志置 1 指令 STI	IF←1

2) 其他处理机控制与杂项操作指令

① 空操作指令 NOP　该指令不执行任何有意义的操作,但占用 1 字节存储单元,空耗一个指令执行周期。该指令常用于程序调试。例如,在需要预留指令空间时用 NOP 填充,代码空间多余时也可以用 NOP 填充,还可以用 NOP 实现软件延时。实际上 NOP 就是 XCHG AX,AX 指令,它们的代码相同。

② 总线封锁前缀指令 LOCK　该指令是一种前缀,它可与其他指令联合,用来维持总线的锁存信号直到与其联合的指令执行完为止。当 CPU 与其他处理机协同工作时,该指令可避免破坏有用信息。

③ 停机指令 HLT(等待一次外中断,之后继续执行程序)　该指令使 CPU 处于暂停状态,不执行任何操作,不影响标志。当 RESET 线上有复位信号、CPU 响应非屏蔽中断、CPU 响应可屏蔽中断,出现以上三种情况之一,CPU 脱离暂停状态,执行 HLT 的下一条指令。

④ 换码指令 ESC

语句格式为:`ESC op,reg/mem`

这条指令在使用协处理器时,可以指定由协处理器执行的指令。指令的第一个操作数即指定其操作码,第二个操作数即指定其操作数。协处理器(如 8087、80287、80387 等)是为提高速度而可以选配的硬件。自 486 起,浮点处理部件已装入 CPU 芯片,系统可直接支持协处理器指令,所以 ESC 指令已成为未定义指令,如遇到程序中的 ESC 指令,将引起一次异常处理。

⑤ 等待指令 WAIT　该指令使处理机处于空转状态，也可以用来等待外部中断发生，但中断结束后仍返回 WAIT 指令继续等待。它也可以与 ESC 指令配合等待协处理机的执行结果。

习题

1. 分别指出下列指令中源操作数和目的操作数的寻址方式。

(1) MOV SI,200

(2) MOV CX,BUF[SI]

(3) MOV[BX],AL

(4) ADD AX,30[BP][SI]

(5) SUB BL,[2010H]

(6) MOV EAX,[EBX][ESI]

(7) INC WORD PTR [BX]

2. 根据以下要求写出相应的汇编语言指令。

(1) 把寄存器 BX 和 DX 的内容相加，结果存入 DX 寄存器中。

(2) 用寄存器 BX 和 SI 的基址变址寻址方式把存储器中的一个字节与 AL 寄存器的内容相加，并把结果送到 AL 寄存器中。

(3) 用位移量为 0524H 的直接寻址方式把存储器中的一个字与立即数 20A8H 相加，并把结果送回该存储单元。

(4) 把数 0CAH 与(AL)相加，并把结果送回 AL 中。

(5) 把首地址为 ARRAY 的字数组的第 5 个字送到 DX 寄存器。要求分别使用寄存器间接寻址、寄存器相对寻址及基址变址寻址方式实现。

3. 现有(DS)＝2000H,(BX)＝0100H,(SI)＝0002H,(20100H)＝12H,(20101H)＝34H,(20102H)＝56H,(20103H)＝78H,(21200H)＝2AH,(21201H)＝4CH,(21202H)＝B7H,(21203H)＝65H,试说明下列各条指令执行后 AX 寄存器的内容。

(1) MOV AX,1200H

(2) MOV AX,BX

(3) MOV AX,[1200H]

(4) MOV AX,[BX]

(5) MOV AX,1100[BX]

(6) MOV AX,[BX][SI]

(7) MOV AX,1100[BX][SI]

4. 判断下列指令是否正确，若有错误请改正。

(1) MOV CX,DL

(2) MOV IP,AX

(3) MOV ES,1234H

(4) MOV ES,DS

(5) MOV AL,500

(6) MOV [SP],AX

(7) MOV 40,BX

(8) MOV CL,DL

(9) MOV [BX],[SI]

(10) MOV DX,BX+DI

(11) POP CS

(12) PUSH CS

5. 在 ARRAY 数组中依次存储了 7 个字数据,紧接着是名为 ZERO 的字单元,表示如下:

```
ARRAY  DW      23, 36, 2, 100, 32000, 54, 0
ZERO   DW      ?
```

(1) 如果 BX 包含数组 ARRAY 的初始地址,请编写指令将数据 0 传送给 ZERO 单元。

(2) 如果 BX 包含数据 0 在数组中的位移量,请编写指令将数据 0 传送给 ZERO 单元。

6. 如果 BUF 为数据段中 0120 单元的符号名,其中存放的内容为 5678H,请问以下两条指令有什么区别? 指令执行完后 AX 寄存器的内容是什么?

```
MOV    AX,BUF
LEA    AX,BUF
```

7. 执行下列指令后,AX 寄存器中的内容是什么?

```
TABLE  DW  10, 20H, 30, 40, 50H
ENTRY  DW  5
       ⋮
       MOV  BX, OFFSET TABLE
       ADD  BX, ENTRY
       MOV  AX, [BX]
```

8. 假设下列程序执行前(SS)=8000H,(SP)=2000H,(AX)=7A6CH,(DX)=3158H。执行下列程序段,画出每条指令执行后,寄存器的内容、堆栈区和 SP 的内容的变化过程示意图。

```
PUSH  AX
PUSH  DX
POP   AX
POP   DX
```

9. 求出以下各十六进制数与十六进制数 62A0H 之和,并根据结果设置标志位 SF、ZF、CF 和 OF 的值。

(1) 1234H (2) 4321H (3) CFA0H (4) 9D60H

10. 求出以下各十六进制数与十六进制数 4AE0H 之和,并根据结果设置标志位 SF、ZF、CF 和 OF 的值。

(1) 1234H (2) 5D90H (3) 9090H (4) EA04H

11. 给出下列各条指令执行后的结果,以及状态标志 CF、OF、SF、ZF 及 PF 的状态。

```
MOV     AX,158AH
AND     AX,AX
OR      AX,AX
XOR     AX,AX
NOT     AX
TEST    AX,0F0F0H
```

12. 分别用一条指令完成如下要求。

(1) AH 高 4 位置 1,低 4 位不变。

(2) BH 高 4 位取反,低 4 位不变,BL 高 4 位不变,低 4 位取反。

(3) 把(DX, AX)中的双字右移 4 位。

13. 编写程序段完成如下要求。

(1) 用移位指令实现 AL(无符号数)乘以 10。

(2) 用逻辑运算指令实现数字 0～9 的 ASCII 码与非压缩 BCD 码的互相转换。

(3) CX 低 4 位清零,其他位不变。

14. 写出执行以下计算的指令序列,其中 X、Y、Z、R 和 W 均为存放 16 位有符号数单元的地址。

(1) Z←W+(Z−X)

(2) Z←W−(X+6)−(R+10)

(3) Z←(W * X)/(Y+6), R←余数

(4) Z←((W−X)/5 * Y) * 2

15. 变量 DAT1 和 DAT2 的定义如下:

```
DAT1  DW  0148H
      DW  2316H
DAT2  DW  0237H
      DW  4052H
```

请按以下要求写出指令序列。

(1) DAT1 和 DAT2 两个字数据相加,和存放在 DAT2 中。

(2) DAT1 和 DAT2 两个双字数据相加,和存放在从 DAT2 开始的字单元中。

(3) DAT1 和 DAT2 两个字数据相乘(用 MUL)。

(4) DAT1 和 DAT2 两个双字数据相乘(用 MUL)。

(5) DAT1 除以 25(用 DIV)。

(6) DAT1 双字除以字 25(用 DIV)。

16. 编写一个程序段求出双字长数的绝对值。双字长数在 A 和 A+2 单元中,结果存放在 B 和 B+2 单元中。

17. 假定(BX)=1A8H,(CL)=3,CF=1,确定下列各条指令单独执行后 BX 中的值。

```
SHR  BX, 1
SAR  BX, CL
SHL  BX, CL
SHL  BL, 1
ROR  BX, CL
ROL  BL, CL
```

```
SAL  BH, 1
RCL  BX, CL
RCR  BL, 1
```

18. 假设数据定义如下：

```
CONAME  DB  'SPACE EXPLORERS INC.'
PRLINE  DB  20 DUP(?)
```

用串处理指令编写程序段分别完成以下功能：

(1) 从左到右把 CONAME 中的字符串传送到 PRLINE。

(2) 从右到左把 CONAME 中的字符串传送到 PRLINE。

(3) 把 CONAME 中的第 3 个和第 4 个字节装入 AX。

(4) 把 AX 寄存器中的内容存入从 PRLINE+5 开始的字节中。

(5) 检查 CONAME 字符串中有无空格字符，如有，则把它传送给 BH 寄存器。

19. 编写程序段，把字符串 STRING 中的'&'字符用空格符代替。

```
STRING  DB   'The date is FEB&03'
```

20. 判断下列程序段跳转的条件。

(1)
```
XOR  AX,1A2BH
JE   EQUAL
```

(2)
```
TEST AL,10100011B
JNZ  THERE
```

(3)
```
CMP  CX,64H
JB   THERE
```

21. 假设寄存器 AX 和 SI 中存放的是有符号数，DX 和 DI 中存放的是无符号数，请用比较指令和条件转移指令实现以下判断。

(1) 若(DX)＞(DI)，转到 ABOVE 执行；

(2) 若(AX)＞(SI)，转到 GREATER 执行；

(3) 若(CX)＝0，转到 ZERO 执行；

(4) 若(AX)－(SI)产生溢出，转到 OVERFLOW 执行；

(5) 若(SI)≤(AX)，转到 LESS_EQ 执行；

(6) 若(DI)≤(DX)，转到 BELOW_EQ 执行。

22. 在下列程序段的括号中分别填入如下指令：

(1) LOOP NEXT

(2) LOOPE NEXT

(3) LOOPNE NEXT

试说明在这三种情况下，当程序执行完后，AX、BX、CX 和 DX 四个寄存器的内容分别是什么？

```
    ⋮
MOV  AX, 1
MOV  BX, 2
MOV  CX, 3
```

```
      MOV  DX, 4
NEXT: INC  AX
      ADD  BX, AX
      SHR  DX, 1
      (    )
           ⋮
```

23. 考虑以下的调用序列：

(1) MAIN 调用 NEAR 的 SUBA 过程(返回的偏移地址为 0400H)；

(2) SUBA 调用 NEAR 的 SUBB 过程(返回的偏移地址为 0A00H)；

(3) SUBB 调用 FAR 的 SUBC 过程(返回的段地址为 B200H,偏移地址为 0100H)；

(4) 从 SUBC 返回 SUBB；

(5) SUBB 调用 NEAR 的 SUBD 过程(返回的偏移地址为 0C00H)；

(6) 从 SUBD 返回 SUBB；

(7) 从 SUBB 返回 SUBA；

(8) 从 SUBA 返回 MAIN；

(9) 从 MAIN 调用 SUBC(返回的段地址为 1000H,偏移地址为 0600H)。

请画出每次调用及返回时的堆栈状态。

24. 假设(EAX)＝9823F456H,(ECX)＝1F23491H,(BX)＝348CH,(SI)＝2000H,(DI)＝4044H。在 DS 段中从偏移地址 4044H 单元开始的 4 个字节单元中,依次存放的内容为 92H、6DH、0A2H 和 4CH,请问下列各条指令执行完后目的地址及其中的内容是什么?

(1) MOV　[SI], EAX

(2) MOV　[BX], ECX

(3) MOV　EBX, [DI]

25. 请给出下列各指令序列执行完后目的寄存器的内容。

(1) MOV　EAX, 5286FF95H
　　ADD　EAX, 0024FFFEH

(2) MOV　EBX, 40000000H
　　SUB　EBX, 10215000H

(3) MOV　EAX, 39393834H
　　AND　EAX, 0F0F0F0FH

(4) MOV　EDX, 9FE35D1H
　　XOR　EDX, 0F0F0F0FH