第3章 汇编语言基础[①]

在中国，随着信息技术和高性能计算的迅速发展，汇编语言在系统底层设计、性能优化以及安全关键应用领域中发挥着越来越重要的作用。预计未来汇编语言将继续在支持新的处理器架构、优化大数据处理和人工智能计算中扮演关键角色。本章将深入介绍汇编语言的基础知识，从最基本的元素、数据类型、指令使用到复杂的内存管理和程序结构设计。通过详细讲解整数和实数的表示、字符与字符串的处理，以及条件控制和循环结构的实现，读者将获得对汇编语言强大功能的理解和应用能力。本章不仅是对汇编语言编程的入门引导，也是对底层计算机操作和优化技术的一次深入探索。通过本章的学习，读者将能够掌握汇编语言的核心概念，为进一步的学习和实践打下坚实的基础。

3.1 汇编语言的基本元素

本节将介绍汇编语言的基本元素，通过逐步熟悉各个元素的定义和使用，读者将初步学会编写可运行、有意义的汇编语言应用程序。

3.1.1 整数常量

整数常量通常由符号(sign)、数字(digit)和表示数制基数的字符后缀(radix)组成。符号是一个可选的部分，数字部分允许由一个或多个数字组成，其基本构成如下：

```
[{ + | - }] 数字 [基数]
Radix(基数后缀)不区分大小写,包括以下类型。
h    十六进制                r    编码实数
q/o  八进制                  t    十进制(可选)
d    十进制                  y    二进制(可选)
b    二进制
```

如果整数常量后面没有后缀，就默认其为十进制数。这里给出一些使用不同基数后缀的例子：

① 本章全部采用微软的语法格式符号，在[..]内的参数是可选参数，在{..}内的参数要求必须从被竖线分隔的多个参数中选择一个，斜体参数代表已知项目的定义或描述。

```
26          十进制数        42o         八进制数
26d         十进制数        1Ah         十六进制数
11010011b   二进制数        0A3h        十六进制数
42q         八进制数
```

注意,以字母开头的十六进制常量前面必须加一个 0(如例子中的 0A3h),以防止汇编器将其解释为标识符。

3.1.2 整数表达式

整数表达式是由整数值和算术运算符共同组成的数学表达式。整数表达式的计算结果是能够存于 32 位数据位中的整数(0～FFFFFFFFh)。表 3-1 按照算术运算符的运算优先级(从高到低)列出了符号和名称。

表 3-1 算术运算符

运 算 符	名 称	优 先 级
()	圆括号	1
+,-	一元加、减	2
* ,/	乘、除	3
MOD	取余数	3
＋,－	加、减	4

优先级指的是如果表达式中同时包含两个或两个以上操作符时,操作符间隐含的运算顺序。以下的例子展示了多个算术运算符之间的运算顺序:

```
6 + 1 - 2              加,减
12 - 8 * 6             乘,减
-9 / 3                 一元减,除
(11 + 4) * 8           加,乘
```

在表 3-2 中,给出了一些更复杂的有效表达式和它们的计算值。

表 3-2 有效的整数表达式和计算值

表 达 式	值
7 mod 2	1
－(11－5)＊(3＋2)	－30
28/(2＊3＋1)	4
46－6/2＊3	37

对于初学者而言,可以尽量在表达式中使用圆括号来显式地表明运算顺序,这样就不必记优先级规则了。

3.1.3 实数常量

实数常量分为两种类型,包括十进制实数和编码(即十六进制)实数。十进制实数常量

通常由符号(sign)、整数(integer)、小数点、表示小数的整数和指数(exponent)五部分组成，其构成如下：

```
[sign]integer.[integer][exponent]
```

其中的符号、指数的格式如下：

```
sign                    {+,-}
exponent                E[{+,-}]integer
```

这里我们给出一些有效的实数常量的例子：

```
6.
78E+04
-24.5
91.E5
```

实数常量最少应该包括一个数字和一个小数点。如果只有一个数字，那么它就是一个整数常量。

编码实数：所谓编码实数是以十六进制数表示一个实数的实数常量，它们遵循 IEEE 浮点数格式。十进制实数＋1.0 的二进制数表示如下：

0011 1111 1000 0000 0000 0000 0000 0000

在汇编语言中，＋1.0 将被编码为单精度实数：3F800000r。

3.1.4　字符常量

字符常量指的是以单引号或双引号括起来的单个字符。汇编器会将其转换成与字符对应的二进制数形式的 ASCII 码，例如：

```
'G'
"w"
```

3.1.5　字符串常量

字符串常量和字符常量类似，是以单引号或双引号括起来的一串字符。例如：

```
'ZHM'
'P'
"Cute Puppy"
'34A5'
```

使用嵌套的引号也是可行的。例如：

```
"This is a difficult "test""
'Order Something, "Geroge"'
```

3.1.6　保留字

MASM 中设置了一些具有特殊含义的保留字，这些保留字只适用于正确的上下文环境，以下是一些不同类型的保留字：

- 指令助记符，如 ADD，SUB，MOV 等。
- 伪指令，告诉 MASM 应该如何编译程序。
- 属性，为变量和操作数提供有关数据尺寸、使用方式的信息，如 BYTE、DWORD 等。
- 运算符，常用于常量表达式中。
- 预定义符号，如@data，在编译时返回整数常量值。

3.1.7 标识符

标识符是程序员设置的名字，它们通常用于标识变量、常量、过程或代码标号。标识符的创建需要符合以下规则：

- 包含 1～247 个字符。
- MASM 默认大小写不敏感。
- 第一个字符必须是字母（A～Z、a～z）、下画线（_）、@、? 或 $，后续的字符可以使用数字。
- 标识符的名字不能与汇编器的保留字相同。

在运行汇编器时，如果在命令行上使用-Cp 选项控制命令，可以要求所有关键字和标识符对大小写敏感。

@符号被汇编器大量用作预定义符号的前缀，因此，程序员在开发时应尽量避免自己定义的标识符使用@符号作为首字符，尽管它也符合要求。尽量使标识符的名字具有描述性且便于理解，这有助于提高开发的效率和正确率。以下是一些有效的标识符：

```
uixm            Number          @first
MIN             files_open      _run
fsWp            _w156           $wrong
```

3.1.8 伪指令

伪指令内嵌在程序源代码中，它由汇编器识别并执行相应动作。与真正的指令不同，伪指令不在程序运行时执行。伪指令可用于定义变量、宏、过程，也可用于命名段，或执行许多其他与汇编器有关的编译任务。MASM 中伪指令对大小写不敏感，如.code、.CODE 和 .Code 是等价的。

这里给出一个例子，说明伪指令不会在程序运行时执行。DWORD 伪指令告知汇编器，要在程序中划出一片双字变量保留空间。直到 MOV 指令在运行时，才真正执行，把 testVal 的内容复制到 EAX 寄存器：

```
testVal    DWORD 26              ; DWORD 伪指令
mov        eax,testVal           ; MOV 指令
```

汇编器都有自己的一套独特伪指令。例如，TASM（Borland）、NASM、MASM 的伪指令之间存在一个公共的交集子集，而 GNU 与 MASM 的伪指令则几乎没有相同之处。

定义段：在汇编语言的伪指令中，有一个重要的功能是定义程序的节（section）或者段（segment）。

.DATA 伪指令标识程序中包含变量的区域：

```
.data
```

.CODE 伪指令标识程序中包含指令的区域：

```
.code
```

.STACK 伪指令标识程序中包含运行时栈的区域，并且需要设定运行时栈的大小：

```
.stack 100h
```

3.1.9　指令

在汇编语言中，指令指的是一条汇编语句，它经过汇编后就变成了可执行的机器指令。汇编器会将汇编指令翻译成机器语言字节码，在程序运行时将其加载到内存交由处理器执行。

一条汇编指令包含以下四部分：

- 标号(可选)。
- 指令助记符(必需)。
- 操作数(通常必需)。
- 注释(可选)。

其基本的格式如下：

[标号:]	指令助记符	操作数	[;注释]

接下来我们分别了解其中的每个部分。

1. 标号

首先是标号域，这是一个可选的域。标号指的是充当指令或者数据位置标记的一种标识符。在指令前的标号指明了指令的地址，类似地，在变量前的标号则指明了变量的地址。

(1) **数据标号**：标识变量地址的标号被称为数据标号，它为在代码中引用变量提供了便利。这里我们定义了一个名为 count 的变量：

```
count DWORD 100
```

汇编器为每个标号都分配了一个数字地址。一个标号后可以定义多个数据项。这里的 array 标识了第一个数字 1024 的位置，而其他相邻的数字在内存中紧随其后：

```
array DWORD 1024, 2048
DWORD 4096, 8192
```

(2) **代码标号**：在代码区域(存放指令的部分)中的标号必须以冒号(：)结尾，它们被称为代码标号。代码标号通常会被用作跳转和循环指令的目标地址。这里的 JMP(跳转)指令会将控制权转换到标号 loop 的位置，从而实现一个循环：

```
loop:
    mov ax, bx
    …
    jmp    loop
```

代码标号既可以独自成行，如上面的 loop，也可以和指令在同一行：

```
L1: mov    ax, bx
L2:
```

数据标号则不能以冒号结尾，标号名应当遵循 3.1.7 节中讨论过的标识符名创建规则。

2. 指令助记符

指令助记符(instruction mnemonic)通常是一个非常简短的单词，用于标识一条指令。所谓助记符，就是辅助开发者记忆指令作用的符号。在汇编语言中，指令助记符给出了关于指令要执行何种类型操作的提示：

```
mov     将一个值移动(赋值)到另一个值中
add     两个值相加
sub     从一个值中减去另一个值
mul     两个值相乘
jmp     将程序控制跳转到一个新的位置上
call    调用一个过程
```

3. 操作数

一条汇编语言指令可以有 0～3 个操作数，每个操作数可以是寄存器、内存操作数、常量表达式或 I/O 端口。在第 2 章中，我们已经讨论过寄存器的名字；在 3.1.2 节中，我们讨论了常量表达式。

内存操作数由变量或包含变量地址的一个或多个寄存器指定。变量名指明了变量的地址，并且会指示计算机引用给定内存地址的内容。以下是一些操作数的例子：

```
34       常量(立即值)
9 * 7    常量表达式
eax      寄存器
count    内存
```

接下来再给出一些带有不同数量操作数的汇编语言指令的例子：

```
STC 指令没有操作数:
stc                 ;设置进位标志
INC 指令有一个操作数:
inc eax             ;eax 加 1
MOV 指令有两个操作数:
mov count,ebx       ;ebx 赋值给变量 count
```

在类似于 MOV 这样具有两个操作数的指令中，第一个操作数被称为目的(标)操作数，第二个操作数则被称为源操作数。指令一般用于修改目的操作数的内容。如上例中，MOV 指令将 ebx(源操作数)的数据复制到 count(目的操作数)中。

4. 注释

注释用于程序开发者和阅读者之间的交流，它说明了程序如何工作。在程序清单的顶部通常会包含如下一些典型信息：

- 程序创建者/修改者的名字。

- 程序创建/修改的日期。
- 程序功能的描述。
- 程序实现的技术注解。

在汇编器中,注释可以用如下 2 种方法实现。

(1) **单行注释**:在一行指令的分号(;)之后,汇编将会忽略同一行分号后的所有字符。

(2) **块注释**:以 COMMENT 伪指令,加上一个用户定义的符号开始。汇编器会省略后面所有的文本段,直到读取到另一个相同的用户定义符号。假设用户定义的符号是感叹号(!):

```
COMMENT !
    ! This is example for comment.
!
```

当然,也可以使用其他符号:

```
COMMENT &
    & is also a user-defined symbol.
&
```

3.1.10　NOP(空操作)指令

NOP 指令是最安全的指令,一条 NOP 指令仅占用一字节的存储,且什么事情都不做。编译器或汇编器会使用 NOP 指令将代码对齐到偶数地址的边界上。在以下指令段,第一个 MOV 指令生成了 3 个机器字节码,而 NOP 指令则将第二条 MOV 指令的地址对齐到双字(4 的倍数)边界上。

```
00000000    66 8B    C3 mov ax,bx
00000003    90       nop              ;用于下一条指令的对齐
00000004    8B D1    mov edx,ecx
```

使用偶数双字地址的原因是:对于一些处理器,如 IA-32,它们在从偶数双字地址处加载代码和数据时会更加迅速。

本节习题

(1) 解释汇编语言中的“操作码”是什么,并举例说明。

(2) 描述“操作数”在汇编指令中的作用,并给出两个示例。

(3) 列举三种基本的汇编指令类型。

(4) 解释什么是“伪指令”以及它们在汇编语言中的作用。

(5) 比较“直接寻址”和“间接寻址”的区别。

(6) 描述“基址寻址”和“变址寻址”各自的特点。

(7) 解释“堆栈”在汇编语言中的作用,并举例说明。

(8) 解释汇编语言中的“标签”作用,并给出使用场景。

(9) 描述寄存器在汇编语言中的作用,并列举至少三种不同类型的寄存器。

(10) 解释“汇编指令”与“机器指令”的区别。

(11) 说明“数据定义指令”的作用,并举例说明如何使用。
(12) 列出两种常见的汇编语言程序结构,并简述其特点。
(13) 描述“宏指令”在汇编语言中的应用。
(14) 解释在汇编语言中如何实现程序的循环控制结构。
(15) 描述汇编语言中条件跳转指令的作用,并给出一个例子。

3.2 例子:整数相加减

我们来尝试编写一个进行整数加减操作的汇编语言程序。寄存器将用于存放中间数据,我们可以调用一个库函数在屏幕上显示寄存器的内容。

以下给出程序的源码:

```
TITLE Add and Subtract                (AddSub.asm)
; This program adds and subtracts 32-bit integers.
INCLUDE Irvine32.inc
.code
main     PROC
    mov eax,10000h                    ; EAX = 10000h
    add eax,40000h                    ; EAX = 50000h
    sub eax,20000h                    ; EAX = 30000h
    call    DumpRegs                  ; display  registers
    exit
main ENDP
END main
```

下面来逐行解释代码:

```
TITLE Add and Subtract                (AddSub.asm)
```

TITLE 伪指令将该行标为注释,因此该行可以填写任意内容。

```
; This program adds and subtracts 32-bit integers.
```

编译器会自动忽略分号右边的所有文本,因此这段内容同样为注释。

```
INCLUDE Irvine32.inc
```

INCLUDE 伪指令将从 Irvine32.inc 文件中复制得到必需的定义和设置信息,Irvine32.inc 文件在汇编器的 INCLUDE 目录中。

```
.code
```

.code 伪指令标记了代码段的开始位置,在代码段中存放程序所有的可执行语句。

```
main     PROC
```

PROC 伪指令标记了过程的开始,在这段程序中只有一段过程,其名字为 main。

```
mov eax,10000h                        ; EAX = 10000h
```

MOV 指令会将整数 10000h 复制到 EAX 寄存器。MOV 指令的第一个操作数(EAX)被称为目的操作数,第二个操作数则被称为源操作数。

```
add eax,40000h                          ; EAX = 50000h
```

ADD 指令会将 40000h 加到 EAX 寄存器上。

```
sub eax,20000h                          ; EAX = 30000h
```

SUB 则会从 EAX 寄存器中减掉 20000h。

```
call    DumpRegs                        ; display  registers
```

CALL 指令调用了一个显示 CPU 寄存器值的过程,这通常用于证明程序的正常运行。

```
exit
main ENDP
```

exit 语句间接调用了一个预定义的 MS-Windows 函数来终止程序。ENDP 伪指令则标记了 main 过程的结束。exit 并不是 MASM 定义的关键字,而是 Irvine32.inc 中定义的命令,它提供了一种结束程序的简便方法。

```
END main
```

END 伪指令标明了该行是汇编源程序的最后一行,编译器会忽略掉该行后面的所有内容。这之后的标识符 main 是程序启动过程,即程序启动时要执行的子程序/程序入口点的名字。

段:汇编语言中的程序是以段为单位组织的,常见的段有代码段、数据段和堆栈段等。代码段包含了程序的全部可执行指令,通常代码段中会包含一个或多个过程,其中一个是启动过程。在上述的 AddSub 程序中,main 过程即为启动过程。数据段用于存放变量,而堆栈段则用于存放过程运行中的参数和局部变量。

程序的输出:下面就是程序的输出值,这是通过调用 DumpRegs 子程序产生的。

```
EAX=00030000    EBX=7FFDF000    ECX=00000101            EDF=FFFFFFFF
ESI=00000000    EDI=00000000    EBP=0012FFF0            ESP=0012FFC4
EIP=00401024    EFL=00000206    CF=0  SF=0  ZF=0  OF=0  AF=0    PF=1
```

输出的前两行是 32 位通用寄存器的十六进制数值。EAX 最终值为 00030000h,该值是通过程序中的 ADD 和 SUB 指令产生的。第三行则显示了 EIP(扩展指令指针)、EFL(扩展标志)寄存器,以及进位、符号、零、溢出、辅助进位、奇偶标志的值。

3.2.1　AddSub 程序的另一个版本

在前面的 AddSub 程序中,我们使用了 Irvine32.inc 文件,这个文件包含了一些能够直接使用的指令,也因此隐藏了一些细节。通过记忆,开发者们可以掌握 Irvine32.inc 的使用方式,但在学习的初期,我们可以简单了解其中的内容。以下是一个不依赖任何包含文件版本的 AddSub 程序,粗体字标识出和前一个版本不同的部分:

```
TITLE Add and Subtract                          (AddSubAlt.asm)
```

```
; This program adds and subtracts 32-bit integers.
.386
.model flat,stdcall
.stack 4096
ExitProcess PROTO,  dwExitCode:DWORD
DumpRegs    PROTO
    .code
    main     PROC
        mov eax,10000h                  ; EAX = 10000h
        add eax,40000h                  ; EAX = 50000h
        sub eax,20000h                  ; EAX = 30000h
        call    DumpRegs                ; display    registers
        INVOKE  ExitProcess,0
    main ENDP
    END main
```

下面讨论与之前版本不同的代码行：

```
.386
```

.386 指出了程序中要求的 CPU 最低版本，即 Intel 386。

```
.model flat,stdcall
```

.MODEL 伪指令控制汇编器为保护模式程序生成代码，而 STDCALL 允许程序调用 MS-Windows 函数。

```
ExitProcess PROTO,  dwExitCode:DWORD
DumpRegs    PROTO
```

这两条 PROTO 伪指令声明了前一版本程序调用的过程的原型：ExitProcess 是一个 MS-Windows 函数，它的作用就是终止当前程序(即当前进程)；DumpRegs 是 Irvine32 链接库中用于显示寄存器的过程。

```
INVOKE  ExitProcess,0
```

程序通过调用 ExitProcess 来结束执行，此时传递给函数的参数是返回码，值为 0。INVOKE 本身是一个用于调用过程和函数的汇编伪指令。

3.2.2 程序模板

汇编语言程序和其他语言一样，有一个简单的基本框架结构。随着情况的不同，开发者可以自由地改变这一框架。在开始学习编写汇编语言程序时，开发者们可以借助基本框架快速创建具有所有元素的空程序，然后填写其中缺少的部分即可。以下我们给出一个保护模式模板(Template.asm)，便于初学者根据需要进行自定义。

```
TITLE    Program Template                    (Template.asm)
; 程序的描述:
; 作者:
; 创建日期:
; 修改:
```

```
; 日期:          修改者:
INCLUDE Irvine32.inc
.data
    ; (在此处键入变量)
.code
main PROC
    ;(在此处插入可执行代码)
    exit
main ENDP
    ; (在此插入其他的子程序)
END main
```

在程序的开始位置插入了几个注释区域。这里包含了程序的描述、作者的名字、创建日期以及后续修改信息，便于开发者和读者快速了解程序。

本节习题

(1) 编写一个汇编语言程序，实现两个 32 位整数的加法操作。
(2) 在整数加法汇编程序中，寄存器如何被用来存储和操作数据？
(3) 编写一个汇编程序段，使用 SUB 指令从一个寄存器值中减去一个立即数。
(4) 解释汇编语言中 ADD 和 SUB 指令的基本语法格式。
(5) 如何使用寄存器间接寻址方式实现两个内存中整数的加法操作？
(6) 在整数加减汇编程序中，如何显示寄存器的当前值？
(7) 编写一个汇编程序，先将两个整数相加，然后从结果中减去第三个整数。

3.3 汇编、链接和运行程序

前面几章，我们介绍了简单的机器语言程序。但是，汇编语言所编写的程序是不能像机器语言一样直接在目标机上运行的，必须被汇编成可执行代码。从效果上来看，汇编器和其他编译器(如 C++ 、Java 的编译器)十分相似。

汇编器会生成一个包含机器语言的文件，称为目标文件。目标文件需要被传递给另一个称为链接器的程序，再由链接器生成可执行文件。这样得到的可执行文件，就可以利用 MS-DOS/MS-Windows 命令提示符来执行了。

1. 总体流程

开发者编辑、编译、链接、执行汇编语言程序的总体流程如图 3-1 所示。

步骤 1：开发者使用编辑器创建 ASCII 文本文件，即源文件(source file)。

步骤 2：汇编器读取源文件并生成目标文件(object file)，即源文件到机器语言的翻译文件。此外，还可以选择生成列表文件(listing file)。如果发生错误，开发者必须回到步骤 1 修改程序。

步骤 3：链接器会读取目标文件，并检查程序是否调用链接库中的过程。链接器将从库中复制需要的过程，并将其与目标文件合并在一起，生成可执行文件(executable file)或映像文件(map file)。

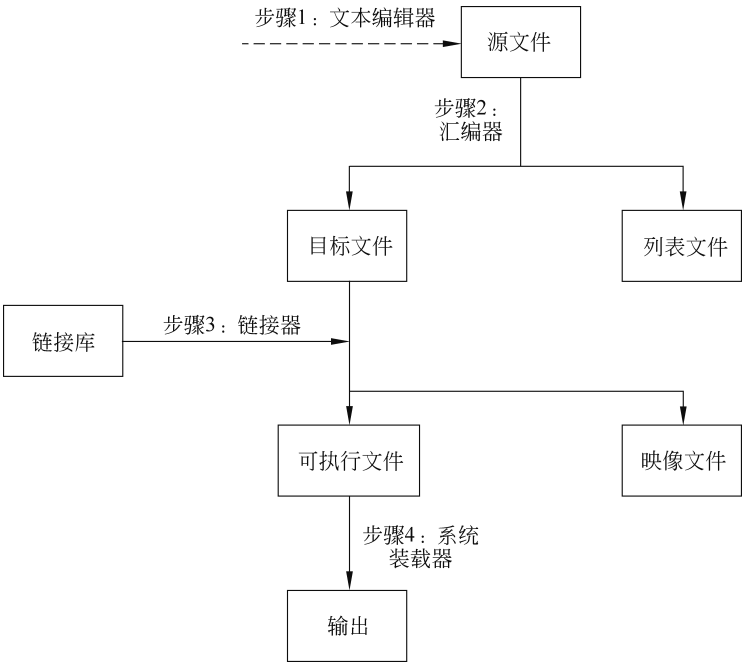

图 3-1　汇编语言执行的总体流程

步骤 4：操作系统的装载器(loader)会将可执行文件读入内存，并驱使 CPU 转移到程序的起始地址开始执行程序。

2. 链接器创建/更新的文件

(1) **程序数据库文件**：如果用调试(-Zi)选项来编译程序，MASM 就会创建程序数据库文件，其扩展名为 PDB。在链接阶段，链接器会读取并更新它。在调试程序时，调试器可以利用 PDB 文件来显示程序的源代码、数据、运行时栈和其他附加信息。

(2) **映像文件**：所谓映像文件是指包含了被链接程序中分段信息的文本文件，映像文件主要包括以下 5 种信息。

- 程序文件头中的时间戳。
- 程序入口地址。
- 模块名，即链接器生成的可执行文件中除扩展名外的部分的基本名。
- 程序中各个段组的列表，其中包含每个段组的起始地址、长度、组名以及类别信息。
- 公共符号的列表，其中包括每个符号的地址、名称、线性地址以及定义符号的模块。

(3) **列表文件**：列表文件包含程序源代码、行号、偏移地址、经过翻译的机器码以及一个符号表，格式适于打印。以下就是我们在 3.2 节创建的 AddSub 程序的列表文件：

```
Microsoft (R) Macro Assembler Version 8.00
Add and Subtract (AddSub.asm)                          Page 1 - 1
TITLE Add and Subtract                                (AddSub.asm)
; This program adds and subtracts 32-bit integers.
INCLUDE Irvine32.inc
C    ; Include file for Irvine32.lib (Irvine32.inc)
C    INCLUDE SmallWin.inc
```

```
00000000    .code
00000000    main PROC
00000000    B8  00010000        mov eax,10000h    ; EAX = 10000h
00000005    05  00040000        add eax,40000h    ; EAX = 50000h
0000000A    2D  00020000        sub eax,20000h    ; EAX = 30000h
0000000F    E8  00000000E        call DumpRegs
           exit
0000001B    main    ENDP
           END main
Structures and Unions: (omitted)
Segments and Groups:
Name        Size        Length        Align    Combine    Class
FLAT……..Group
STACK……...32    Bit 00001000    DWord    Stack    'STACK'
_DATA……32     Bit 00000000    DWord    Public   'DATA'
_TEXT……32     Bit 0000001B    DWord    Public   'CODE'
Procedures,parameters and locals (list abbreviated):
Name        Type        Value    Attr
CloseHandle…P Near 00000000 FLAT    Length=00000000    External    STDCALL
ClrScr……..P Near 00000000 FLAT    Length=00000000    External    STDCALL
.
.
main……..P Near 00000000 _Text    Length=0000001B    Public    STDCALL
Symbols    (list    abbreviated):
Name                                Type        Value        Attr
@CodeSize……..Number    00000000h
@DataSize……..……..Number    00000000h
@InterSize……..……..Number   00000003h
@Model…..……..Number    00000007h
@code……..Text                    _TEXT
@data.……..Text                    FLAT
@fardata?……...……..Text                    FLAT
@fardata……..……..Text                    FLAT
@stack……..Text                    FLAT
.
.
exit……...Text INVOKE  ExitProcess,0
    0 Warnings
    0 Errors
```

本节习题

(1) 描述汇编器的功能,以及它是如何将汇编语言代码转换成机器语言代码的。

(2) 什么是链接器? 以及为什么在汇编语言程序开发中需要链接器?

(3) 解释汇编语言程序的运行过程,并讨论与高级语言程序运行的区别。

(4) 在创建汇编语言程序时,如何处理和解决模块间的引用问题?

(5) 描述汇编语言程序调试的一般步骤和常见问题。

3.4 定义数据

在本节中，我们将探讨汇编语言在定义和处理数据方面的广泛应用，包括不同数据类型的定义和优化存储方案。展望未来，汇编语言在支持中国科技创新和高性能计算领域中扮演着至关重要的角色。随着物联网、智能硬件及人工智能技术的不断发展，对于能直接操控硬件并最大限度提高运算效率的编程语言的需求日益增加。汇编语言的这些特性使得它在开发需要极致性能优化的系统时，成为了不可或缺的工具。因此，深入了解如何使用汇编语言精确地定义和操控数据，将直接影响到未来技术解决方案的效率和创新速度。

3.4.1 内部数据类型

MASM 中定义了多种内部数据类型，这些数据类型描述了变量、表达式的取值集合。数据类型以数据位的数目度量的大小为基本特征：8，16，32，48，64，80 位。数据类型的符号、指针、浮点等其他特征是为了方便开发者记忆所提供的。例如，DWORD 变量在逻辑上存储的是一个 32 位无符号整数，但是实际上也可以存放一个有符号位的 32 位整数、32 位指针或 32 位浮点数。

在编程时，MASM 汇编器本身对于大小写是不敏感的，如伪指令 BYTE 可写作 byte、Byte、bYte 等大小写混合的格式。

在表 3-3 中，除了最后三种 REAL 类型之外，其余所有的数据类型都是整数数据类型。表格中 IEEE 符号指的是由 IEEE 委员会发布的标准实数格式。

表 3-3 内部数据类型

类　型	用　途
BYTE	8 位无符号整数
SBYTE	8 位有符号整数
WORD	16 位无符号整数(在实地址模式下可为近指针)
SWORD	16 位有符号整数
DWORD	32 位无符号整数(在保护模式下可为近指针)
SDWORD	32 位有符号整数
FWORD	48 位整数
QWORD	64 位整数
TBYTE	80 位(10 字节)整数
REAL4	32 位(4 字节)IEEE 短实数
REAL8	64(8 字节)IEEE 长实数
REAL10	80 位(10 字节)IEEE 扩展精度实数

3.4.2　数据定义语句

数据定义语句会在存储空间中为变量保留对应的位置，并且给变量一个指定的名字。数据定义语句会创建表 3-3 中汇编器内部数据类型对应的变量，其定义格式如下。

```
[变量名] 数据定义伪指令 初始值[,初始值]…
```

(1) **变量名**：数据定义语句中变量的名字是可选择的，必须遵循标识符名的创建规则(3.1.7 节)。

(2) **数据定义伪指令**：可以是 BYTE、WORD、DWORD、SBYTE、SDWORD 或表 3-3 列出的任何其他类型。数据定义伪指令也可以是表 3-4 中历史遗留的数据定义伪指令，其他汇编器(如 TASM、NASM)也支持这些伪指令。

表 3-4　历史遗留的数据定义伪指令

伪　指　令	用　　途
DB	定义 8 位整数
DW	定义 16 位整数
DD	定义 32 位整数或实数
DQ	定义 64 位整数或实数
DT	定义 10 字节(80 位)

(3) **初始值**：在数据定义语句中，初始值必须要有一个指定值，即使其为 0。如果有多个初始值，应该以逗号分隔。对于整数数据类型，其初始值可以是与变量的数据类型尺寸相匹配的整数常量或表达式。对于一些不想初始化变量的特殊情况，也可以用符号“?”作为初始值。不论初始值的格式如何，其均由编译器转换为二进制数据。因此，00110100b、34h 和 52d 都会产生同样的二进制值。

3.4.3　定义 BYTE 和 SBYTE 数据

在数据定义语句中使用 BYTE(定义字节)、SBYTE(定义有符号字节)伪指令，可以为一个或多个有符号、无符号字节分配存储空间，每个初始值必须是 8 位的整数表达式或字符常量。例如：

```
val1    BYTE     0          ; 最小的无符号字节常量
val2    BYTE     255        ; 最大的无符号字符常量
val3    BYTE     'A'        ; 字符常量
val4    SBYTE    -127       ; 最小的有符号字节常量
val5    SBYTE    +128       ; 最大的有符号字节常量
```

使用问号代替初始值可以定义未初始化的常量，在可执行指令运行时，这一变量将被动态赋值。

```
val1    BYTE    ?
```

变量名是一个标号，标记相对于段开始位置的偏移。用一个例子来说明，假如 val1 位

于数据段的偏移 0000 处并占用 1 字节的存储空间，那么 val2 将位于段内偏移值 0001 的地方。

```
val1    BYTE    10h
val2    BYTE    20h
```

遗留的 DB 伪指令可以定义有符号或无符号的 8 位变量：

```
val1    DB  255             ; 无符号字节
val2    DB  -128            ; 有符号字节
```

1. 多个初始值

如果一条数据定义语句中有多个初始值，那么标号仅仅代表第一个初始值的偏移。以下面的 list 变量为例，假设 list 位于偏移 0000 处，那么值 10 将位于偏移 0000 处，值 20 位于偏移 0001 处，值 30 位于偏移 0002 处，值 40 位于偏移 0003 处。

```
list  BYTE 10,20,30,40
```

图 3-2 展示了 list 的定义情况。

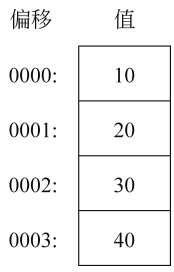

图 3-2 list 的定义情况

并非所有的数据定义都需要标号，如果想在以上 list 的定义基础上继续定义以 list 开始的字节数组，可以在随后的行中继续定义其他数据。

```
list    BYTE 10,20,30,40
BYTE 50,60,70,80
BYTE 81,82,83,84
```

在单条数据定义语句中，初始值可使用不同基数，字符和字符串也可以自由混用。在以下例子中，list1 和 list2 的内容是等同的。

```
list1    BYTE 10,32,41h,00100010b
list2    BYTE 0Ah,20h,'A',22h
```

2. 定义字符串

定义字符串需要将一组字符用单引号或双引号括起来。最常见的字符串是以空字符（即 NULL 字符，等价于数值 0）来结尾的字符串，在其他的语言如 C、C++ 、Java 程序中会使用以下字符串。

```
string1 BYTE    "happy new year",0
string2 BYTE    "Merry Christmas",0
```

这里的每个字符都占用一字节。对于其他类型的初始值来说，每个初始值之间都必须用逗号隔开，而字符串则是一个例外。字符串可以占用多行，不需要为每一行都提供标号，例如：

```
string1 BYTE    "Happy New Year!",0dh,0ah,
        BYTE    "I wish you all"
        BYTE    "all the best.",0
```

十六进制字节 0DH 和 0AH 即为行结束字符（CR/LF，回车换行符），在向标准输出设备上写的时候，回车换行符会将光标移至下一行。

续行符(\)则将两行连接成一条程序语句,续行符只能放在每行的末尾。对于下面两条语句来说,其内容是等价的。

```
string1 BYTE    "Happy New Year"
string1 \
BYTE    "Happy New Year"
```

3. DUP 操作符

DUP 操作符使用一个常量表达式作为计数器为多个数据项来分配存储空间。在为字符串和数组分配空间时,DUP 伪指令非常有用。初始化和未初始化的数据都可以使用 DUP 伪指令来定义。

```
BYTE    10    DUP(0)          ; 10 字节,全部等于 0
BYTE    10    DUP(?)          ; 10 字节,未经初始化
BYTE    3     DUP("ABC")      ; 9 字节,值为"ABCABCABC"
```

3.4.4　定义 WORD 和 SWORD 数据

在数据定义语句中如果使用 WORD(用于定义字)和 SWORD(用于定义有符号字)伪指令就可以为一个或多个 16 位整数分配存储空间,以下给出一些例子:

```
word1   WORD      65535       ; 最大无符号字
word2   SWORD     -32768      ; 最小有符号字
word3   WORD      ?           ; 未初始化的无符号数
```

此外,也可以使用遗留的 DW 指令。

```
dw1     DW        65535       ; 无符号
dw2     DW        -32768      ; 有符号
```

字数组:可以通过显示指令初始化每个元素,或使用 DUP 操作符创建字数组。以下是包含特定初始值的字数组的例子:

```
wordList    WORD    1,2,3,4,5
```

图 3-3 即为该数组在内存中的存储情况,我们假设 wordList 从偏移 0000 处开始,地址以 2 递增(每个元素值要占用 2 字节)。

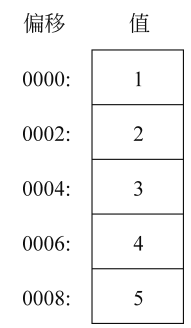

图 3-3　wordList 的存储情况

DUP 操作符为初始化多个字提供方便。

```
array    WORD    5    DUP(?)     ; 5 个未初始化的值
```

3.4.5　定义 DWORD 和 SDWORD 数据

在数据定义语句中使用 DWORD(定义双字)和 SDWORD(定义有符号双字)伪指令,可以为一个或多个 32 位的整数分配存储空间,例如:

```
val1    DWORD     12345678h          ; 无符号数
val2    SDWORD    -2147483648        ; 有符号数
```

```
val1    DWORD    20  DUP(?)       ; 无符号数数组
```

此外，也可以使用遗留的 DD 伪指令。

```
val1    DD       12345678h       ; 无符号
val2    DD       -2147483648     ; 有符号
```

双字数组：所谓双字数组，就是指可以通过显式指令初始化数组每个元素，或使用 DUP 操作符来创建双字数组。例如下面的数组即为一个包含无符号初始值的双字数组：

```
dwordList    DWORD    1,2,3,4,5
```

图 3-4 即为该数组在内存中的存储情况，我们假设 dwordList 从偏移 0000 处开始，注意此时地址以 4 递增。

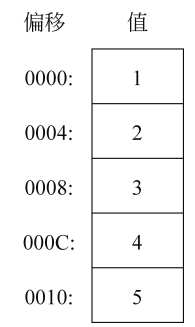

图 3-4 dwordList 的存储情况

3.4.6 定义 QWORD 数据

使用 QWORD(定义 8 字节)伪指令可以定义 64 位的数据。

```
val1    QWORD    1234567812345678h
```

此外，使用遗留的 DQ 伪指令也可以获得相同的效果。

```
val1    DQ    1234567812345678h
```

3.4.7 定义 TBYTE 数据

使用 TBYTE(定义 10 字节)伪指令可以定义 80 位的数据。该数据类型一开始是为了用二进制编码存储的十进制数而准备的，对于这类数据进行操作时，必须使用浮点指令集中的特殊指令。

```
val1 TBYTE 100000000001234567 89Ah
```

此外，也可以使用遗留的 DT 伪指令。

```
val1    DT  100000000001234567 89Ah
```

3.4.8 定义实数

REAL4 定义 4 字节的单精度实数，REAL8 定义 8 字节的双精度实数，REAL10 定义 10 字节的扩展精度实数，对于每个伪指令，都要求一个或多个与其数据尺寸相匹配的实数常量初始值。

```
val1    REAL4    -2.1
val2    REAL8    3.2E-260
val3    REAL10   4.6E+4096
val4    REAL4    20  DUP(0.0)
```

在表 3-5 中，列出了每种实数类型的最小有效位数及其表示范围。

表 3-5　实数类型的最小有效位数及表示范围

数 据 类 型	有效数据位数	表 示 范 围
单精度实数	6	$1.18\times10^{-38}\sim3.40\times10^{38}$
双精度实数	15	$2.23\times10^{-308}\sim1.79\times10^{308}$
扩展精度实数	19	$3.37\times10^{-4932}\sim1.18\times10^{4932}$

此外，遗留的 DD、DQ 和 DT 伪指令也都可以用于定义实数。

```
rVal1    DD  -1.2                    ; 短实数
rVal2    DQ  3.2E-260                ; 长实数
rVal3    DT  4.6E+4096               ; 扩展精度实数
```

3.4.9　小端字节序

所谓小端字节序(little-endian order)，是指 Intel 处理器存取内存数据的一种方案。小端指的是变量的最低有效字节存储在地址值最小的地址单元中，而其余字节将在内存中按照顺序连续存储。

以双字 12345678h 为例，画出其在内存中的存储情况。假设我们将该双字存储在偏移 0 处，78h 会存储在第一字节中，56h 则会存储在第二字节中，其余继续按顺序存储在第三和第四字节中，如图 3-5 所示。

其他有些计算机系统使用大端字节序(big-endian order)来存储内存数据。图 3-6 展示了从偏移 0 开始的双字 12345678h。

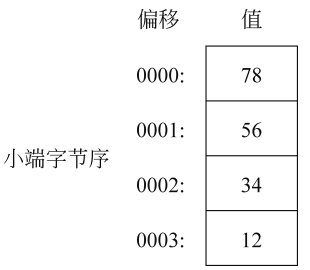

图 3-5　小端字节序存储 12345678h

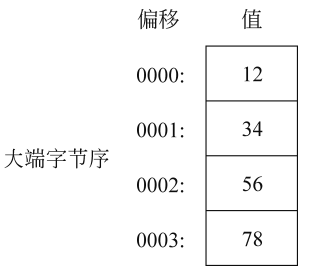

图 3-6　大端字节序存储 12345678h

3.4.10　为 AddSub 程序添加变量

现在重新回顾之前讨论的 AddSub 程序，用学过的数据来定义伪指令，可以再添加一个包含多个双字变量的数据段。经过修改之后的程序称其为 AddSub2：

```
TITLE Add and Subtract                                    (AddSub2.asm)
; This program adds and subtracts 32-bit unsigned.
; integers and stores the sum in a variable.
INCLUDE Irvine32.inc
.data
val1    DWORD    10000h
val2    DWORD    40000h
```

```
val3    DWORD    20000h
finalVal    DWORD    ?
    .code
    main    PROC
        mov eax, val1                          ; start with 10000h
        add eax, val2                          ; add 40000h
        sub eax, val3                          ; subtract 20000h
        mov finalVal, eax                      ; store the result (30000h)
        call DumpRegs                          ; display the registers
        exit
    main ENDP
    END main
```

下面简要解释其工作流程。

首先,变量 val1 里的整数值被送到 EAX 寄存器。

```
mov eax, val1                                  ; start with 10000h
```

然后,变量 val2 中存储的整数值被加载到 EAX 寄存器中。

```
add eax, val2                                  ; add 40000h
```

接着,EAX 寄存器内的整数值减掉变量 val3 内的整数值。

```
sub eax, val3                                  ; subtract 20000h
```

最后,EAX 寄存器内的整数被复制到变量 finalVal 中。

```
mov finalVal, eax                              ; store the result (30000h)
```

3.4.11 未初始化数据的声明

.DATA? 伪指令可以用于声明未初始化的数据,.DATA? 在定义大块的未初始化数据时非常有用,可以有效减小编译后的程序尺寸。以下面的声明语句为例:

```
data
smallArray  DWORD  10   DUP(0)                ; 40 字节
.data?
.bigArray   DWORD  5000 DUP(?)                ; 20000 未初始化字节
```

相反地,以下的例子在编译后会产生大于 20000 字节的程序:

```
.data
smallArray  DWORD  10   DUP(0)                ; 40 字节
bigArray    DWORD  5000 DUP(?)                ; 20000 字节
```

混合代码和数据:汇编器允许程序在代码和数据之间自由切换。对于一些定义在局部程序中使用的变量,这是非常方便的。以下例子在两段代码中直接插入并创建了一个名为 test 的变量:

```
.code
mov eax, ebx
```