第 4 章　汇编语言程序设计

4.1　汇编语言概述

4.1.1　汇编语言的特点

汇编语言是在早期使用计算机的过程中自然产生的。使用助记符代替操作码，用变量名、标号、子程序名等代替地址码，这样对机器语言符号化处理的结果就是汇编语言。汇编语言相比机器语言更便于阅读和理解、书写和编程、调试和维护。汇编语言可以看成是对计算机的简单抽象，但其本质上还是机器语言。

从计算机语言的角度来看，汇编语言是面向机器的低级语言，通常是为特定的计算机或系列计算机专门设计的。显而易见，既然汇编语言是面向机器的，必然就与人的语言习惯和思维习惯存在较大差异，不适于描述和反映现实问题，编程相对复杂和困难；既然汇编语言是低级的，那么就要求使用者了解和掌握更多的计算机概念、理论、方法和技能，这不利于计算机的普及和推广；既然汇编语言依赖于特定的计算机结构，汇编语言程序就必然缺乏通用性和可移植性。这些都是汇编语言的缺点，但换个角度说，汇编语言是计算机硬件系统逻辑功能的直接描述，所以，它是系统级程序员必不可少的工具，也是理解整个计算机系统的最佳起点和最有效途径。

从计算机程序的角度分析，低级语言保持了机器语言的优点：直接和简捷，表现在以下两方面。

① 汇编语言可有效地访问、控制计算机的各种硬件设备，如磁盘、存储器、CPU、I/O 端口等。高级语言为了保持语言与硬件的独立性，放弃了对硬件的直接控制。汇编语言对硬件的直接控制更能充分发挥硬件的特性。

② 汇编语言程序目标代码简短，占用内存少，执行速度快。汇编语言和机器语言是一一对应关系，使用汇编语言编写的程序省掉了高级语言翻译为机器语言时的额外代价，所以目标代码简短。由人所完成的程序优化工作往往要比高级语言编译器更有效，时间代价和空间代价更少，所以汇编语言程序占用内存一般都很小，执行速度快。

通过上述汇编语言的特点可以看出，汇编语言的优点在于适合编写高效且需要对机器硬件精确控制的程序。现在汇编语言经常与高级语言配合使用，应用仍然十分广泛。主要应用在两方面：① 底层的系统软件。底层的系统软件往往需要直接面对计算机系统结构或者直接操作硬件，所以多采用汇编语言。②高效程序。例如，在一些高级绘图、视频处理及实时处理等领域，程序的效率是系统的关键，一些关键算法也多采用汇编语言编写。

4.1.2 汇编程序

使用汇编语言编写的程序，机器不能直接识别，要由一种语言处理程序将汇编语言翻译成机器语言，这种翻译作用的程序就是汇编程序。

在 DOS 系统下，使用汇编语言进行软件开发的步骤如下：

① 使用文本编辑器程序编辑汇编源程序；

② 用汇编程序处理汇编源程序，将源程序转化为目标程序；

③ 用连接程序把目标程序转化为 DOS 可执行程序；

④ 调试和运行。

如图 4.1 所示，是使用汇编语言编写程序并运行的全过程。

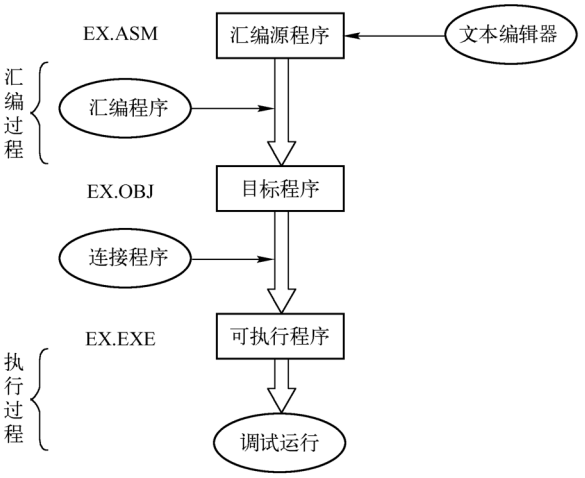


图 4.1　使用汇编语言编写程序并运行的全过程

在上述工作过程中，涉及几个名词和概念。

首先是关于程序的概念如下：

① 源程序：是用计算机语言书写的程序文本。使用汇编语言书写的就是汇编源程序。

② 目标程序：是与源程序等价的由机器代码构成的中间文件，等待与其他目标文件或库文件相连接。

③ 可执行程序：可由操作系统直接加载运行的程序。

其次是两个相关的系统程序概念如下：

① 汇编程序：又称汇编器，它将汇编语言书写的程序翻译成与之等价的机器语言程序。

② 连接程序：又称连接器，它将分别在不同的目标文件中编译或汇编形成的机器代码收集到一个可直接执行的文件中。

另外，还有两个相关的过程：

① 汇编过程：将汇编语言源程序转化为目标程序的过程。

② 执行过程：可执行程序在计算机上的运行。

与汇编语言息息相关的是汇编程序。在 80x86 体系中，最常用的是 Microsoft 公司推出

MASM（Macro AsSeMbler）汇编程序。MASM 支撑的汇编语言又称为宏汇编语言。支持 8086 的 MASM 的主要版本如表 4.1 所示。

表 4.1　MASM 汇编器主要版本

版　本	说　明
MASM 4.00	最先广泛使用的一个 MASM 版本，适用于 DOS 下的汇编编程
MASM 5.00	MASM 5.00 比 4.00 在速度上快了很多，并将段定义的伪指令简化为类似“.code”与“.data”之类的定义方式，同时增加了对 80386 处理器指令的支持，对 4.00 版本的兼容性很好
MASM 5.10	增加了对“@@”标号的支持
MASM 5.10B	1989 年推出，比上一个版本更稳定、更快，它是传统的 DOS 汇编器中最完善的版本

4.1.3　汇编语言的组成

先看下面例 4.1 中的一个汇编语言实例，该实例代码源程序名为 asm4_1.asm。

【例 4.1】　汇编语言实例。

```
DATA    SEGMENT                      ; 数据段开始          ┐
HELLO   DB   'HELLO!', 0DH, 0AH, '$' ; 数据定义            │
DATA    ENDS                         ; 数据段结束          ├ 伪指令语句
CODE    SEGMENT                      ; 代码段开始          │
        ASSUME CS:CODE, DS:DATA      ; 段指定              ┘
START:
        MOV AX, DATA                                       ┐
        MOV DS, AX                   ; DS 指向 DATA 段     │
        LEA DX, HELLO                                      │
        MOV AH, 09                                         ├ 指令语句
        INT 21H                      ; 屏幕输出字符串      │
        MOV AX, 4C00H                                      │
        INT 21H                      ; 结束程序，返回 DOS  ┘
CODE    ENDS                         ; 代码段结束          ┐
        END START                    ; 程序结束            ┘ 伪指令语句
```

如实例中显示的，汇编语言由指令和伪指令组成。指令的作用是产生机器指令，它们最终由 CPU 执行，实现程序的功能。而伪指令是对汇编过程起某种控制作用的特殊命令，它们并不产生机器指令。

对例 4.1 进行如下汇编过程处理。

（1）汇编

```
C:\masm>masm asm4_1.asm;↵
```

说明：MASM.EXE 是汇编程序，asm4_1.asm 是汇编源程序，汇编程序给出如下回答：

```
Microsoft (R) Macro Assembler Version 5.00
Copyright (C) Microsoft Corp 1981-1985, 1987.    All rights reserved.
50722 + 415390 Bytes symbol space free
0 Warning Errors
```

```
0 Severe  Errors
```

汇编程序没有发现错误，并产生了目标文件“asm4_1.obj”。

（2）连接

```
C:\masm>link asm4_1.obj;↵
```

说明：LINK.EXE 是连接程序，连接程序给出如下回答：

```
Microsoft (R) Overlay Linker  Version 3.60
Copyright (C) Microsoft Corp 1983-1987.  All rights reserved.
LINK : warning L4021: no stack segment
```

连接程序警告没有堆栈段，该例确实没用到堆栈段，所以没有错误，连接成功产生了可执行文件“asm4_1.exe”。

（3）执行

```
C:\masm>asm4_1↵
```

说明：屏幕输出如下结果：

```
Hello!
```

下面使用调试工具 DEBUG 对可执行文件 asm4_1.exe 进行分析。

（1）代码分析

代码分析如表 4.2 所示。

表 4.2　代码分析

地　址	机 器 指 令	指令助记符	指 令 语 句
13F4:0000	B8F313	MOV AX, 13F3	MOV AX, DATA
13F4:0003	8ED8	MOV DS, AX	MOV DS, AX
13F4:0005	8D160000	LEA DX, [0000]	LEA DX, HELLO
13F4:0009	B409	MOV AH, 09	MOV AH, 9
13F4:000B	CD21	INT 21	INT 21H
13F4:000D	B8004C	MOV AX, 4C00	MOV AX, 4C00H
13F4:0010	CD21	INT 21	INT 21H

（2）数据分析

数据分析如表 4.3 所示。

表 4.3　数据分析

地　址	十六进制形式	ASCII 形式
13F3:0000	68 65 6C 6C 6F 21 0D 0A-24 00 00 00 00 00 00 00	hello!..$ .....

从上述分析可以看出，伪指令是在汇编过程中起作用的，指示汇编程序如何准确地将指令语句翻译为机器指令，汇编过程结束后伪指令也就消失了，在目标文件和最终的可执行

文件中只剩下由指令语句汇编产生的机器指令。

作为计算机语言，伪指令是汇编语言不可缺少的语言成分，起着十分重要的作用，主要包括以下三方面：

（1）说明。伪指令的基本功能是控制汇编程序对指令的汇编方法或提供汇编所需要的上下文环境。从计算机语言的角度看，伪指令起着对程序、数据及指令的准确的无二义性的说明作用。

（2）定义。伪指令中还包括起定义性作用的伪指令，完成数据定义和空间分配的工作。

（3）抽象。为了提供更便捷和更容易理解的编程方法，有的伪指令还具有抽象的意义。例如，数据的结构化、宏等。

另外，使用 C 语言也可以实现与例 4.1 同样的功能（printf），在 DOS 下使用 Turbo C 2.0 编译产生的目标文件有 3.5 KB 左右，而例 4.1 中汇编产生的目标文件只有 124 B。

4.2 伪指令

4.2.1 程序格式

在汇编语言中，指令的格式为：

[标号：]　　指令助记符　　[操作数]　　[；注释]

注：在计算机文档说明规则中，中括号“[]”表示可选项。

其中，标号是一个标识符，表示指令的地址。标识符的组成规则如下：

① 标识符由 1～31 个字符组成；

② 字符可以是大小写字母、数字或特殊字符（“@”，“_”下划线，“?”，“$”等）；

③ 标识符的第一个字符必须是字母或下划线，不能以数字开头；

④ 标识符不能是系统保留关键字，如寄存器名称等；

⑤ MASM 标识符不区分大小写。

十六进制数如果以 A～F 开头，前面要加“0”。如果不加“0”，汇编程序就会把它当成一个标识符。

与指令格式相似，伪指令的格式为：

[名字]　　伪指令定义符　[操作数]　　[；注释]

从指令的格式和伪指令格式的说明可以看出，汇编语言的语句格式是统一的，都由如下 4 项组成：

[名字项]　　操作项　[操作数项][；注释项]

① 名字项：可选。可以是变量或标号，表示本语句的偏移地址。

② 操作项：必选。可以是指令、伪操作或宏指令。

③ 操作数项：可选。由一个或多个表达式组成。

④ 注释项：可选。用来描述语句的功能。

由此，在书写汇编语言程序应尽量遵循以下规则：

① 一条语句占一行；

② 注释由分号引导直至行尾，注释可占整行。汇编程序对注释不做处理，但为了保证程序的可读性，注释不应省略。

③ 与组成语句的 4 项相对应，程序应分成 4 栏书写，保持项的对齐。

4.2.2 段定义

1．段定义伪指令

段定义的格式为：

```
段名    SEGMENT[定位类型] [，组合类型]  [，‘分类名’]
        ……
        (段体)
        ……
段名    ENDS
```

伪指令 SEGMENT 和 ENDS 用于段的封装，即任何一个段都以 SEGMENT 语句开始到 ENDS 语句终止。

伪指令 SEGMENT 中有以下几个可选项。

（1）定位类型

定位类型表示对段的起始边界的要求，可以是以下类型：

BYTE：指定段可以从任何地址开始。

WORD：指定段必须从字的边界开始，即起始地址必须是 2 的倍数。

PARA：指定段必须从小段（16 字节）边界开始，即起始地址必须是 16 的倍数。

PAGE：指定段必须从页（256 字节）的边界开始，即起始地址必须是 256 的倍数。

定位类型默认是 PARA。

（2）组合类型

组合类型指定同名段的连接方式，常使用以下几种：

PRIVATE：不与其他模块中的同名段合并。

PUBLIC：可以与其他模块中的同名段连接而形成一个段，段长是各分段长度之和。

COMMON：可以和其他模块中的同名段重叠而形成一个段，段长是最大分段的长度。

STACK：指明是堆栈段，组合方式与 PUBLIC 相同。

AT 表达式：指定段址是表达式所指出的 16 位地址，不能用来指定代码段。

组合类型默认是 PRIVATE。

（3）分类名

分类名必须用单引号括起来，并且不超过 40 个字符。MASM 在存储段时将相同分类名的段相邻存放。

使用段定义伪指令时需要注意以下三点：

① 段名是段的符号名称，在程序引用段时，它是标记，不可省略。虽然 MASM 并没

有对段名提出要求，但段名应尽量符合常规的约定。例如 DATA 表示数据、CODE 表示代码、STACK 表示堆栈、CONST 表示常量等。

② SEGMENT 和 ENDS 应配对出现，而且段名必须相同。

③ 堆栈段应使用 STACK 组合类型指出，否则连接时将产生警告错误。

2．段指定伪指令

段指定伪指令格式为：

ASSUME 段寄存器:段名 [,段寄存器:段名,……]

伪指令 ASSUME 的作用是将所定义的段与段寄存器关联起来。与 CS 关联的段是代码段，存储要执行的指令序列，即程序；与 DS 关联的段是数据段，存储程序相关的数据；与 SS 关联的段是堆栈段，提供堆栈空间；与 ES 关联的段是附加段，常与串操作相关。这是段寄存器的一般使用方法，也是一般程序的组织方法。

ASSUME 伪指令通知汇编程序，自 ASSUME 以下的程序中，存在有 ASSUME 中定义的段与段寄存器的对应关系，作为后续语句汇编时的条件。有了这样的条件，汇编程序在处理地址时，例如，标号所体现的指令地址、变量所体现的数据地址等，就能将地址中的段地址和相对应的段联系起来。

【例 4.2】 段指定和段初始化示例。

```
DATA    SEGMENT                              ; 数据段开始
HELLO   DB   'HELLO!', 0DH, 0AH, '$'         ; 数据定义
DATA    ENDS                                 ; 数据段结束
STACK   SEGMENT STACK                        ; 堆栈段开始
        DW   100H DUP(?)                     ; 堆栈空间
STACK   ENDS                                 ; 堆栈段结束
CODE    SEGMENT                              ; 代码段开始
        ASSUME CS:CODE, DS:DATA, SS:STACK    ; 段指定
START:  MOV AX, DATA
        MOV DS, AX                           ; DS 指向 DATA 段
        LEA DX, HELLO
        MOV AH, 09
        INT 21H                              ; 屏幕输出字符串
        MOV AX, 4C00H
        INT 21H                              ; 结束程序，返回 DOS
CODE    ENDS                                 ; 代码段结束
        END START                            ; 程序结束，程序从 START 标号处开始执行
```

这才是在执行时真正让 DS 执行的 DATA

例 4.2 是段指定和段初始化的示例，操作系统在加载该例的程序并运行时，CS 初始化指向代码段 CODE（与 ASSUME CS:CODE 相关），SS 初始化指向堆栈段 STACK（与 SEGMENT 的 STACK 组合类型相关）。而 DS 和 ES 的初始化需要程序自己处理，所以 CODE 段中前两条指令就是给 DS 赋值（其值等于 DATA 段的段址），让 DS 指向 DATA 段。

3．程序结束伪指令

在例 4.1 和例 4.2 中，最后一条语句“END START”就是程序结束伪指令。

程序结束伪指令格式为：

END　[标号]

END 伪指令的作用包括如下两点：

① 标识程序结束。汇编程序在处理源程序时，碰到 END 时就终止汇编，所以在 END 之后的内容对汇编程序没有意义。

② END 的标号是可选项，指示程序开始执行的起始地址，即程序要执行的第一条指令（不一定是代码段的第一条指令）。如果多个模块相连，只有主模块的 END 语句需要使用标号可选项。

4．程序重定位

在例 4.2 的 CODE 段中，第一条指令是“MOV AX, DATA”，将 DATA 段的段址送到 AX 寄存器中，其中 DATA 是段名。在汇编语言中，段名的引用表示段址，是个立即数。实际上，包括将在后续章节介绍的对段址的其他引用方法，段址的引用总是一个立即数。这样就出现了一个问题，段址只有在段被加载到内存的时候才有真正的值，那么此前在程序中用来表示段址的立即数又是什么呢？

汇编程序在处理段址时将其处理为立即数，在指令编码中占两字节，作为预留值。在程序被加载到内存后，段被映射到实际空间上，再使用实际段址修改预留值，如图 4.2 所示，这一过程称为程序重定位。有了重定位机制后，段可以被定位到任何位置，程序也可以在任何内存位置上执行。

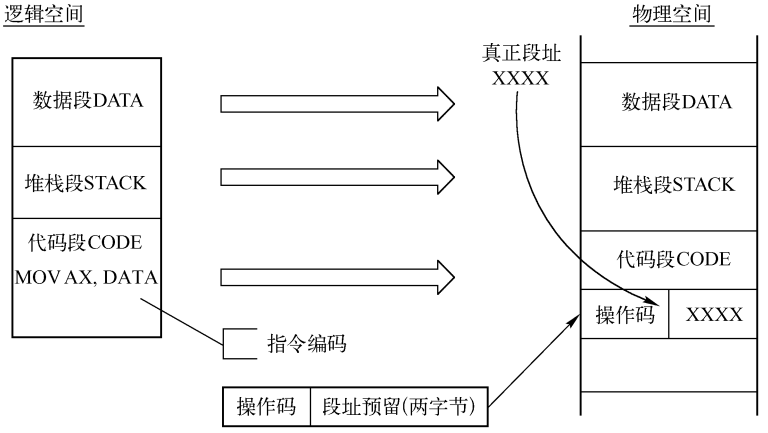


图 4.2　程序重定位

5．EXE 文件结构

DOS 系统的重定位机制需要可执行文件（EXE 文件）的支持。EXE 文件是可重定位的文件，其结构如图 4.3 所示。

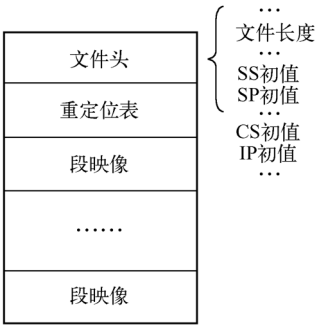


图 4.3　EXE 文件结构

在 EXE 文件中包含重定位表，是一个指针表。其中的指针表示段映像被装载到内存后，哪些地方需要用真实的段址修改。

EXE 文件开始处是 EXE 文件头，包含控制信息。主要的控制信息包括文件长度、堆栈初值、程序入口等。操作系统执行 EXE 文件的过程大致如下：

① 分配内存；

② 建立程序段前缀表 PSP（PSP 是程序与 DOS 操作系统的接口）；

③ 加载段映像；

④ 重定位；

⑤ 初始化堆栈，即对 SS 寄存器和 SP 寄存器赋初值；

⑥ 初始化 DS 和 ES，指向 PSP；将 CPU 控制权交给程序，即对 CS 寄存器和 IP 寄存器赋值。

由上述过程，可以看出以下三点：

（1）关于程序入口

程序入口就是程序要执行的第一条指令的地址，由 END 伪指令之后的标号指出。在例 4.2 中就是 START 标号，它所确定的地址，包括段址和偏移地址，就是 EXE 文件头中 CS 和 IP 的初值的来源，而且标号所在段需要由 ASSUME 伪指令将其与 CS 关联。

（2）关于堆栈初始化

EXE 文件头中 SS 和 SP 的初值和组合类型 STACK 有关。在例 4.2 中，STACK 段的定义“STACK SEGMENT STACK”使用了组合类型 STACK，它的段址用于初始化 SS，段长用于初始化堆栈指针 SP。

如果程序的堆栈段没有使用组合类型 STACK 来说明，那么堆栈的初始化需要程序自己完成，即需要在程序开始时使用指令对 SS 和 SP 赋初值。例如：

```
MOV AX, STACK
MOV SS, AX          ; SS 指向 STACK 段
MOV SP, 100H        ; SP 指向栈底
```

（3）关于 DS 和 ES 的初始化

在程序执行时，DS 和 ES 的初始值指向了程序段前缀表 PSP，即使程序已经使用 ASSUME 进行了 DS 和 ES 的指定。所以 DS 和 ES 的初始化应在程序开始时添加指令由程序自己完成。

6．简化段定义

除前面介绍的完整的段定义方法外，MASM 在 5.0 版本以后还提供了一种简单易用的简化段定义方法。

（1）存储模式

在使用简化段定义方式之前，必须使用存储模式说明伪指令描述源程序所采用的存储模式。程序存储模式说明伪指令的格式如下：

```
.MODEL  存储模式
```

在 DOS 环境下，常用的存储模式有：Tiny、Small、Compact、Medium、Large 等，如表 4.4 所示。其中，Small 存储类型是独立汇编语言源程序常用的存储模型。

表 4.4　MASM 5.0 支持的主要存储模式

存 储 模 式	功　　能
Tiny	程序只有一个段存储所有数据和代码
Small	程序只能有一个代码段和一个数据段
Medium	程序只能有一个数据段，但可以有多个代码段
Compact	程序只能有一个代码段，但可以有多个数据段
Large	程序可以有多个代码段和多个数据段

伪指令.MODEL 必须写在源程序的首部（其前内容只能是注释），且只能出现一次。

（2）简化段定义伪指令

简化段定义伪指令在说明一个新段即将开始的同时，也说明了上一个段的结束。.CODE、.DATA 和.STACK 是最常用的简化段定义伪指令，分别定义代码段、数据段和堆栈段，其说明如表 4.5 所示。

表 4.5　常用的简化段定义伪指令

简化段定义伪指令	功　　能	注　　释
.CODE [段名]	创建一个代码段	段名为可选项，如不给出段名，则采用默认段名“_TEXT”
.DATA	创建一个数据段	段名是“_DATA”
.CONST	创建一个常数据段	段名是“CONST”
.STACK [大小]	创建一个堆栈段并指定堆栈段大小	段名是“STACK”。如不指定堆栈段大小，则默认值为 1024 字节

（3）简化段段名的引用

当使用简化的段定义时，段名的引用需要使用预定义的符号，包括如下：

① “@CODE”由.CODE 伪指令定义的代码段段名。

② “@DATA”由.DATA 伪指令定义的数据段段名。

③ “@STACK”由.STACK 伪指令定义的堆栈段段名。

使用简化段定义方法可将例 4.2 书写成例 4.3 所示的形式，特别注意段名的引用。

【例 4.3】 简化例 4.2 的段定义示例。

```
.MODEL SMALL                          ; 模式定义
.DATA                                 ; 数据段开始
HELLO    DB   'HELLO!',0DH,0AH,'$'    ; 数据定义
```

```
.STACK    200H                          ; 数据段结束，
                                        ; 堆栈段开始，堆栈大小 200H 字节
.CODE                                   ; 堆栈段结束，代码段开始
START:    MOV AX, @DATA                 ; @DATA 引用数据段段名
          MOV DS, AX                    ; DS 指向数据段
          LEA DX, HELLO
          MOV AH, 09
          INT 21H                       ; 屏幕输出字符串
          MOV AX, 4C00H
          INT 21H                       ; 结束程序，返回 DOS
          END START                     ; 程序结束，程序从 START 标号处开始执行
```

4.2.3 变量定义

变量定义的伪指令格式为：

```
[变量名]    DB/DW/DD   操作数 [,操作数]
```

其中：

DB 定义字节，其后每个操作数分配一字节并赋值。

DW 定义字，其后每个操作数分配一个字并赋值。

DD 定义双字，其后每个操作数分配两个字并赋值。

在变量定义中，变量名是可选项，它的物理意义是变量定义所分配空间的首地址，其段址隐含由其所处段来代表，所以，实际上与变量名相关联的是段内地址，没有变量名的语句就只有分配空间的作用；关键字 DB/DW/DD 规定了分配单位（字节/字/双字）或存储方式（字节序，字节/字/双字），DB/DW/DD 还为变量建立了 BYTE（字节）、WORD（字）和 DWORD（双字）的数据类型概念；操作数可以是常数、表达式、字符串、问号（？）（即预留存储空间）、变量或者标号等，不管操作数是什么形式，其最终都是汇编程序能直接得到或通过计算得到的值，汇编程序根据操作数决定空间的分配和初始化。

根据操作数形式，变量定义常有以下 7 种使用方法。

（1）使用数值常数定义变量

```
例如：K1 DB 5AH              ; 定义字节变量，初值为 5AH
      K2 DW 55AAH            ; 定义字变量，初值为 55AAH
      K3 DD 123455AAH        ; 定义双字变量，初值为 123455AAH
```

K1、K2、K3 的存储格式如图 4.4（a）所示。

（2）使用字符和字符串定义变量

```
例如：CH   DB 'A'            ; 定义字节变量，初值为字符 A 的 ASCII 码
      STR DB 'HELLO'         ; 分配 5 字节存储字符串，定义字节变量指向字符串首址
      OK   DW 'OK'           ; 分配 2 字节，按字方式存储字符串，
                             ; 并定义字变量指向字符串首址
```

CH、STR、OK 的存储格式如图 4.4（b）所示。特别需要注意 OK 的定义，字符串‘OK’决定了空间大小，DW 决定了存储格式（字节序），为了与 DW 相对应，字符串的长

度应是偶数。在编程中，像 OK 这样的定义容易引起误解，应尽量避免。

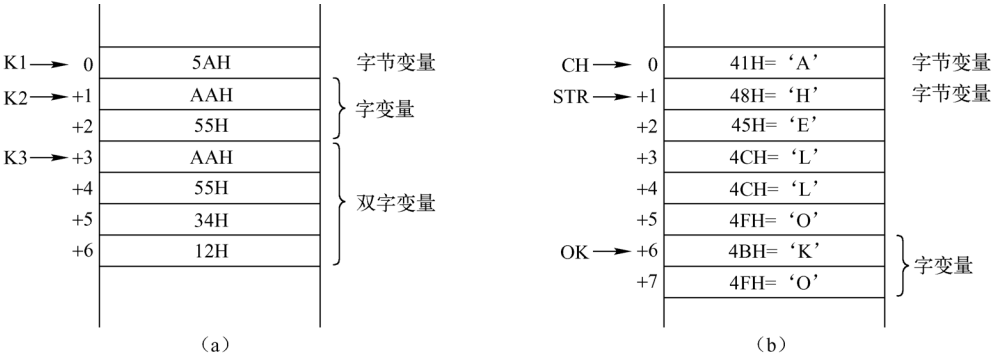


图 4.4　变量存储格式（一）

（3）使用问号（？）定义变量

问号（？）表示不预设初值，只分配空间。

```
例如：V1 DB ？     ；定义字节变量，不设定初值
      V2 DW ？     ；定义字变量，不设定初值
      V3 DD ？     ；定义双字变量，不设定初值
```

（4）使用变量或标号定义变量

变量定义中的操作数可以是其他变量或标号，表示对变量或标号所代表的段内地址的引用。

```
例如：DATA SEGMENT
      STR     DB  '12345'        ；定义字节变量 STR 指向字符串分配的空间
      STR_P   DW  STR            ；定义字变量 STR_P，其初值是 STR 代表的地址
      DATA ENDS
```

STR 和 STR_H 的存储格式如图 4.5（a）所示。

（5）使用表达式定义变量

变量定义中的操作数可以使用表达式，表达式是语言范畴的表示形式，它的计算是由汇编程序处理的，即汇编后变量的初值是表达式计算出的值。

```
例如：CHR DB  '9' + 1      ；定义字节变量，初值是'9'后一个字符的 ASCII 码
      W1   DW 100*3-10     ；定义字变量，初值是 100*3-10=290
      L1   DD 0            ；定义双字变量，初值是 0
      P1   DW L1+2         ；定义字变量，初值是 L1 地址加 2，即指向双字中的高字
```

存储格式见图 4.5（b）。

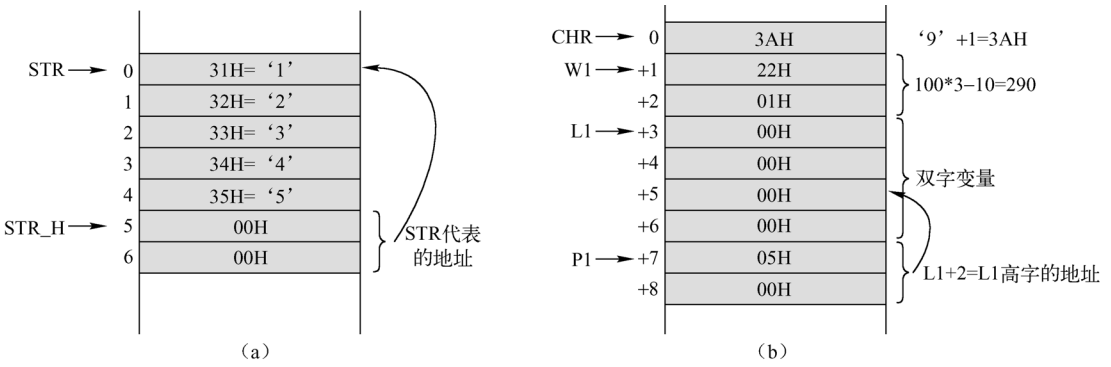


图 4.5　变量存储格式（二）

如果表达式不能在汇编过程中计算得出结果，则表达式就是错误的。

例如：Da1 DW AX+1　　；错误：AX 寄存器内容是执行时的概念
　　Da2 DW [var]　　；错误：变量 var 的内容只有执行时才知道

（6）使用多个操作数定义变量

变量定义中的操作数可以有多个，使用逗号分隔。汇编程序按操作数定义次序逐个按定义方式分配空间并赋初值。

例如：PRO　　DB ‘prompt’，0dH，0aH，‘$’
　　等价于
　　PRO　　DB　‘prompt’
　　DB　0dH
　　DB　0aH
　　DB　$’

```
又如：ARY     DW 0, 1, 2, ?, ?                    ; 对应 5 个操作数逐一分配
                                                  ; 5 个字（连续 10 字节）并
                                                  ; 赋初值，空间首地址是 ARY。
                                                  ; 即 5 个字组成的数组
      ARY_P   DW ARY, ARY+2, ARY+4, ARY+6, ARY+8  ; 同上，5 个字分配 ARY 的 5 个字的地址
      ARY_L   DW ARY_P−ARY, (ARY_P−ARY)/2         ; 分配两个字，第一个字是数组
                                                  ; ARY 的空间大小；第二个字是
                                                  ; 数组元素个数
```

（7）使用 DUP 表达式

变量定义中的操作数可以是 DUP 表达式，用于表示操作数的重复。格式为：

n DUP (操作数)

其中：*n* 是重复的次数，DUP 是关键字，小括号内是要重复的内容。

例如：ARY DW 5 dup(?)
　　等价于
　　ARY DW ?, ?, ?, ?, ?

DUP 表达式常用于对较大空间的分配。

例如：STACK SEGMENT STACK

```
DW 100H DUP(?)  ; 分配 256 个字共 512 字节的空间
STACK ENDS
```

不难想象，如果没有 DUP 表达式，就需要重复写 256 个问号，对程序员来说是个灾难。

DUP 表达式允许嵌套，这极大地丰富了 DUP 的功能和作用。

```
例如：ARRAY DB  2 DUP (2 DUP (1, 2, 3))
                        ; 从里向外解开：
                        ; 2 DUP ((1,2,3),(1,2,3))
                        ; ((1,2,3),(1,2,3)),((1,2,3), (1,2,3))
```

在汇编语言中，DUP 表达式常用于分配数组空间。

4.2.4 常数、变量、标号、表达式

1．常数

常数是没有任何属性的纯数值。在汇编时常数的值已经确定，并且在程序的运行过程中，常数的值不会改变。

常数分为两种类型：数值型常数和字符串常数。

（1）数值型常数

数值型常数可以表示为二进制、八进制、十进制或十六进制形式，分别以字母 B、Q 或 O、D、H 结尾。

例如，常数 102 可以表示为：

二进制数：01100110B。

八进制数：146O 或 146Q。

十进制：102D 或 102，默认是十进制，结尾字母 D 可以省略。

十六进制：66H。

为了与标识符区分，十六进制常数如果以 A～F 开始，前面要加“0”引导。例如，0A0H、0B1H、0C2H、0D3H、0E4H、0F5H。

（2）字符串常数

字符串常数是用单引号括起来的字符或字符串，例如：‘A’、‘12345’、‘hello’等。字符在计算机内以 ASCII 码存储。

2．变量

变量用于定义存放在存储器单元中的数据，程序运行时可以随时修改。为了便于对变量进行修改，需要为其起一个名字，这就是变量名。变量名是存放数据的存储单元的符号地址。变量可以使用变量定义伪指令 DB/DW/DD 来定义。

变量有三方面的属性：

（1）段：变量所处的逻辑段。

（2）偏移地址：变量的段内地址。

（3）类型：变量所占的字节数。

其中，变量的类型包括字节类型（BYTE）、字类型（WORD）和双字类型（DWORD），分别占 1、2、4 字节。变量的类型在变量定义时确定，伪指令 DB、DW、DD 确定变量的类型分别是 BYTE、WORD 和 DWORD。

3．标号

标号是指令语句的符号地址，代表该指令在内存中的位置。

标号常和转移指令相配合，作为转移的目标地址。

例如：　NEXT:　……

……

LOOP NEXT

与变量类似，标号也有三方面的属性：

（1）段：标号所处的逻辑段。

（2）偏移地址：标号的段内地址。

（3）类型：指出该标号是在本段内引用还是在其他段中引用。

其中，标号的类型包括段内转移和段间转移，使用“NEAR”和“FAR”标识。

4．地址计数器

汇编程序在汇编过程中，为了确定变量或标号的偏移地址设置了一个地址计数器，为数据或指令分配存储空间后就将计数器加上其空间的大小（字节数），变量或标号的偏移地址就是地址计数器的当前值。例 4.4 是地址计算的过程示例分析。

【例 4.4】 分析如图 4.6 所示程序段的地址计算过程。

```
    DATA        SEGMENT
①   OK          DB          'OK' ②
③   D1          DW          ④ 1,  ⑤ 2,  ⑥ 3
    DATA        ENDS
    CODE        SEGMENT
                ⑦ ASSUME CS:CODE,DS:DATA  ⑧
⑨   START:      ⑩ MOV AX,DATA
                ⑪ MOV DS,AX
                ⑫ MOV CX,0
⑬   LP:         ⑭ NOP
                LOOP LP
                MOV AX,4C00H
                INT 2LH
    CODE        ENDS
                ENE STAR
```

图 4.6　地址计算过程

分析：

① 段开始，计数器清零（段内地址总是从 0 开始）。变量 OK 的偏移地址=0，即计数器的当前值。

② 为‘OK’分配了两字节，计数器加 2，当前值=2。

③ 变量 D1 的偏移地址=2。

④ 计数器加 2，当前值=4。

⑤ 计数器加 2，当前值=6。

⑥ 计数器加 2，当前值=8。
⑦ 段开始，计数器清零，当前值=0。
⑧ 伪指令不占空间，计数器不变。
⑨ 标号 START 的偏移地址=0。
⑩ 计数器加 3（本指令字节数），当前值=3。
⑪：计数器加 2，当前值=5。
⑫：计数器加 3，当前值=8。
⑬：标号 LP 的偏移地址=8。
⑭：计数器加 1，当前值=9。
在汇编程序中，可以使用‘$’符号引用地址计数器的当前值，称为当前地址。
例如：ARY　DW 100H DUP (0)
　　ARY_SIZE DW $-ARY　；当前地址-ARY 地址=数组 ARY 所占字节数

5．变量和标号的声明

变量和标号还可以用伪指令 LABEL 进行声明。
伪指令 LABEL 的格式为：

```
名字　LABEL　类型
```

在 LABEL 语句中，“名字”是变量或标号的名字；“类型”用于确定变量或标号的类型属性，包括声明变量的类型（BYTE、WORD 或 DWORD）和声明标号的类型（NEAR 或 FAR）。变量或标号的段属性也很容易确定，就是伪指令 LABEL 所处的段。变量或标号的偏移地址属性如何确定呢？实际上，就是当前地址，即地址计数器的当前值。

```
例如：V1　LABEL　BYTE　　；声明 BYTE 变量 V1，偏移地址为当前地址，伪指令，
　　;　　地址计数器不变
　　V2　LABEL　DWORD　　；声明 DWORD 变量 V2，偏移地址为汇编程序，伪指令，
　　;　　地址计数器不变
　　VAR　DW　1234H　　；定义 WORD 变量 VAR，偏移地址为当前地址，
　　;　　地址计数器加 2
```

在例中，V1 是 BYTE 类型，V2 是 DWORD 类型，VAR 是 WORD 类型，但它们的地址相同，也就是同一个数据空间可以通过不同类型的变量访问，如果要当成字来访问就使用 VAR，要当成字节来访问就使用 V1。
用 LABEL 声明的标号与上述类似。

```
例如：Route_F LABEL FAR　；声明 FAR 类型标号，偏移地址为当前地址，
　　;　　伪指令，地址计数器不变
　　Route:　NOP　　；定义标号，类型隐含为 NEAR，偏移地址为当前地址
　　……　　;
```

在例中，标号 Route_F 和标号 Route 指向相同位置，但类型不同。

6．表达式

表达式是操作数的常见形式。从计算机语言层面上讲，表达式具有两个重要作用：首

先，表达式是由汇编程序计算的，对编程极为便利；其次，表达式往往比直接使用值具有更丰富的逻辑意义，增加了程序的可读性。

例如：CHR DB ‘9’+1

一般地，程序员很难记住 ASCII 码表中‘9’后面的字符是什么，与其去查表，不如用表达式“‘9’+ 1”方便；而且从表达式很容易理解程序员就是想表达‘9’后面的字符。所以，在汇编语言程序设计中应尽量发挥表达式的优势，但必须要注意的是：表达式的运算不是在程序运行时由 CPU 完成的，而是在汇编过程中由汇编程序计算确定的，并将其结果作为操作数参加指令执行，完成汇编语句所规定的操作。

在汇编语言中，表达式由常数、变量、标号及连接它们的运算符组成。MASM 允许的表达式分为数值表达式和地址表达式。

（1）数值表达式

数值表达式的结果是数值，一般由常数和运算符构成。数值表达式作为指令操作数时，汇编程序将其翻译为立即数。

例如：
```
‘9’+1          ; 字符 9 的 ASCII 码为 39H，再加 1
offset VAR +5  ; VAR 是变量，offset 是取变量偏移地址的运算符
```

（2）地址表达式

地址表达式的结果表示存储单元的地址，它可由标号、变量名和由括号括起来的基址或变址寄存器组成。地址表达式作为指令操作数时，汇编程序将其翻译为直接寻址或间接寻址的存储器操作数。

例如：
```
VAR+5          ; VAR 是变量
[BX+1]         ; 基址寻址表达式
```

7．运算符

构成表达式的运算符主要分为算术运算符、逻辑和移位运算符、关系运算符、数值回送运算符和属性运算符五大类。

（1）算术运算符

算术运算符规则说明如表 4.6 所示。

表 4.6　算术运算符

运　算　符		运 算 结 果	实　　例
符　　号	名　　称		
+	加	和	1+1=2；‘0’+1=‘1’
-	减	差	2-1=1；‘1’-1=‘0’
*	乘	积	2*5=10
/	除	商	10/2=5；10/3=3
MOD	取模	余数	12 MOD 3=0；12 MOD 5=2

（2）逻辑和移位运算符

逻辑和移位运算符规则说明如表 4.7 所示。

表 4.7　逻辑和移位运算符

运　算　符		运 算 结 果	实　　例
符　　号	名　　称		
AND	与	布尔运算“与”结果	9FH AND 80H = 80H
OR	或	布尔运算“或”结果	1FH OR 80H = 9FH
NOT	非	布尔运算“非”结果	NOT 0AAH = 55H
XOR	异或	布尔运算“异或”结果	9FH XOR 80H = 1FH
SHL	左移	左移后的二进制数	1FH SHL 1 = 3EH
SHR	右移	右移后的二进制数	9FH SHR 2 = 27H

要注意的是，汇编语言的有些运算符和指令助记符相同，如逻辑和移位运算符与相关指令运算符。但它们有本质的不同：指令助记符代表指令是在执行过程中由 CPU 执行的，而运算符是在汇编过程中由汇编程序处理的。

例如：AND AL, 9FH AND 80H　　；前一个 AND 是指令，而后一个 AND 是运算符，
　　；经过汇编后，指令变为：AND AL,80H
　　OR AL, BL OR 80H　　；错误：后一个 OR 运算符构成的表达式汇编程序
　　；　　无法处理

（3）关系运算符

关系运算符规则说明如表 4.8 所示。

表 4.8　关系运算符

运　算　符		运 算 结 果	实　　例
符　　号	名　　称		
EQ	等于	结果为真，输出全“1”； 结果为假，输出全“0”	8 EQ 100B =全“0”
NE	不等于		8 NE 100B =全“1”
LT	小于		8 LT 1000B =全“0”
LE	小于等于		8 LE 1000B =全“1”
GT	大于		8 GT 1000B =全‘0’
GE	大于等于		8 GE 1000B =全‘1’

（4）数值回送运算符

常用的数值回送运算符包括以下 6 种。

① SEG：回送变量或标号的段地址。

格式：SEG 变量 / 标号

例如：MOV AX, SEG message　　；取变量 message 的段址传送到 AX 寄存器

② OFFSET：回送变量或标号的偏移地址。

格式：OFFSET 变量 / 标号

例如：MOV SI, OFFSET message　　；取变量 message 的偏移址传送到 SI 寄存器

这条指令与指令“LEA SI, message”在功能上是等价的，它们的不同之处仍然在于汇编过程和执行过程不同。“OFFSET message”是表达式，由汇编程序计算出偏移地址，是一个立即数，并通过 MOV 指令传送到 SI 寄存器；而“LEA SI, message”的取地址工作是 LEA

指令的功能，是在执行时由 CPU 完成的，源操作数的寻址方式是直接寻址。

③ TYPE：回送以字节数表示的变量或标号的类型。

格式：TYPE 变量 / 标号

对于变量，BYTE 类型返回 1，WORD 类型返回 2，DWORD 类型返回 4；对于标号，NEAR 类型返回−1，FAR 类型返回−2；对于常数，返回 0。

```
例如：ARY     DW 100 DUP(?)
      DAT    DB 1, 2, 3, 4
      ……
      ADD    SI, TYPE    ARY        ; TYPE ARY 返回 2，即 ADD SI, 2
      ADD    BX, TYPE    DAT        ; TYPE DAT 返回 1，即 ADD BX, 1
```

④ LENGTH：回送分配给变量的单元数。

格式：LENGTH 变量

对于使用 DUP 定义的变量，返回分配给该变量的单元数；否则返回 1。

```
例如：ARY     DW 100 DUP(?)
      DAT    DB 1, 2, 3, 4
      ……
      MOV CX, LENGTH ARY        ; LENGTH ARY 返回 100
      MOV CX, LENGTH DAT        ; LENGTH DAT 返回 1
```

⑤ SIZE：回送分配给变量的字节数。

格式：SIZE 变量

SIZE 运算符返回的分配给变量的字节数，它是 TYPE 回送的类型与 LENGTH 回送的单元数的乘积。

```
例如：ARY     DW 100 DUP(?)
      DAT    DB 1, 2, 3, 4
      ……
      MOV CX, SIZE ARY          ; SIZE ARY 返回 200=2(TYPE)*100(LENGTH)
      MOV CX, SIZE DAT          ; SIZE DAT 返回 1=1(TYPE)*1(LENGTH)
```

⑥ LOW 和 HIGH：分别返回表达式的低字节和高字节

格式：LOW / HIGH 表达式

```
例如：MOV AL, LOW 1234H        ; LOW 1234H 返回 34H
      MOV AH, HIGH 1234H       ; HIGH 1234H 返回 12H
```

（5）属性运算符

属性运算符包括以下两种。

① SHORT

SHORT 用来定义 JMP 指令的短跳转属性。

```
例如：JMP SHORT NEXT            ; 定义 JMP 使用 8 位相对转移
```

② PTR

格式：类型 PTR 变量 / 标号

其中，对于变量，类型有 BYTE、WORD 和 DWORD 共 3 种类型属性；对于标号，类型有 NEAR 和 FAR 两种类型属性。

PTR 用于对变量或标号的强制类型说明，以保证操作数的准确性（即无二义性）和一致性。

例如：

```
TWO_BYTE DW   1234H                ; 定义字变量 TWO_BYTE，类型为 WORD
……
MOV   BX, TWO_BYTE                 ; 正确：BX 是 16 位寄存器，字属性，操作数类型相符
MOV   BL, TWO_BYTE                 ; 错误：BL 是 8 位寄存器，字节属性，操作数类型不相符
MOV   BL, BYTE PTR TWO_BYTE        ; 正确：将 TWO_BYTE 强制为 BYTE 类型
MOV   [DI], 10                     ; 错误：二义性，无法确定 DI 间接寻址指向的是字还是字节
MOV   WORD PTR [DI], 10            ; 正确：强制说明 DI 间接寻址指向的是字，常数 10
                                   ; 也随之变为字（000AH）
```

4.2.5 符号定义

符号定义伪指令用来将常数、变量、标号、表达式等用一特定的符号名称来表示。

符号定义伪指令格式为：

```
符号名称 EQU 表达式
```

或

```
符号名称 = 表达式
```

等号“=”和 EQU 的区别在于：EQU 定义的符号名称不允许重复定义，而等号“=”允许重复定义。

符号定义语句又称为赋值语句，可以理解为将 EQU 或等号“=”右边表达式的值赋给左边的符号。符号定义是伪指令，汇编程序并不分配空间，只是记住左边的符号名称和右边表达式的对应关系，在处理后面的语句时，如果出现了定义的符号名称，就先将符号名称用其关联表达式进行替换，然后再处理。

【例 4.5】 符号定义示例。

```
ARY_INIT     EQU     ?
ARY_MAX      EQU     100
ARY          DW      ARY_MAX dup (ARY_INIT)
ARY_TYPE     EQU     2
ARY_SIZE     EQU     ARY_MAX* ARY_TYPE
ARY_COUNT    EQU     ARY_MAX
ARY_SEG      EQU     SEG ARY
ARY_OFF      EQU     OFFSET ARY
ARY_IND      EQU     [BX+ARY]
```

在例中，使用 DUP 表达式定义了一个字数组 ARY，DUP 的重复次数和重复内容引用了符号 ARY_MAX 和 ARY_INIT，如果需要改变数组初值和最大空间数，只需要修改 ARY_INIT 或 ARY_MAX 的定义。其他符号定义给出了对常数、变量、表达式的符号定义实例。这些符

号定义可以从其名字理解其含义，例如 ARY_SIZE 是 ARY 的空间大小，ARY_COUNT 是 ARY 的单元数，ARY_SEG 是 ARY 的段址，ARY_SEG 是 ARY 的偏移地址等。

在例中，ARY_IND 对使用 BX 的基址寻址操作数进行了符号定义，方便对 ARY 数组元素的访问，例如：

```
MOV BX, 0
MOV AX, ARY_IND        ; 取 ARY 的第 0 个元素
ADD BX, ARY_TYPE
MOV AX, ARY_IND        ; 取 ARY 的第 1 个元素
```

从上面的实例可以看出，使用符号定义的几个优点。

① 使用符号定义便于编程，可以通过符号简化对表达式的引用，特别是对需要重复引用的表达式，并且可以保证表达式重复引用的一致性；

② 使用符号定义可以增加程序的可读性，使用符号名称可以一目了然地了解表达式的含义；

③ 使用符号定义可以增强程序的可维护性，当需要修改时，一般只需要修改符号定义，而程序中对符号的引用不需要修改。

4.2.6 地址对齐

地址对齐伪指令主要有 ORG 伪指令和 EVEN 伪指令。

（1）伪指令 ORG 的格式为：

```
ORG 表达式
```

ORG 的功能是使下一变量或指令的地址成为表达式的值。

（2）伪指令 EVEN 的格式为：

```
EVEN
```

EVEN 的功能是使下一变量或指令的地址成为偶数。

例如：

```
DATA    SEGMENT
                        ; 段开始，地址计数器清零
        ORG 50H         ; 使下一变量地址成为 50H，当前地址=50H
VAR1    DW 2000H        ; VAR1 地址是 50H，分配一个字，当前地址=52H
VAR2    DB 11H          ; VAR2 地址是 51H，分配一个字节，当前地址=53H
        EVEN            ; 使下一变量为偶数地址，当前地址为奇数需加 1，
                        ; 当前地址=54H
VAR3    DW 1234H        ; VAR3 地址是 54H
DATA    ENDS            ; DATA 段总长 56H
```

ORG 伪指令常用来保留空间，在上例中，“ORG 50H”又可理解为保留 80 字节；EVEN 伪指令常用来优化存储，保证数据按字边界对齐，这样对字数据的访问速度最快。

4.2.7 结构定义

结构是汇编语言提供的复合数据类型的说明方法，它可以将不同类型的数据组合在一起，便于编程使用。

1．结构类型的说明

用 STRUCT 和 ENDS 可以把一系列数据定义语句括起来作为一种新的用户定义的结构类型。其一般说明格式如下：

```
结构名    STRUC
          数据定义语句序列
结构名    ENDS
```

例如：

```
PERSONAL   STRUC
xNAME      DB 'ZHANG'      ; 字段 xNAME 相对于结构的偏移量是 0
xAGE       DB ?            ; 字段 xAGE 相对于结构的偏移量是 5
xHIGH      DB ?            ; 字段 xHIGH 相对于结构的偏移量是 6
xWEIGHT    DW ?            ; 字段 xWEIGHT 相对于结构的偏移量是 7
PERSONAL   ENDS
```

在 STRUC 说明中，结构名是自定义类型的名字，其中的各个数据定义语句是结构中字段的定义，字段名是该字段相对于结构的偏移量的符号名称。

2．结构变量的定义和赋值

在定义某个结构类型后，就可以说明该结构类型的变量了。它的说明形式与前面介绍的简单数据类型的变量说明基本上一致。其定义格式如下：

```
[变量名]   结构名    <[字段值表]>
```

① 变量名即为该结构类型的变量名；

② 字段值表是给字段赋初值，中间用逗号“,”分开，其字段值的排列顺序及类型应与该结构说明中的各字段相一致；

③ 如果结构变量中某字段用默认值，可以不赋初值；如果所有字段都如此，则可省去字段值表，仅保留一对尖括号“<>”。

例如：

```
DATA     SEGEMNT
WANG     PERSONAL   < 'WANG', 17 , 170, 60>     ; 定义 WANG 变量并给各个字段赋值
ZHANG    PERSONAL   < , 15 , 160, 46>          ; 定义 ZHANG，XNAME 缺省
         PERSONAL   <>                         ; 分配结构空间，各个字段不赋值
DATA     ENDS
```

3．结构字段的使用

对于结构中的字段可以使用点扩展方式访问，形式是：“结构变量.字段名”。

例如：MOV AX, WANG . xWEIGHT ; 有效地址：WANG 的偏移地址 + 7

MOV BX, [SI]. xWEIGHT ; 有效地址：SI + 7

汇编程序在处理“结构变量.字段名”时，是将其看成一个地址表达式，“.字段名”会变成加上字段名所代表的字段相对于结构的偏移量。这种方式提供了与高级语言的字段引用

方式完全一致的方法，便于编程使用。

4.3 程序的基本结构

4.3.1 基本的程序框架

一个汇编源程序的基本框架如下：

```
DATA    SEGMENT
        ……                  ; 存储数据的数据段
DATA    ENDS
STACK   SEGMENT STACK
        ……                  ; 堆栈段
STACK   ENDS
CODE    SEGMENT
        ASSUME CS:CODE, DS:DATA, SS:STACK
                            ; 段指定
START:  MOV AX, DATA
        MOV DS, AX          ; 初始化 DS，DS 指向 DATA 段

        ……                  ; 核心程序

        MOV AX, 4C00H
        INT  21H            ; 4CH 号系统功能调用：返回操作系统
CODE    ENDS
        END START           ; 程序结束，程序从 START 标号开始执行
```

有关程序的基本框架有以下几点说明：

（1）一个汇编语言程序一般具有代码段、数据段和堆栈段，有时还可能会有附加段。

（2）ASSUME 只是用于建立段和段寄存器的关联，DS、ES（有时还包括 SS）需要在程序执行开始时，用指令设置段址。

（3）使用 END 结束程序，并由其后的标号指出程序的入口。

（4）程序需要使用 4CH 号系统功能调用返回操作系统。

“INT 21H”是 DOS 通过软中断指令向高层提供的服务，称为系统功能调用，由(AH)指定具体的服务类别，称为功能号。除 4CH 号功能调用外，常用的还有以下 4 种：

① 01H 号：带回显的键盘输入。

```
例如：MOV AH, 1   ; 入口：(AH)=1 功能号
      INT  21H    ; 1 号功能调用
                  ; 出口：(AL)中是从键盘输入字符的 ASCII 码
```

② 02H 号：在屏幕上显示字符。

```
例如：MOV AH, 2       ; 入口：(AH)=2 功能号
      MOV DL, 'A'     ;       (DL)中是要显示的字符
      INT  21H        ; 2 号功能调用
                      ; 出口：无
```

③ 08H 号：不带回显的键盘输入。

```
例如：MOV AH, 8          ; 入口：(AH)=8 功能号
      INT  21H           ; 8 号功能调用
                         ; 出口：(AL)中是从键盘输入字符的 ASCII 码
```

④ 09H 号：显示以“$”字符结束的字符串。

```
例如：MOV AH, 9                    ; 入口：(AH)=9 功能号
      MOV DX, OFFSET STRING        ;       DS:DX 指向字符串的首地址
      INT  21H                     ; 9 号功能调用
                                   ; 出口：无
```

4.3.2 程序结构

程序结构对于程序设计极为重要，不讲究结构的程序就如一团乱麻难以理清头绪。结构较好的程序逻辑严谨、流程清晰。对于高级语言，程序的基本结构已经包含在语言的设计之内；而对于汇编语言编程，程序员必须自己完成对程序结构的控制，因此必须对程序结构给予更多的关注。

无论是高级语言还是汇编语言，程序结构都是一样的，即顺序、分支、循环三个基本结构以及子程序结构，如图 4.7 所示。

（1）顺序结构：控制程序按指令顺序逐条向下执行。

（2）分支结构：控制程序根据条件选择不同的分支路径。

（3）循环结构：控制程序有限次地重复执行。

（4）子程序结构：控制程序调用子程序并在子程序完成后返回继续执行。

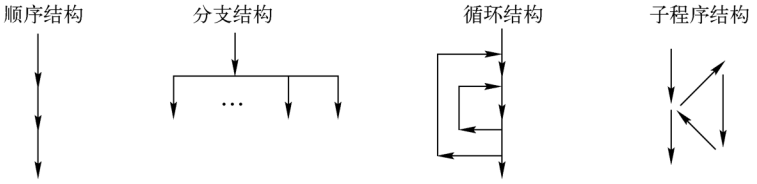


图 4.7　程序结构

4.3.3 顺序结构

前面已经提到，CS:IP 总是指向下一条要执行的指令，CPU 从内存取出指令后 IP 将加上指令的长度，使之自动指向下一条指令。这样就使控制程序按照地址空间上的次序逐条指令地执行，这就是顺序结构，即程序的执行顺序就是指令的编写顺序。

【例 4.6】 计算 Z=X·Y，X 和 Y 是 32 位无符号数，结果 Z 是 64 位的。

因为 8086 没有直接的 32 位乘法指令，所以需要将其拆分通过 16 位乘法指令计算。将 X 分为高 16 位 XH 和低 16 位 XL，Y 也分为 YH 和 YL，如图 4.8 所示，可得到如下结果：

$$X \cdot Y=YH \cdot YH \cdot 2^{32}+YH \cdot XL \cdot 2^{16}+YL \cdot XH \cdot 2^{16}+XL \cdot YL$$

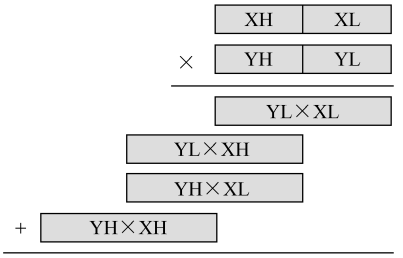


图 4.8　32 位乘法算法示意图

源程序如下：

```
DATA    SEGMENT
X       DD   12345678H        ; 定义双字 X
XL      EQU  WORD PTR X       ; XL 是 X 的低字的符号定义
XH      EQU  WORD PTR X+2     ; XH 是 X 的高字的符号定义
Y       DD   55AA55AAH        ; 定义双字 Y
YL      EQU  WORD PTR Y       ; YL 是 Y 的低字的符号定义
YH      EQU  WORD PTR Y+2     ; YH 是 Y 的高字的符号定义
Z       DW   0, 0, 0, 0       ; Z 是由 4 个字构成的 64 位数，初值为 0
DATA    ENDS
CODE    SEGMENT
        ASSUME CS:CODE, DS:DATA
START:  MOV AX, DATA
        MOV DS, AX            ; DS 指向 DATA 段
        ; 以下实现 32 位无符号数乘法，计算 Z=X·Y
        MOV AX, YL            ; YL→AX
        MOV BX, AX            ; AX→BX，即(BX)=YL
        MUL XL                ; YL(在 AX 中)*XL，乘积在 DX:AX
        MOV Z, AX
        MOV Z+2, DX           ; 保存 YL*XL 乘积到 Z（最低两字）
        MOV AX, BX            ; BX→AX，即(AX)=YL
        MUL XH                ; YL(在 AX 中)*XH，乘积在 DX:AX
        ADD Z+2, AX
        ADC Z+4, DX
        ADC Z+6, 0            ; 将 YL*XH 乘积向左移 16 位(一个字)与 Z 相加
        MOV AX, YH            ; YH→AX
        MOV BX, AX            ; AX→BX，即(BX)=YH
        MUL XL                ; YH(在 AX 中)*XL，乘积在 DX:AX
        ADD Z+2, AX
        ADC Z+4, DX
        ADC Z+6, 0            ; 将 YH*XL 乘积向左移 16 位(一个字)与 Z 相加
        MOV AX, BX            ; BX→AX，即(AX)=YH
        MUL XH                ; YH(在 AX 中)*XH，乘积在 DX:AX
        ADD Z+4, AX
        ADC Z+6, DX           ; 将 YH*XH 乘积向左移 32 位(两个字)与 Z 相加
        ;
        MOV AX, 4C00H
```

```
        INT 21H                 ; 返回操作系统
CODE    ENDS
        END START
```

4.3.4 分支结构

分支结构是根据条件选择程序流向的重要程序结构。分支结构要求打破程序顺序执行机制，即要修改 IP 或 CS 和 IP 的值，所以转移指令是分支结构的实现基础。另一方面，分支结构中的条件是依靠标志位体现的，主要有 CF、OF、ZF、SP、PF 等，包括产生标志位和测试标志位两个步骤。在设计分支结构程序时，首先应根据处理的问题选择使用比较、测试或算术运算、逻辑运算指令产生标志位，然后根据分支条件选择合适的条件转移指令测试相应的标志位。

通常一条条件转移指令只有两个分支，所以两分支结构是基本的分支结构，如图 4.9 所示。两分支结构包括“if … endif”和“if … else … endif”两种形式，复杂的多分支结构一般都是由这两种分支结构嵌套组合而成的。

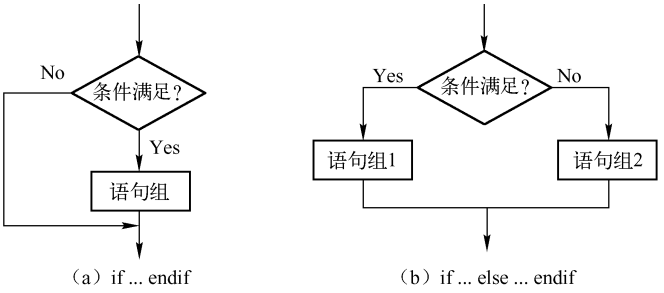


图 4.9　分支结构示意图

在编写汇编语言程序时，分支结构要求清晰，杂乱无章的跳转会使程序非常混乱，而且应尽量避免长距离的转移。

【例 4.7】 X、Y、Z 是三个 16 位有符号数，编程将其中的最大者送入 MAX 单元。

源程序如下：

```
DATA    SEGMENT
X       DW  ?
Y       DW  ?
Z       DW  ?
MAX     DW  ?
DATA    ENDS
CODE    SEGMENT
        ASSUME CS:CODE, DS:DATA
START:  MOV   AX, DATA
        MOV   DS, AX            ; DS 指向 DATA 段
                                ; 以下实现 Z = MAX (X, Y, Z)
        MOV   AX, X             ; X→AX
        CMP   AX, Y
        JGE   BIGGER            ; AX>=Y 则转移到 BIGGER
```

```
        MOV   AX, Y              ; Y→AX
BIGGER: CMP   AX, Z              ; (AX)中是 X 和 Y 的较大值，继续与 Z 比较
        JGE   BIGGEST            ; AX>=Y 则转移到 BIGGEST
        MOV   AX, Z              ; Z→AX
BIGGEST:MOV   MAX, AX            ; (AX)中是最大值，AX→MAX

        MOV AX, 4C00H
        INT   21H                ; 返回操作系统
CODE    ENDS
        END   START
```

【例 4.8】 编写一程序段，计算下列函数值。其中：变量 X 和 Y 是有符号字变量。

$$Y=\begin{cases} X+10 & X<0 \\ 30X & 0 \quad X<10 \\ X-190 & X \quad 10 \end{cases}$$

源程序如下：

```
DATA    SEGMENT
X       DW  ?
Y       DW  ?
DATA    ENDS
CODE    SEGMENT
        ASSUME CS:CODE, DS:DATA
START:  MOV AX, DATA
        MOV DS, AX               ; DS 指向 DATA 段
                                 ; 以下实现函数计算
        MOV AX, X                ; X→AX
        CMP  AX, 0
        JGE   XGE0               ; X>=0 则转移到 XGE0
        ADD   AX, 10             ; X<0 分支：AX+10，(AX)是函数值
        JMP   YFIN               ; 计算完成转移到 YFIN
XGE0:   CMP   AX, 10
        JL    XLT10              ; X<10 则转移到 XLT10
        SUB   AX, 190            ; X>=10 分支：AX−190，(AX)是函数值
        JMP   YFIN               ; 计算完成转移到 YFIN
XLT10:  MOV   BX, 10
        MUL   BX                 ; 0<=X<10 分支：AX*30，(AX)是函数值
YFIN:   MOV   Y, AX              ; AX→Y
                                 ;
        MOV   AX, 4C00H
        INT   21H                ; 返回操作系统
CODE    ENDS
        END   START
```

如果多分支结构的条件较容易进行可整数化，还可以使用跳转表实现多分支结构。

【例 4.9】 实现下列 C 语言的 switch 语句的汇编实现方法，其中，a 和 b 是有符号的整型变量。

C 程序：

```
switch (a)
{
case 0: b=32;
        break;
case 1:
case 2: b=a+64;
        reak;
case 3: b*=2;
        break;
case 4: b−−;
        break;
case 5:
case 6:
case 7: b=b/2+a;
        break;
default:
       printf(“Error.\n”);
       break;
}
```

汇编程序：

```
DATA    SEGMENT
A       DW  ?
B       DW  ?
TABLE   DW  CASE0, CASE12, CASE12, CASE3
        DW  CASE4, CASE567, CASE567, CASE567
        DW  DEFAULT
        ; 跳转表：9 个分支
ERR     DB   ‘ERROR.’, 0DH, 0AH, ‘$’
DATA    ENDS
CODE    SEGMENT
        ASSUME CS:CODE, DS:DATA
START:  MOV AX, DATA
        MOV DS,  AX             ; DS 指向 DATA 段
        ; 以下实现 SWITCH
        MOV BX, A               ; A→BX
        CMP  BX, 8
        JBE   CASE0TO8
        MOV BX, 8
CASE0TO8:                       ; (BX)规整为 0 到 8
        SHL   BX, 1             ; 跳转表是字类型，
                                ; 下标乘 2
        JMP   TABLE[BX]         ; 由 BX 选择跳转表分支,
                                ; 间接转移
CASE0:  MOV  B, 32
        JMP   CASEEND
```

```
CASE12:  MOV    AX, A
         ADD   AX, 64
         MOV   B, AX
         JMP    CASEEND
CASE3:   SHL    B, 1
         JMP    CASEEND
CASE4:   DEC    B
         JMP    CASEEND
CASE567:
         SAR   B, 1
         MOV AX, A
         ADD   B, AX
         JMP    CASEEND
DEFAULT:
         MOV   AH, 9
         MOV   DX, OFFSET ERR
         INT    21H
CASEEND:
         ;
         MOV   AX, 4C00H
         INT    21H                    ; 返回操作系统
CODE     ENDS
         END   START
```

4.3.5 循环结构

循环结构具有重复执行某段程序的功能。

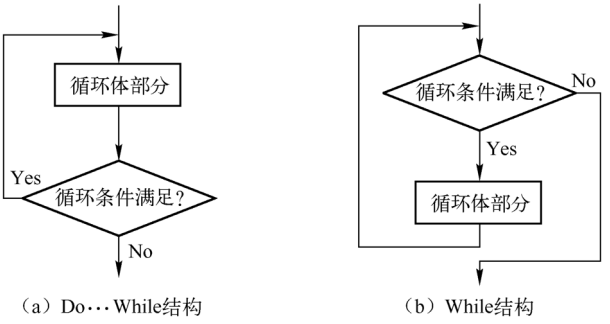
循环结构一般包括以下三个组成部分：

（1）初始化：初始化循环次数以及设置循环体初始值。

（2）循环体：每次循环要执行的程序代码。

（3）循环控制：控制循环继续的条件，由控制转移语句构成。

如图 4.10 所示，循环结构分为前判断结构（Do…While 结构）和后判断结构（While 结构）两种。LOOP 指令实现的是典型的后判断循环，前判断循环可以使用 JMP 和条件转移指令配合实现。



（a）Do…While结构　　（b）While结构

图 4.10　常用的循环结构示意图

【例 4.10】 在屏幕上以十六进制的形式显示字变量。

根据题意，我们需要把 BX 中的内容从左到右每 4 位一组作为一个十六进制数码显示在屏幕上，一共包含 4 个十六进制数码。显然可以采用循环结构，每次循环显示一个十六进制数码，循环次数是 4。每次循环首先使用循环右移指令将 BX 的最高 4 位移到最低 4 位，如图 4.11（a）所示，然后将 BX 的最低 4 位转换得到其对应十六进制数码的 ASCII 码并显示在屏幕上。在 ASCII 码表中，数字字符‘0’～‘9’是连续的，字母字符‘A’～‘Z’也是连续的，并且数字是排在字母前面的。所以，将数 0H～0FH 转换为字符时，可先加上字符‘0’的 ASCII 码，这对于 0～9 就足够了，而对于 0AH～0FH 还要再加上字符‘A’～‘9’之间的字符数进行修正，如图 4.11（b）所示，字符‘A’～‘9’之间的字符数是‘A’−(‘9’+1)。

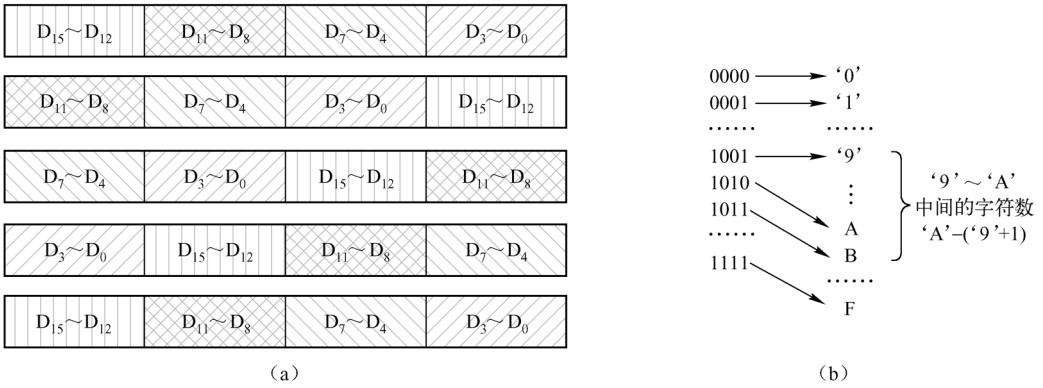


图 4.11　算法示意图

源程序如下：

```
DATA    SEGMENT
A       DW  ?
DATA    ENDS
CODE    SEGMENT
        ASSUME CS:CODE, DS:DATA
START:  MOV  AX, DATA
        MOV  DS, AX              ; DS 指向 DATA 段
        ; 以下将 A 以十六进制形式输出
        MOV  BX, A
        MOV  CH, 4               ; 初始化循环次数
ROTATE: MOV  CL, 4
        ROL  BX, CL              ; 将 BX 循环左移 4 位
        MOV  AL, BL
        AND  AL, 0FH             ; 取出低四位到 AL
        ADD  AL, ‘0’             ; 加上 ‘0’
        CMP  AL, ‘9’             ; 是否是 ‘0’～‘9’
        JLE  PRINTIT      ;
        ADD  AL, ‘A’–(‘9’+1)
                                 ; 调整，加上 ‘9’～‘A’间的字符数
PRINTIT:
        MOV  DL, AL
        MOV  AH, 2
```

```
        INT   21H                ; 2 号功能调用，显示(DL)中的字符
        DEC   CH                 ; 循环次数减 1
        JNZ   ROTATE             ; 循环次数是否到达
        ;
        MOV  AX, 4C00H
        INT   21H                ; 返回操作系统
CODE    ENDS
        END START
```

在循环体中又包含有循环的结构称为多重循环。多重循环的设计方法与单重循环是一致的，但要注意保持层次的清晰性，从外层循环进入内层循环时应重新对内层循环进行初始化，同时内层循环也不要破坏外层循环的环境。在多重循环中应自觉避免类似跨层转移等破坏结构的方法。

【例 4.11】 将首地址为 A 的字数组从大到小排序。

我们使用冒泡排序算法对数组 A 进行排序。冒泡排序是一种交换排序的算法，不需要额外的辅助空间。它从第一个数开始依次对相邻的两个数比较，前一个数小于后一个数（题意要求从大到小排序）则将两个数交换位置，这样处理完全部的数后，数组中最小的数一定排到了最后的位置，比较次数是数组个数减 1，然后重复此过程直到数组排序完成。显然，冒泡排序算法是一个两重循环，内层循环完成数组的两两比较和交换，循环次数是数组剩余个数减 1（既然每次将最小的数已经排到了最后，它就不必再参与下次的比较和交换了）；外层循环重复上述两两比较和交换的过程，循环次数是数组个数减 1。

源程序如下：

```
DATA    SEGMENT
A       DW  100, 30, 78, 99, 15, -1, 66, 54, 189, 256
A_TYPE  EQU 2
A_COUNT EQU 10
DATA    ENDS
CODE    SEGMENT
        ASSUME CS:CODE, DS:DATA
START:  MOV  AX, DATA
        MOV  DS,  AX          ; DS 指向 DATA 段
                              ; 以下将数组 A 从大到小排序
        MOV  CX, A_COUNT-1
                              ; 外层循环次数是数组个数减 1
LOOP1:  MOV  DI,  CX          ; 保存循环次数以免被内层循环破坏
                              ; 内层循环次数正好与外层循环次数相同，无须初始化 CX
        MOV  BX, 0            ; 内层循环的初始状态，从头开始
LOOP2:  MOV  AX, A[BX]
        CMP  AX, A[BX+2]      ; 同后面一个数比较
        JGE  CONTI
        XCHG AX, A[BX+2]      ; 小于后面一个数则交换
        MOV  A[BX], AX
CONTI:  ADD  BX, A_TYPE       ; 下一个数的下标
        LOOP LOOP2
        MOV  CX, DI           ; 恢复外层循环次数
```

```
        LOOP  LOOP1
        ;
        MOV AX, 4C00H
        INT   21H              ; 返回操作系统
CODE    ENDS
        END   START
```

4.4 子程序结构

在程序设计过程中常常把多次引用的相同程序段编成一个独立的程序段，当需要执行这个程序段时，可以使用调用指令调用它。具有这种独立功能的程序段称为过程或子程序。使用子程序可以缩短源程序长度、节省代码空间，也可以提高程序的可读性、可维护性和共享性。

4.4.1 子程序定义

子程序的一般格式如下：

```
子程序名        PROC      [NEAR | FAR]
                …                              ;子程序体
子程序名        ENDP
```

子程序定义需要注意以下几点：

① “子程序名”必须是一个合法的标识符，并且前后二者要一致；

② PROC 和 ENDP 必须是成对出现的关键字，它们分别表示子程序定义开始和结束；

③ 子程序的类型有段内调用（NEAR）、段间调用（FAR）之分，其默认的类型是 NEAR 类型；

④ 如果一个子程序要在不同段中被调用，其类型应定义为 FAR，否则定义为 NEAR；

⑤ 子程序至少要有一条返回指令。返回指令是子程序的出口语句；

⑥ 子程序名是子程序入口的符号地址，实际上它与标号是一样的，也包含三方面属性：段、偏移地址和类型（NEAR 和 FAR）。

编写子程序除了要考虑实现子程序功能的方法外，还要养成书写子程序说明信息的好习惯。其说明信息一般包括以下几方面内容：

① 功能描述；

② 入口和出口参数；

③ 所用寄存器，此项可选，最好采用寄存器的保护和恢复方法，使之使用透明化；

④ 所用额外存储单元，此项可选，可以减少为子程序定义自己的局部变量；

⑤ 子程序所采用的算法，此项可选，如果算法简单，可以不写；

⑥ 调用时的注意事项，此项可选，尽量避免除入口参数外还有其他的要求；

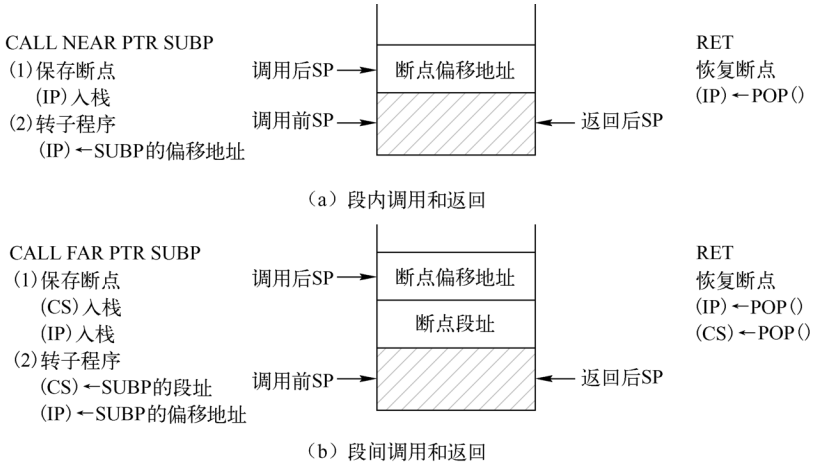
⑦ 子程序的编写者，此项可选，为将来的维护提供信息；

⑧ 子程序的编写日期，此项可选，用于确定程序是否是最新版本。

4.4.2 子程序调用和返回

子程序的调用和返回是由指令 CALL 和 RET 完成的，它们隐含使用堆栈保存断点和恢复断点，如图 4.12 所示。CALL 指令分为段内调用和段间调用，与 CALL 指令相对应，RET 指令也有段内返回和段间返回。

由图可以看到堆栈在子程序调用和返回时的作用，在子程序结构中要特别注意堆栈的使用。



（a）段内调用和返回

（b）段间调用和返回

图 4.12 调用和返回

首先，必须保证 CALL 和 RET 的匹配，不要出现段内的 CALL 和段间的 RET 或段间的 CALL 和段内的 RET 相对应，那样既不能保证正确地转移到子程序，RET 也不能正确地从堆栈中恢复断点。

汇编程序在处理子程序时，根据 PROC 后的子程序类型的说明来确定子程序的类型属性，也确定了是将 RET 助记符翻译为段内返回（NEAR 类型）还是段间返回（FAR 类型）；又可以根据子程序类型，确定对没有 PTR 强制的直接调用（CALL 子程序名）的翻译方法。所以，上述问题就集中到了子程序定义时的子程序类型说明上。在编程时，只有确定了子程序是在本段内调用的情况下才使用 NEAR 类型；对于直接调用，不需要用 PTR 强制说明调用类型，让汇编程序自己选择即可。

其次，在子程序中必须保证堆栈平衡，即进入子程序时 SP 的值与返回时 SP 的值是一样的，这样才能保证 RET 正确返回断点。

4.4.3 环境的保存和恢复

在汇编语言中，子程序只是 CALL 和 RET 指令的执行效果，并没有类似高级语言中作用域的限制。实际上，由于硬件资源只有一套，子程序和主程序访问的是相同的寄存器和内存，面对的是同样的环境。因此就可能出现这样的错误，当主程序调用子程序后，由于子程序对寄存器或内存的修改而破坏了主程序的执行状态。

例如：

```
主程序                                    子程序
    ……                                    ; 将 DX:AX 左移 4 位
```

```
        MOV CX, 8                       SUB1 PROC NEAR
LOOP1:                                      MOV CL, 4
        ……                                  MOV BL, AH
        CALL SUB1                           SHR BL, CL
        LOOP LOOP1   ; 循环                 SHL DX, CL
                                            SHL AX, CL
                                            OR  DL, BL
                                            RET
                                        SUB1 ENDP
```

在例子中，子程序修改了 CX，造成了主程序循环控制（LOOP 指令隐含使用 CX 控制循环）的破坏，变成了死循环。

在子程序设计中应注意环境（主要是寄存器）的保存和恢复，以便当返回调用程序后仍然保持正确的状态继续执行。再延伸一下，在子程序中通过环境的保存和恢复，应该对所有的调用程序都没有影响，这就是子程序的独立性。其实现方法主要是：在子程序开始时，把它要用到的所有寄存器都入栈保存，在返回前再出栈恢复。

在上例中，子程序 SUB1 开始时应保存寄存器 BX 和 CX，RET 指令返回前应恢复寄存器 CX 和 BX，这样就不会对主程序产生影响了。

在使用堆栈保存和恢复寄存器时，应注意以下几点：

① 不能破坏堆栈的平衡，即有多少入栈就应多少出栈；

② 注意堆栈的“先进后出”特性，入栈和出栈的次序应相反；

③ 通常约定在子程序中不保存标志寄存器；

④ 除了用来传递参数（主要是出口参数）的寄存器外，所有使用的寄存器都应保存。

4.4.4 参数的传递

调用程序在调用子程序时需要传递一些数据给子程序，作为子程序运算的原始数据，这些数据称为子程序的入口参数；子程序完成后也可能需要将处理结果返回给调用程序，这些结果称为子程序的出口参数。

调用程序和子程序的参数传递，可以使用三种方法：①通过寄存器传递；②通过存储器传递；③通过堆栈传递。

1．通过寄存器传递参数

通过寄存器传递参数速度快，但参数个数受寄存器数量的限制，适用于参数很少的情况。

【例 4.12】 输入一个十进制数（−32 768～32 767）求绝对值后输出。

程序由主程序、十进制输入子程序 DECIN、十进制输出子程序 DECOUT、求绝对值子程序 GETABS 和回车换行子程序 CRLF 组成。

源程序如下：

```
DATA    SEGMENT
TEMP    DB   5 DUP (0), '$'          ; 十进制输出使用的缓冲区
BUFREAR EQU OFFSET TEMP+5            ; 缓冲区尾，使用缓冲区是从后向前填充的
CR      =    0DH                     ; 回车符
LF      =    0AH                     ; 换行符
```

```
DATA    ENDS
STACK   SEGMENT STACK
        DW   100H DUP (?)          ; 堆栈空间
STACK   ENDS
CODE    SEGMENT
        ASSUME CS:CODE, DS:DATA, SS:STACK
START:  MOV  AX, DATA
        MOV  DS, AX                ; DS 指向 DATA 段
        ; 以下是主程序
MLOOP:  CALL DECIN                 ; 十进制输入, 输入的数在 AX 中
        CALL CRLF                  ; 换行
        OR   AX, AX
        JZ   MLOOPFIN              ; 输入的是 0 则退出
        CALL  GETABS               ; 将(AX)求绝对值
        CALL  DECOUT               ; 十进制输出
        CALL  CRLF                 ; 换行
        JMP   MLOOP
MLOOPFIN:
        ;
        MOV AX, 4C00H
        INT   21H                  ; 返回操作系统
        ;
        ; DECIN: 输入一个十进制数(−32 768～32 767)
        ; 入口：无
        ; 出口: (AX)是输入的数
DECIN   PROC NEAR
        PUSH BX
        PUSH CX
        PUSH DX
        PUSH SI                    ; 保存使用到的所有寄存器
        MOV BX, 0                  ; 0→BX, 开始时数为 0
        MOV SI, 0                      ; 0→SI, 开始时为正数
INLOOP: MOV AH, 8
        INT   21H                      ; 调用 8 号功能，输入字符到(AL)但不显示
        CMP  AL, CR
        JZ     INLOOPEND               ; 回车结束输入，跳出循环
        CMP  AL, '−'
        JNZ    IS0TO9                  ; 不是负号跳到判断数字字符
        OR     BX, BX
        JNZ     INLOOP                 ; BX 非零表示已有数字输入，则输入负号非法
        OR      SI, SI
        JNZ    INLOOP                  ; SI 非零表示已输入过负号，则输入负号非法
        MOV   SI, −1                   ; 最小负数
        MOV   DL, AL
        JMP    SHOWDL                  ; 转移到显示 DL 中字符
IS0TO9: CMP  AL, '0'
        JB     INLOOP                  ; 抛弃非数字字符重新输入
        CMP  AL, '9'
```

```
        JA      INLOOP                  ; 抛弃非数字字符重新输入
        PUSH AX                         ; 乘法隐含使用 AX，保存 AX
        MOV  AX, BX
        MOV   CX, 10
        IMUL  CX
        MOV   CX, AX                    ; (CX)=(BX)*10
        POP    AX                       ; 恢复 AX
        JO      INLOOP                  ; 超过字范围，抛弃输入字符
        MOV   DL, AL                    ; AL→DL，准备显示
        SUB   AL, '0'                   ; 变为 0～9 的数
        MOV AH, 0                       ; 扩展到 AX
        OR    SI, SI
        JZ     ADDINN
        NEG  AX                         ; SI!=0 表示为负数，对(AX)算术求反
ADDINN: ADD CX, AX                      ; (CX)=(BX)*10+(AX)
        JO     INLOOP                   ; 超过字范围，抛弃输入字符
        MOV BX, CX                      ; (BX)是本次输入后的数值
SHOWDL: MOV AH, 2
        INT    21H                      ; 显示 DL 中的字符
        JMP    INLOOP                   ; 继续输入
INLOOPEND:
        MOV  AX, BX                     ; BX→AX，(AX)是输入的数值
        POP    SI
        POP    DX
        POP    CX
        POP    BX                       ; 恢复寄存器内容
        RET
DECIN   ENDP
        ;
        ; DECOUT：以十进制形式输出一个无符号数(字)
        ; 入口：(AX)需要输出的正数
        ; 出口：无
        ; 存储单元：需要使用 TEMP 定义的缓冲区
DECOUT  PROC   NEAR
        PUSH   AX
        PUSH   BX
        PUSH   CX
        PUSH   DX                       ; 保存使用到的所有寄存器
        MOV    BX, BUFREAR              ; BX 初始指向缓冲区尾
OUTLOOP:OR      AX, AX
        JZ      OUTLOOPFIN              ; (AX)=0 则转换结束
        MOV    DX, 0                    ; 将 AX 扩展到 DX:AX
        MOV    CX, 10
        DIV     CX                      ; (AX)=( DX:AX)/10，余数在 DX 中
        ADD    DL, '0'                  ; 将 0～9 转换为字符
        DEC     BX
        MOV    BYTE PTR [BX], DL        ; 将 DL 中的字符保存到缓冲区
        JMP     OUTLOOP
```

```
OUTLOOPFIN:
        MOV   DX, BX
        MOV   AH, 9
        INT   21H                    ; 显示缓冲区中的字符串
        POP   DX
        POP   CX
        POP   BX
        POP   AX                     ; 恢复寄存器内容
        RET
DECOUT  ENDP
        ;
        ; GETABS：求绝对值
        ; 入口：(AX)原数值
        ; 出口：(AX)绝对值
GETABS  PROC NEAR
        OR AX, AX
        JNS   DONOTHING
        NEG  AX
DONOTHING:
        RET
GETABS  ENDP
        ;
        ; CRLF：屏幕换行
        ; 入口：无
        ; 出口：无
CRLF    PROC   NEAR
        PUSH   AX
        PUSH   DX                    ; 保存使用到的所有寄存器
        MOV    DL, CR
        MOV    AH, 2
        INT    21H
        MOV    DL, LF
        MOV    AH, 2
        INT    21H
        POP    DX
        POP    AX                ; 恢复寄存器内容
        RET
CRLF    ENDP
        ;
CODE    ENDS
        END   START
```

2．通过存储器传递参数

当相互传送的参数较多时，可以使用存储器进行传送。在编程时，应尽量避免直接使用变量传递参数，因为这样会造成子程序过度依赖具体的变量而影响子程序的通用性。例 4.13 介绍了一种利用地址表传递参数的方法。

【例 4.13】 利用地址表传递参数的方法求数组累加和。

```
DATA    SEGMENT
ARY     DW    10, 20, 30, 40, 50, 60, 70, 80, 90, 100
COUNT   DW    10                ; 数组单元个数
SUM     DW     ?                ; 累加和
TABLE   DW    3   DUP   (?)     ; 地址表
DATA    ENDS
STACK   SEGMENT STACK
        DW  100H DUP (?)        ; 堆栈空间
STACK   ENDS
CODE    SEGMENT
        ASSUME CS:CODE, DS:DATA, SS:STACK
START:  MOV  AX, DATA
        MOV  DS, AX             ; DS 指向 DATA 段
        ; 以下是主程序
        MOV TABLE, OFFSET ARY
        MOV TABLE+2, OFFSET COUNT
        MOV TABLE+4, OFFSET SUM
                                ; 地址表由 ARY 地址、COUNT 地址、SUM 地址组成
        MOV BX, OFFSET TABLE    ; BX 指向地址表
        CALL PROADD             ; 求累加和
        ;
        MOV AX, 4C00H
        INT   21H               ; 返回操作系统
        ;
        ; PROADD: 求数组累加和
        ; 入口：(BX)是地址表的地址，
        ;          地址表由数组地址、数组单元数地址、累加和地址组成
        ; 出口: 无
PROADD  PROC   NEAR
        PUSH   AX
        PUSH   CX
        PUSH   SI
        PUSH   DI               ; 保存寄存器
        MOV    SI, [BX]         ; 数组地址→SI
        MOV    DI, [BX+2]       ; 数组单元数地址→DI
        MOV    CX, [DI]         ; 取出数组单元数到(CX)
        MOV    DI, [BX+4]       ; 累加和地址→DI
        XOR    AX, AX           ; 0→AX，求和的初值
NEXT:   ADD    AX, [SI]
        ADD    SI, 2
        LOOP   NEXT             ; 循环将所有元素累加，和在(AX)
        MOV    [DI], AX         ; 保存累加和
        POP    DI
        POP    SI
        POP    CX
        POP    AX               ; 恢复寄存器
```

```
        RET
PROADD  ENDP
        ;
CODE    ENDS
        END   START
```

3．通过堆栈传递参数

通过堆栈传递参数的方法是：调用子程序前将参数压入堆栈，子程序再通过访问堆栈空间获得参数。通过寄存器和存储器传递的参数都具有全局性质，因为寄存器和变量是主程序和子程序均可访问的，并不随着子程序的结束而消失。而通过堆栈传递的参数却是局部性的，它们的生命期随着子程序结束后堆栈的复原而结束。

例 4.14 是一个通过堆栈传递参数的实例。

【例 4.14】 利用堆栈传递参数的方法求数组累加和。

由子程序 ProAdd 实现求数组累加和，通过堆栈传递参数，其过程如图 4.13 所示。源程序如下：

```
DATA    SEGMENT
ARY     DW    10, 20, 30, 40, 50, 60, 70, 80, 90, 100
COUNT   DW    10                ; 数组单元个数
SUM     DW     ?                ; 累加和
DATA    ENDS
STACK   SEGMENT STACK
        DW  100H DUP (?)        ; 堆栈空间
STACK   ENDS
CODE    SEGMENT
        ASSUME CS:CODE, DS:DATA, SS:STACK
START:  MOV   AX, DATA
        MOV   DS, AX            ; DS 指向 DATA 段
        ; 以下是主程序
        MOV   AX, OFFSET ARY
        PUSH  AX                ; 第一个参数入栈：ARY 地址
        PUSH  COUNT             ; 第二个参数入栈：COUNT
        CALL  PROADD            ; 求累加和
        MOV   SUM, AX           ; 保存累加和
        ;
        MOV   AX, 4C00H
        INT   21H               ; 返回操作系统
        ;
        ; PROADD: 求数组累加和
        ; 入口：从堆栈中传递数组地址、单元数
        ; 出口: (AX)返回累加和
PROADD  PROC  NEAR
        PUSH  BP                ; 保存寄存器
        MOV   BP, SP            ; 当前堆栈指针→BP，BP 指向此时栈顶
        PUSH  CX
        PUSH  SI                ; 保存寄存器
```

```
        MOV   SI, [BP+6]          ; 取第一个参数：数组地址→SI
        MOV   CX, [BP+4]          ; 取第二个参数：单元→CX
        XOR   AX, AX              ; 0→AX，求和的初值
NEXT:   ADD   AX, [SI]
        ADD   SI, 2
        LOOP  NEXT                ; 循环将所有元素累加，和在(AX)
        POP   SI
        POP   CX
        POP   BP                  ; 恢复寄存器
        RET 4                     ; 返回并抛弃堆栈中的参数
PROADD  ENDP
        ;
CODE    ENDS
        END   START
```

例 4.14 展示的是一个标准的通过堆栈传递参数的方法，由例可以看出，该方法的重点有以下 4 点。

（1）调用程序在调用子程序前，将参数压入堆栈。

为了说明方便，按照入栈次序命名为第一个参数、第二个参数，以此类推。

（2）子程序开始先保存 BP（入栈），再将(SP)送到(BP)。

子程序开始处执行两条指令：

```
PUSH BP
MOV  BP, SP
```

这样 BP 就指向了 PUSH BP 后的栈顶位置。

（3）子程序按照入栈的相反次序，使用 BP 的基址寻址访问参数。

如图 4.13 所示，最后一个入栈参数使用[BP+4]访问，倒数第二个参数使用[BP+6]访问，以此类推。

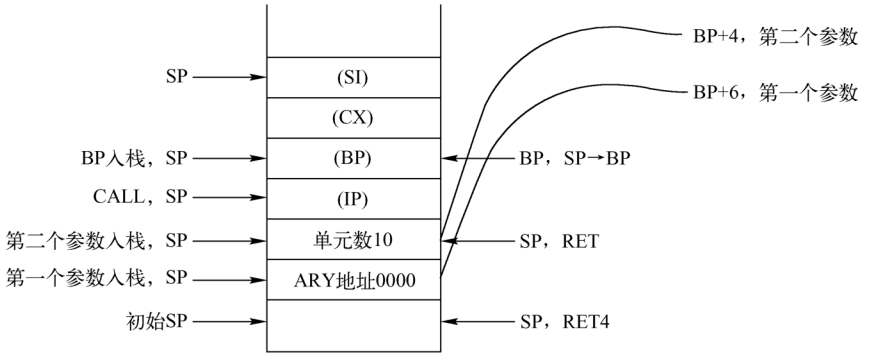


图 4.13　堆栈传递参数时堆栈的变化

使用 FAR 的调用与使用 NEAR 的调用不同，FAR 的调用将 CS 和 IP 入栈。因此，从堆栈中取参数时位移量的计算是不同的，FAR 比 NEAR 多 2，即最后一个入栈参数使用[BP+6]访问，倒数第二个参数使用[BP+8]访问等。

如图 4.14 所示是将例 4.14 中的 PROADD 子程序改为 FAR 类型的堆栈变化情况。

```
PROADD    PROC    FAR
          PUSH    BP                ; 保存寄存器
          MOV     BP, SP            ; 当前堆栈指针→BP
          PUSH    CX
          PUSH    SI                ; 保存寄存器
          ……
```

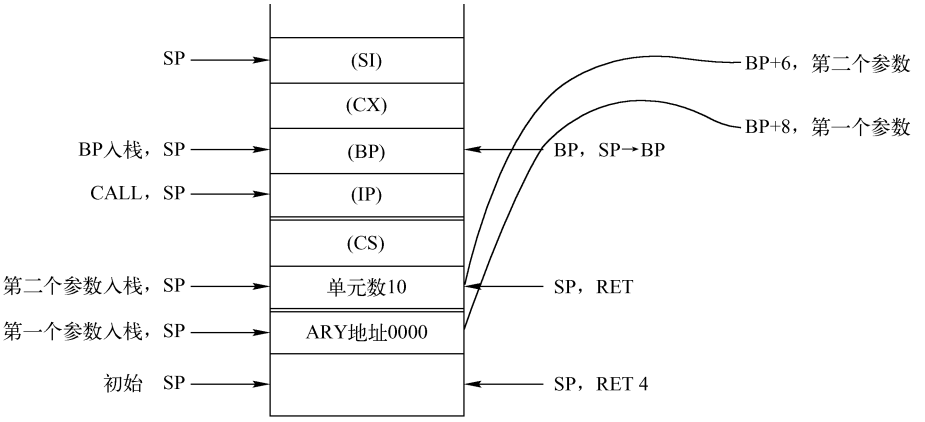


图 4.14　堆栈传递参数时堆栈的变化

（4）子程序结束时需要抛弃参数。

子程序结束时需要丢弃堆栈中已经没有用处了的参数，使堆栈复原到参数入栈前的状态。方法是使用“RET *n*”指令，它在执行 RET 操作后将 SP 加上 *n*，*n* 是参数在堆栈中所占空间（应该是偶数）。前面例子中，两个参数使用“RET 4”指令。

另外，也可以使用如下方法由调用程序负责丢弃参数，平衡堆栈。

调用者：

```
MOV    AX, OFFSET ARY
PUSH AX
PUSH COUNT
CALL PROADD
ADD    SP, 4                    ; 丢弃 2 个参数
```

子程序：

```
PROADD  PROC    NEAR
          ……
          RET                    ; 直接用 RET 返回，不丢弃参数
PROADD    ENDP
```

4.4.5　高级语言参数传递

在高级语言中，函数或过程的参数传递使用的就是堆栈传递方法。这样就实现了函数或过程中参数的局部性，在函数或过程中对参数进行修改对调用者没有任何影响。另外，函数或过程中的局部变量也是在堆栈中分配空间的。

图 4.15 所示是常见的几种高级语言的参数传递方法。

C 语言是重视效率的语言，它可根据编程模式对函数选择 NEAR 或 FAR 调用方法，而 Pascal 等语言则都处理为 FAR 调用。而且，C 语言和 Pascal 等语言的参数入栈次序是相反的，C 语言是从右往左的（其目的在于支持可变参数的函数），Pascal 等语言则是从参数表按自左至右的次序入栈。

有了堆栈传递参数方法的基础，只要遵守相应的约定就可以实现由高级语言调用汇编语言编写的子程序，反之也一样。这样就可安排系统中有的模块使用高级语言编写，有的模块使用汇编语言编写，这称为混合语言编程，有助于发挥高、低级语言各自的优势，从而获得最佳的系统效率。

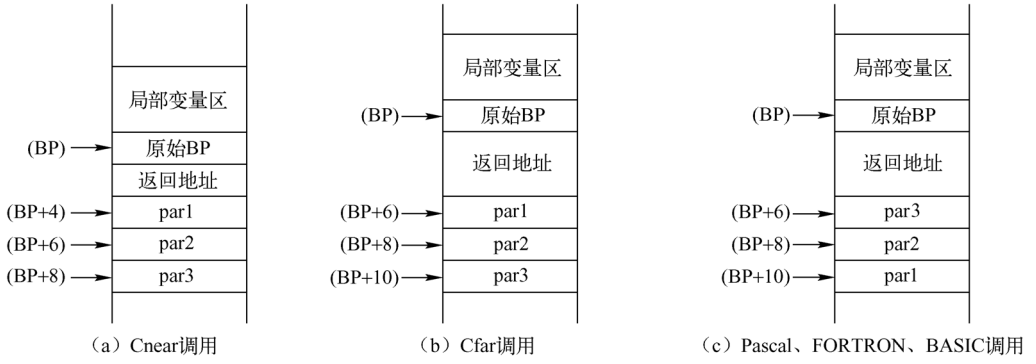


图 4.15　高级语言的参数传递方法

4.4.6　递归

在子程序中再调用子程序称为子程序的嵌套。更特殊的情况是子程序直接或间接地调用自身，这就是递归。递归的实现需要使用堆栈传递参数的方法。例 4.15 就是一个用递归实现 *n*!计算的实例。

【例 4.15】 使用递归方法计算 *n*!。

根据阶乘定义，*n*!=*n**(*n*−1)!，采用递归实现。源程序如下：

```
FRAME   STRUC
        DW ?
        DW 2 DUP (?)
N       DW ?
R_ADDR  DW ?
FRAME   ENDS                    ; 使用 FRAME 结构描述堆栈中参数的结构
DATA    SEGMENT
N_V     DW 3
RESULT  DW ?                    ; 存储 N_V 变量的阶乘
DATA    ENDS
STACK   SEGMENT STACK
        DW   100H DUP (?)       ; 堆栈空间
STACK   ENDS
CODE    SEGMENT
```

```
        ASSUME   CS:CODE, DS: DATA, SS:STACK
START:  MOV AX, DATA
        MOV DS, AX
        ; 以下是主程序
        MOV BX, OFFSET RESULT
        PUSH   BX                ; 变量 RESULT 地址入栈，第一个参数
        PUSH   N_V               ; 变量 N_V 入栈
        CALL FACT                ; 求阶乘
        ;
        MOV AX, 4C00H
        INT 21H                  ; 返回操作系统
CODE    ENDS
CODE1   SEGMENT
        ASSUME CS:CODE1
        ; FACT：求阶乘
        ; 入口：从堆栈传递结果的地址、求阶乘的数
        ; 出口：无
        ; 存储单元：通过结果变量的地址保存结果
FACT    PROC   FAR
        PUSH BP                  ; 保存 BP 寄存器
        MOV BP, SP               ; BP 指向栈顶
        PUSH DX
        PUSH BX
        PUSH AX                  ; 保存子程序要使用的所有寄存器
        MOV BX, [BP].R_ADDR      ; 结果的地址→BX
        MOV AX, [BP].N           ; N→AX
        CMP AX, 0
        JE DONE                  ; N=0 则转到 DONE, 递归出口
        PUSH BX                  ; (BX)即结果的地址入栈，第一个参数
        DEC AX                   ; N−1
        PUSH AX                  ; (AX)即 N−1 入栈
        CALL FACT                ; 求(N−1)!
        MOV BX, [BP].R_ADDR      ; 结果的地址→BX
        MOV AX, [BX]             ; 取出求(N−1)!的结果
        MUL [BP].N               ; 计算 N*(N−1)!，(AX)=N!
        JMP SHORT   RETURN       ; 转到结束
DONE:   MOV AX,1                 ; 0!=1，递归出口
RETURN: MOV [BX], AX             ; 保存结果
        POP AX
        POP BX
        POP DX
        POP BP                   ; 恢复寄存器
        RET 4                    ; 返回并丢弃参数
FACT    ENDP
CODE1   ENDS
        END START
```

递归子程序是自己调用自己。因为使用了堆栈进行参数传递，所以子程序执行所需的

上下文环境都在堆栈中。这样每次递归执行的代码虽然相同，但环境却不同，这就是递归的本质，高级语言也是这样的。

例 4.15 的堆栈变化如图 4.16 所示。在图中，用帧的形式显示每次递归子程序使用的堆栈区域。由此可以看出，递归算法虽然简单但却是以耗费堆栈空间为代价的。

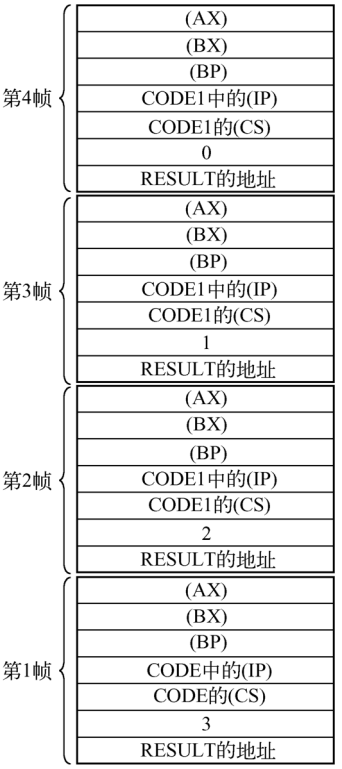


图 4.16　递归的堆栈变化

4.5　宏和条件汇编

宏是程序设计的另一个基本概念，它把一段程序代码用一个特定标识符（即，宏名）来表示。这样，在编写源程序时，程序员就可以直接使用该标识符来代替一段代码的编写，从而减少了重复代码的编写工作，也降低了错误，提高了程序的可读性和可维护性。宏在高级语言中也有广泛的使用。

4.5.1　宏

1．宏定义

在使用宏之前必须先定义宏，定义宏一般格式如下：

```
宏名    MACRO   [形参 1] [, 形参 2 , ……]
        ……                                  ;宏的定义体
        ENDM
```

在宏定义中要注意以下几点：

① MACRO 和 ENDM 是两个必须成对出现的关键字，它们分别表示宏定义的开始和结束；

② MACRO 和 ENDM 之间的部分是宏的定义体，它是由指令、伪指令或引用其他宏所组成的程序片段，是宏所包含的具体内容；

③ 在 ENDM 的前面不要再写一次宏名，这与段或子程序定义的结束方式有所不同；

④ 宏名可以与指令助记符、伪指令名相同，宏指令具有优先级。为了减少误解，应尽可能不使用指令助记符、伪指令作为宏名；

⑤ 宏可以使用形式参数，形参之间用逗号分隔。

下面例 4.16 和例 4.17 是无参数的宏和有参数的宏定义的示例。

【例 4.16】一个无参数的宏。

```
Savereg  MACRO
         PUSH   AX
         PUSH   BX
         PUSH   CX
         PUSH   DX
         PUSH   SI
         PUSH   DI
         ENDM
```

Savereg 是一个保存寄存器的宏，宏定义体中将通用寄存器全部入栈保存。

【例 4.17】 一个有参数的宏。

```
Multiply MACRO OPR1, OPR2, RESULT
         PUSH  DX
         PUSH  AX
         MOV   AX, OPR1
         IMUL  OPR2
         MOV   RESULT, AX
         POP     AX
         POP     DX
         ENDM
```

Multiply 用宏封装了乘法操作，它实现 OPR1 乘以 OPR2，乘积由 RESULT 返回。

2．宏调用

在源程序中，一旦定义了宏，就可以在该程序的任何位置通过宏名引用该宏，而不必重复编写相应的程序段，这就是宏调用。

宏调用的一般格式是：

```
宏名   [实参 1] [, 实参 2, ……]
```

其中，实参的位置与形参的位置要对应，但实参的个数可以与形参的个数不相等。当实参的个数多于形参的个数时，多出的实参被忽略；当实参的个数少于形参的个数时，没有实参对应的形参则用“空”来对应。

3．宏展开

源程序汇编时，对于宏调用，汇编程序首先把宏定义体复制到调用宏指令的位置上，同时按照参数定义顺序用实参逐一取代形参，然后再对替换后形成的程序段进行汇编处理，这一过程称为宏展开。

下面程序段在一个子程序中调用了例 4.16 定义的宏 SAVEREG 的例子。

```
SUB1    PROC   NEAR
        SAVEREG
        ......
SUB1    ENDP
```

宏展开的结果为：

```
SUB1   PROC   NEAR
       + PUSH   AX          ; 用宏定义体替换了宏调用语句
       + PUSH   BX
       + PUSH   CX
       + PUSH   DX
       + PUSH   SI
       + PUSH   DI
       ……
SUB1   ENDP
```

从宏展开原理来看，首先，宏定义只是宏的说明，宏定义体中的指令或伪指令都不会被处理，只有在宏调用展开后，宏定义体中的程序段才真正存在；其次，宏展开本质上是符号替换过程，用宏定义体替换宏调用，用实参替换形参，至于这种替换会不会产生错误的问题，交给随后的汇编过程处理。

图 4.17 所示是对例 4.17 定义的宏 Multiply 的调用和展开。

在图 4.17（a）中，宏调用指令“Multiply 240, BX, VAR”是没有问题的。在图 4.17（b）中，宏调用指令“Multiply 240, BX, VAR”展开后，实参 BX 替换形参 OPR1，240 替换形参 OPR2，VAR 替换形参 RESULT。其中，240 替换形参 OPR2 就产生了语句“IMUL 240”，这是错误的，因为乘法指令不能使用立即数寻址，随后汇编程序对展开结果进行汇编时就会发现这个错误。

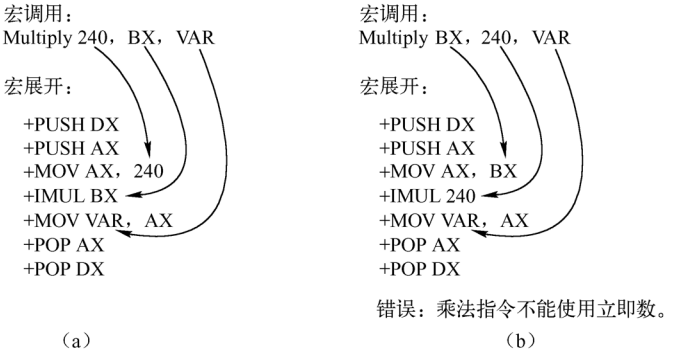


图 4.17　宏 Multiply 的调用和展开

4．宏中标号的使用

在宏中可以使用标号。例如：

```
ABSOL   MACRO   OPER
        CMP     OPER,0
        JGE     NEXT
        NEG     OPER
NEXT:
        ENDM
```

此宏定义只能被调用一次。如果在程序中多次调用该宏定义，每次展开都会出现一个标号“NEXT”的定义，这是标号重复定义错误。

宏定义中的标号需要使用伪指令 LOCAL 来说明该标号是局部标号。伪指令 LOCAL 的格式为：

```
LOCAL   标号 1 [, 标号 2, ……]
```

汇编程序在每次进行宏展开时，总是把由 LOCAL 说明的标号用一个唯一的符号（从??0000 到??FFFF）来代替，从而避免了标号重复定义的错误。

例如，上述宏 ABSOL 应改为如表 4.9 所示的实例

表 4.9　使用标号的宏示例

宏 定 义	宏 调 用	宏 展 开

ABSOL MACRO OPER
LOCAL NEXT
CMP OPER, 0
JGE NEXT
NEG OPER
NEXT:
ENDM
	ABSOL BX
	⋮
	ABSOL AL
		CMP BX, 0
		JGE NEXT
		NEG BX
		??0000:
		……
		CMP AL, 0
		JGE NEXT
		NEG AL
		??0001:
		……

5．宏的连接运算符

形参不仅可以作为操作数，还可以作为指令操作码。

如表 4.10 所示是作为指令操作码的形参示例。

表 4.10　作为指令操作码的形参示例

宏 定 义	宏 调 用	宏 展 开

FOO MACRO P1, P2
MOV P1, AX
P2 P1
ENDM
	FOO BX, INC
	⋮
	FOO VAR, DEC
		MOV BX, AX
		INC BX
		⋮
		MOV VAR, AX
		DEC VAR

在宏定义中，形参可以使用连接运算符“&”与其他字符连接在一起，替换形参的实参将与连接的其他符号合并形成一个符号。

如表 4.11 所示是使用连接符的宏示例。

表 4.11　使用连接符的宏示例

宏　定　义	宏　调　用	宏　展　开
LEAP　MACRO　COND, LAB 　　J&COND　　LAB 　　ENDM	LEAP Z, HERE 　　⋮ LEAP NZ, THERE	JZ HERE 　　⋮ JNZ THERE

6．宏嵌套

宏定义中允许调用其他宏，但需要注意必须先定义后引用。如表 4.12 所示是宏嵌套的示例。

表 4.12　宏嵌套的示例

宏　定　义	宏　调　用	宏　展　开

INT21　MACRO FUNCTION
　　MOV AH, FUNCTION
　　INT 21H
　　ENDM
DISP　MACRO CHAR
　　MOV DL, CHAR
　　INT21 02H
　　ENDM
	DISP '？'	MOV DL,'?'; 先展开 DISP
		MOV AH, 02H　；再展开 INT21
		INT 21H

宏定义中还允许嵌套宏定义。

例如：

```
DefMac MACRO MacNam, OPERATOR
        MacNam MACRO      X, Y, Z
                PUSH      AX
                MOV       AX, X
                OPERATOR  AX, Y
                MOV       Z, AX
                POP       AX
                    ENDM
        ENDM
```

内层　外层

其中，MACNAM 是外层的形参，又是内层的宏名，因此调用 DEFMAC 时就形成了一个宏定义。例如：

宏调用：DEFMAC ADDITION,ADD

宏展开：形成了 ADDITION 的宏定义

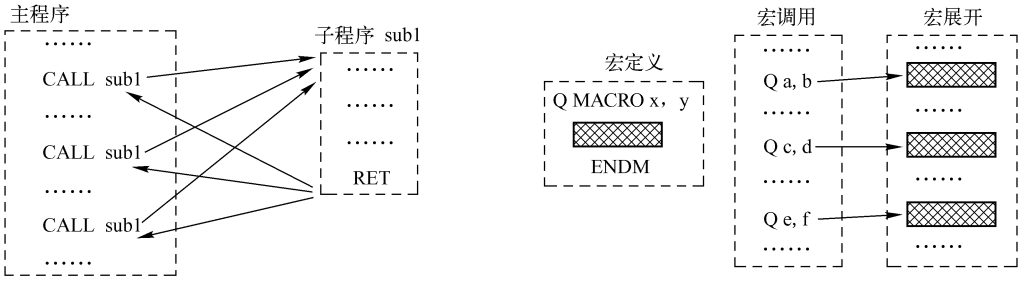
```
ADDITION MACRO X,Y, Z
        PUSH    AX
        MOV     AX, X
        ADD     AX, Y
        MOV     Z, AX
        POP     AX
        ENDM
```

这种宏定义方式虽然有一定的灵活性，但它使宏的定义更加隐蔽，削弱了程序的可读性，降低了程序的可维护性。因此，这种宏定义方式在实际的编程中很少使用。

7．宏与子程序的区别

虽然宏和子程序在编程效果上看，都是一种调用，都可以降低程序的编写量，提高程序的编程效率和可读性。但它们是有本质区别的，原因在于：子程序是 CALL 和 RET 指令的执行结果，是在程序执行过程中处理的，一般来说，子程序的多次调用比不用子程序节省代码；而宏是汇编过程中进行宏展开的效果，不会节省代码。

如图 4.18 所示是宏和子程序的对比。



（a）子程序：在程序执行期间被调用，只占用自身大小的一段空间

（b）宏调用：在汇编期间被展开，每调用一次就把宏定义体展开一次

图 4.18　宏和子程序的对比

在应用中，宏常常用来实现对指令、系统调用或子程序调用等进行功能扩充，这种扩充既保证了程序的效率，又提高了程序的编程效率及程序的可读性和可维护性。对比子程序，宏没有子程序结构相关的保存断点、恢复断点、保存环境、恢复环境等额外代价，程序的效率得到了保证。当然，对于模块功能的实现，使用宏就不合适了。

4.5.2　条件汇编

条件汇编伪指令通知汇编程序根据条件确定一组程序段是否加入到目标程序中。使用条件汇编可以使同一个源程序能根据不同的汇编条件生成不同功能的目标程序，增强了宏定义的使用范围。

条件汇编伪指令的一般格式如下：

```
IFXX 表达式
    语句组 1
[ELSE
    语句组 2]
ENDIF
```

其中，IFXX 是表 4.13 中的条件汇编伪指令。

条件汇编伪指令是在汇编程序将源程序汇编成目标程序时起作用的，其一般含义是：若条件汇编伪指令后面的表达式为真，则汇编语句组 1；否则，汇编语句组 2（如果含有 ELSE 伪指令）。

表 4.13　条件汇编伪指令及其功能

伪　指　令	含　　义
IF exp	数值表达式 exp 的值不为 0 时是真

IFE exp	数值表达式 exp 的值为 0 时是真
IFDEF label	符号 label 有定义或说明时是真
IFNDEF label	符号 label 没有定义或说明时是真
IFB <参数>	参数为空时是真，即宏引用时形参没有相对应的实参
IFNB <参数>	参数为空时不是真，即宏引用时形参有相对应的实参

例如：

（1）IF 的使用

```
BRANCH MACRO X
    IF ($-X) LT 128
        JMP SHORT X        ; 如果当前地址到目标的距离小于 128，使用 8 位相对转移
    ELSE
        JMP NEAR PTR X     ; 如果当前地址到目标的距离不小于 128，使用 16 位相对转移
ENDM
```

（2）IFDEF 的使用

宏定义

```
ADDITION  MACRO VAR1, VAR2, RESULT
    PUSH AX
    MOV AX, VAR1
    ADD AX, VAR2
IFDEF RESULT
    MOV RESULT, AX
ELSE
    MOV VAR1, AX
ENDIF
    POP   AX
        ENDM
```

宏调用

```
ADDITION DX, VAR
        ⋮
ADDITION DX, VAR, CX
```

宏展开

```
PUSH AX
MOV AX, DX
ADD AX, VAR
MOV DX, AX
POP   AX
……
PUSH AX
MOV AX, DX
ADD AX, VAR
MOV CX, AX
POP   AX
```

调用 ADDITION 宏时，如果只给两个实参就执行 VAR1=VAR1+ VAR2，如果给三个实参就执行 RESULT=VAR1+ VAR2。

（3）IFB 的使用

宏定义

```
GOTO  MACRO L, X, REL, Y
IFB <REL>
    JMP  L
ELSE
    MOV  AX, X
    CMP  AX, Y
    J&REL     L
ENDIF
    ENDM
```

宏调用

```
GOTO LOOP1, SUM, NZ, 15
        ⋮
GOTO EXIT
```

宏展开

```
MOV AX, SUM
CMP AX, 15
JNZ LOOP1
    ⋮
JMP EXIT
```

习题 4

1．简述汇编语言的优缺点及应用场合。

2．什么是汇编过程？其中，汇编程序起什么作用？

3．伪指令 END 有何作用？

4．画图说明下列语句所分配的存储空间及初始化数据情况。

（1）B_VAR DB 'ABCD', 12, −12, 3 DUP (?, 0, 2 DUP (1, 2))

（2）W_VAR DW 'AB', 256, −1, 5 DUP (?, 1, 2)

5．变量名、标号的属性有哪些？

6．有数据段定义如下：

```
DATA SEGMENT
    V1   DW ?
    V2   DB 16 DUP (?)
    V3   DD ?
    DSIZE EQU $−V2
DATA ENDS
```

变量 V1、V2、V3 的地址是多少？DSIZE 的值是多少？

7．试在数据段内定义字节变量 B1 和字变量 W1，它们共享 20 个存储单元。

8．在定义一个字变量时，如果想保证其地址是偶地址，应如何实现？这样做有何意义？

9．指令 MOV [BX], 1 是否正确？为什么？如果有错误，应如何修改？

10．简述指令 MOV BX, VAR 和 MOV BX, OFFSET VAR 之间的区别（VAR 是已定义的字变量）。

11．已知数据段有变量定义如下，试写出符合要求的语句或序列。

```
VAR DW  ‘AB’, 256, −1, 5 DUP (?, 1, 2)
```

（1）取 VAR 的段址到 AX 中；

（2）取 VAR 中−1 到 AX 中；

（3）取 VAR 中−1 的地址到 BX 中；

（4）取 VAR 第 4 个字节内容到 AL 中；

12．判断下面各语句的正确性（其中 VAR 为数据段中定义的字变量）。

（1）XOR BX, [SI][DI]

（2）SUB SI, −91

（3）OUT 300H, AL

（4）JMP DWORD PTR BX

（5）MOV BX, SS:[SP]

（6）TEST AL, 8000H

（7）CMP BYTE PTR [BX], BYTE PTR [SI]

（8）ANDVAR, OFFSET VAR

（9）SUB [SI], 1

（10）ADD BX, SS:VAR

（11）MOV AX, [BX−SI]

（12）MOV 25, BL

（13）SUB CX, $+10

（14）MOV DS, BP

（15）MOV AX, BX + VAR

13．编写一个程序，统计在双字变量 DDVAR 的内容中二进制位是 1 的位数，并存入变量 COUNT 中。

14．假设有三个字变量 a、b 和 c，编写一个程序，它可判断它们能否构成一个三角形，若能则 CF 为 1，否则 CF 为 0。

15．编写一个程序把字符串 String 两端的空格删除（字符串以 0 结束）。

16．假设从变量 Buff 开始存放了 200 个字，编写一个程序统计出其正数、0 和负数的个数，并把它们分别存入 N1、N2 和 N3 中。

17．用双重循环将下三角乘法表存入从 Result 开始的 45 字节中。

18．简述在子程序结构中堆栈的作用。

19．如何区分 RET 指令是段内返回还是段间返回？

20．在子程序中如果要使其所用寄存器对调用者是透明的，试举例说明达到其目的的方法。

21．子程序参数传递主要有哪些方法？它们各有什么优缺点？

22．分别编写子程序实现下列功能（所有变量都是字类型）。

（1）ABS(x)=|x|

（2）F(x)=$3x^2 + 5x - 8$

（3）strlen(String)，（求字符串长度，字符串以 0 结束）

23．给定一个正数 $n \geqslant 1$ 存放在 NUM 字变量，试编写递归子程序计算 FIB(n)，结果保存到 RESULT 变量中。

Fibonacci 数定义如下：

$$\begin{cases} \mathrm{FIB}(1) = 1 \\ \mathrm{FIB}(2) = 1 \\ \mathrm{FIB}(n) = \mathrm{FIB}(n-2) + \mathrm{FIB}(n-1) \end{cases}$$

24．C 语言是如何实现可变参数的函数的？举例说明。

25．宏和子程序的主要区别有哪些？一般在什么情况下选用宏较好，在什么情况下选用子程序较好？

26．在宏调用时，是否要求实参与形参的个数相等？若不要求，当个数不一致时会出现什么情况？

27．宏应如何正确使用标号？

28．定义宏指令 FINSUM：比较两个数 X 和 Y，若 $X>Y$，则执行 SUM=$X + 2 \cdot Y$，否则，执行 SUM=$2 \cdot X + Y$。

29．使用条件汇编的目的是什么？

30．编写只有一个形式参数的宏 PRINT，其具体功能如下：

（1）若引用时带有参数，则在屏幕上显示其参数字符，例如，若 PRINT 'A'，则显示字符 A；

（2）若引用时不带实参，则显示回车和换行，例如，PRINT。

31．一个学生的信息包括姓名、班级、学号、成绩，其中成绩需要精确到 1 位小数。

试编写程序实现以下功能：

（1）可以录入学生成绩（十进制形式）；

（2）可以按要求（如学号或成绩）进行排序显示；

（3）可以统计平均成绩；

（4）可以统计不及格、60～70、70～80、80～90、90～100 各分数段的人数。

提示：应该从上至下规划程序结构，划分各个子程序的功能和调用关系。