Contents

x86 处理器架构 3
32 位 x86 处理器架构 3
x86-64 处理器架构 5
汇编语言基础 5
常用汇编器 5
汇编语言常量 6
汇编语言保留字/关键字 6
标识符 identifier 6
伪指令 directive 6
指令 8
列表文件 listing file 9
数据类型和数据定义 9
数据操作相关运算符和指令 11
操作数类型 11
MOV、MOVZX、MOVSX 指令 12
LAHF、SAHF 指令 12
XCHG 指令 12
加减运算和相关指令 12
OFFSET 运算符 13
ALIGH 伪指令 13
PTR 运算符 13
TYPE 和 SIZEOF 运算符 14
LENGTHOF 运算符 14
LABEL 伪指令 14
TYPEDEF 运算符 14
汇编语言条件判断 15
布尔和比较指令 15
检查奇偶标志 16
置位和清除单个 CPU 标志 17
JMP 指令 17
条件跳转指令 17
LOOP 相关指令 18

32 位条件控制流伪指令 19
汇编语言整数运算 20
逻辑移位和算术移位 20
MUL 指令 21
IMUL 指令 22
DIV 指令 23
IDIV 指令 23
符号扩展指令 23
ADC 和 SBB 指令 23
汇编语言 ASCII 和非压缩十进制运算 24
压缩十进制运算 24
汇编语言过程 25
PUSH 和 POP 及相关指令 25
CALL 和 RET 指令 25
USES 运算符 25
汇编语言高级过程 25
调用规范 25
LEA 指令 27
ENTER 和 LEAVE 指令 27
LOCAL 伪指令 27
INVOKE 伪指令 28
ADDR 运算符 28
PROC 和 ENDP 伪指令 28
PROTO 伪指令 29
EXTERN 伪指令 29
Java 虚拟机 JVM 工作原理 30
汇编语言字符串和数组 30
字符串基本指令 30
汇编语言结构和宏 31
结构体 31
联合 union 32
宏过程简述 33
条件汇编伪指令简述 34

宏汇编运算符简介 34
宏函数 35
重复语句伪指令 35
浮点数处理与指令编码 36
FPU 浮点数计算单元简介 36
浮点数异常 37
浮点数指令集 37
FINIT 初始化 38
FLD 加载浮点数值 38
FST，FSTP 保存浮点数值 38
FCHS 和 FABS 38
FADD、FADDP、FIADD 38
FSUB、FSUBP、FISUB 39
FMUL、FMULP、FIMUL 39
FDIV、FDIVP、FIDIV 40
FCOM、FCOMP、FCOMPP 40
FWAIT 指令（WAIT） 41
其他 FPU 指令 41
高级语言接口 42
.MODEL 伪指令 42
__asm 伪指令 42
部分汇编器特殊语法积累 43
MASM 43
Visual Studio 常用功能积累 43
查看 OBJ 文件中所有过程名 43
查看编译器生成的汇编语言代码 43

x86 处理器架构

32 位 x86 处理器架构

1）操作模式：

保护模式 protected mode——一般的支持虚拟内存的模式
虚拟 8086 模式 Virtual-8086——8086 虚拟机，仅 1MB 内存，但可以创建多个
实地址模式 Real-Address——MS-DOS 模式，直接寻址，仅 1MB 内存和单程序
系统管理模式 System Management——仅由操作系统和机器开发使用

2）地址空间：
最大 4GB，从 P6 处理器开始，一种被称为扩展物理寻址 (extended physical addressing) 的技术使得可以被寻址的物理内存空间增加到 64GB
3）基本程序执行寄存器 basic program execution registers
——8 个 32 位通用寄存器：
EAX：扩展累加器 extended accumulator，乘除指令默认使用。也用于子程序存放返回值。
EBX：
ECX：CPU 默认使用 ECX 为循环计数器
EDX：
EBP：扩展帧指针 extended frame pointer，用于引用堆栈数据，指向当前调用堆栈帧首地址。极少作算术和传输。
ESP：扩展堆栈指针 extended stack pointer，用于寻址堆栈数据，指向调用堆栈的顶部地址。极少作算术和传输。
ESI：扩展源变址 extended source index，用于高速存储器传输指令，也常常用作基址-偏移量寻址中存放指针数据。
EDI：扩展目的变址 extended destination index，用于高速存储器传输指令
注：EAX 的低 16 位叫 AX，AX 的高低 8 位分别叫 AH 和 AL，以此类推（仅限 ABCD 这 4 个通用寄存器可以拆开使用）。
——6 个 16 位段寄存器：
CS
SS
DS
ES
FS
GS
注：实地址模式中，16 位段寄存器表示的是预先分配的内存区域的基址，这个内存区域称为段。保护模式中，段寄存器中存放的是段描述符表指针。
——1 个指令指针寄存器（EIP）：包含下一条将要执行指令的地址
——1 个处理器状态标志寄存器（EFLAGS）：
控制标志位——控制标志位控制 CPU 的操作。
状态标志位——
进位标志位 CF——无符号算术运算结果太大或为负时，设置该标志位
溢出标志位 OF——有符号算术运算结果太大或太小时，设置该标志
符号标志位 SF——算术或逻辑操作产生负结果时，设置该标志位。
零标志位 ZF——算术或逻辑操作产生的结果为零时，设置该标志位。
辅助进位标志位 AC——算术操作在 8 位操作数中产生了位 3 向位 4 的进位
奇偶校验标志位 PF——结果的最低有效字节包含偶数个 1 时设置，否则清除标志
——MMX 寄存器：8 个 64 位 MMX 寄存器支持称为 SIMD 的特殊指令
——XMM 寄存器：8 个 128 位 XMM 寄存器，它们被用于 SIMD 流扩展指令集
——浮点单元 FPU, floating-point unit：执行高速浮点算术运算。FPU 中有 8 个 80 位浮点数据寄存器，ST(0)~ST(7)。还有 48 位指令指针寄存器，48 位数据指针寄存器，16 位标识寄存器，16 位控制寄存器，16 位状态寄存器，操作码寄存器。

x86-64 处理器架构

1）操作模式 IA-32e：

兼容模式

64 位模式

2）基本执行环境：

64 位模式下，虽然处理器现在只能支持 48 位的地址，但是理论上，地址最大为 64 位。从寄存器来看，64 位模式与 32 位最主要的区别如下所示：

16 个 64 位通用寄存器（32 位模式只有 8 个通用寄存器）

8 个 80 位浮点寄存器

1 个 64 位状态标志寄存器 RFLAGS （只使用低 32 位）

1 个 64 位指令指针寄存器 RIP

32 位标志寄存器和指令指针寄存器分别称为 EFLAGS 和 EIP。此外，还有一些 x86 处理器用于多媒体处理的特殊寄存器：

8 个 64 位 MMX 寄存器

16 个 128 位 XMM 寄存器（32 位模式只有 8 个 XMM 寄存器）

3）通用寄存器的变化：

64 位模式下，操作数的默认大小是 32 位，并且有 8 个通用寄存器。但是，给每条指令加上 REX（寄存器扩展）前缀后，操作数可以达到 64 位，可用通用寄存器的数量也增加到 16 个：32 位模式下的寄存器，再加上 8 个有标号的寄存器，R8 到 R15。下表给出了 REX 前缀下可用的寄存器。

操作数大小	可用寄存器
8 位	AL、BL、CL、DL、DIL、SIL、BPL、SPL、R8L、R9L、R10L、R11L、R12L、R13L、R14L、R15L
16 位	AX、BX、CX、DX、DI、SI、BP、SP、R8W、R9W、R10W、R11W、R12W、R13W、R14W、R15W
32 位	EAX、EBX、ECX、EDX、EDI、ESI、EBP、ESP、R8D、R9D、R10D、R11D、R12D、R13D、R14D、R15D
64 位	RAX、RBX、RCX、RDX、RDI、RSI、RBP、RSP、R8、R9、R10、R11、R12、R13、R14、R15

还有一些需要记住的细节：

• 64 位模式下，单条指令不能同时访问寄存器高字节，如 AH、BH、CH 和 DH，以及新字节寄存器的低字节（如 DIL）。

• 64 位模式下，32 位 EFLAGS 寄存器由 64 位 RFLAGS 寄存器取代。这两个寄存器共享低 32 位，而 RFLAGS 的高 32 位是不使用的。

• 32 位模式和 64 位模式具有相同的状态标志。

汇编语言基础

常用汇编器

Microsoft 宏汇编器（称为 MASM）

TASM（Turbo 汇编器）

NASM（Netwide 汇编器）

MASM32（MASM 的一种变体）

GAS（GNU 汇编器）和 NASM 是两种基于 Linux 的汇编器

• 在这些汇编器中，NASM 的语法与 MASM 的最相似
• 汇编语言的指令虽然一般来讲是和机器指令对应的，是否是一对一取决于指令集具体的结构设计，但汇编语言已经是人类可读的最接近机器语言逻辑的语言形式，它也是一种编程语言。如何将其转化为真正的机器指令和可执行程序仍需要汇编器做很多除了翻译以外的工作，亦即汇编指令的编译器。而使得程序员可以在基础操作外更轻松使用更多编程功能的要素便是伪指令和运算符，它们实际上就是代替了一些经常需要使用的汇编指令序列。因此可以说含伪指令的汇编程序会被编译为不含伪指令的汇编程序（亦即编译过程中的中间步骤），进而编译为目标代码。

汇编语言常量

指令层面，常量被直接编码于指令，所以无法修改。

汇编语言保留字/关键字

保留字（reserved words）有特殊意义并且只能在其正确的上下文中使用。默认情况下，保留字是没有大小写之分的。比如，MOV 与 mov、Mov 是相同的。
保留字有不同的类型：

指令助记符，如 MOV、ADD 和 MUL。
寄存器名称。
伪指令，告诉汇编器如何汇编程序。
属性，提供变量和操作数的大小与使用信息。例如 BYTE 和 WORD。
运算符，在常量表达式中使用。
预定义符号，比如 @data，它在汇编时返回常量的整数值。

标识符 identifier

标识符是由程序员选择的名称，它用于标识变量、常数、子程序和代码标签。
标识符的形成有一些规则：

可以包含 1 到 247 个字符。
不区分大小写。
首字符须为字母 (A---Z, a---z) 、下划线 (_)、@、? 或 $。其后的字符可以是数字。
标识符不能与汇编器保留字相同。

提示：可以在运行汇编器时，添加 -Cp 命令行切换项来使得所有关键字和标识符变成大小写敏感。

伪指令 directive

伪指令是嵌入源代码中的命令，由汇编器识别和执行。伪指令不在运行时执行，但是它们可以定义变量、宏和子程序；为内存段分配名称，执行许多其他与汇编器相关的日常任务。默认情况下，伪指令不区分大小写。
尽管 Intel 处理器所有的汇编器使用相同的指令集，但是通常它们有着不同的伪指令。比如，Microsoft 汇编器的 REPT 伪指令对其他一些汇编器就是无法识别的。一般由.点号开始的命令、带有全部大写标识符的命令、不符合一般指令格式的命令都是伪指令。
汇编器伪指令的一个重要功能是定义程序区段，也称为段 (segment)。程序中的段具有不同的作用。
1）段声明伪指令

.data 一个用于定义变量的段，未初始化数据在此段仍然占用空间
.data? 一个用来声明未初始化数据的段。未初始化数据在此段不占用空间
.code 伪指令标识的程序区段包含了可执行的指令
2）系统设置与调用伪指令
32 位：
.stack
　　标识的程序区段定义了运行时堆栈，并设置了其大小：.stack 100h
.386
　　它表示这是一个 32 位程序，能访问 32 位寄存器和地址。
.model memory_mode call_convention
　　可选择程序的内存模式（如 flat）并确定子程序的调用规范（如 stdcall）
function PROTO, paraname1:type [,para2:type…]
　　声明子程序原型和参数列表
INVOKE function, arg1 [,arg2…]
　　调用子程序 function 并附带参数列表
END main
　　标识程序结尾位置以及程序的入口（main）。
64 位：
function PROTO
　　声明子程序原型，无参数列表，参数由寄存器通过 mov 传输，调用通过 call 指令
END
　　标识程序结尾位置，不需注明程序入口
3）等号=、EQU、TEXTEQU 伪指令
①等号伪指令（equal-sign directive）把一个符号名称与一个整数表达式连接起来，这在汇编语言中相当于宏定义。
　　name = expression
②EQU 伪指令把一个符号名称与一个整数表达式或一个任意文本连接起来，它有 3 种格式：
　　name EQU expression
　　name EQU symbol
　　name EQU <text>
第一种格式中，expression 必须是一个有效整数表达式。第二种格式中，symbol 是一个已存在的符号名称，已经用 = 或 EQU 定义过了。第三种格式中，任何文本都可以出现在<...>内。当汇编器在程序后面遇到 name 时，它就用整数值或文本来代替符号。
③TEXTEQU 伪指令，类似于 EQU，创建了文本宏（text macro）。它有 3 种格式：第一种为名称分配的是文本；第二种分配的是已有文本宏的内容；第三种分配的是整数常量表达式：
　　name TEXTEQU <text>
　　name TEXTEQU textmacro
　　name TEXTEQU %constExpr　;%运算符的介绍见后文
文本宏可以相互嵌套构建。
• 注意：在同一源代码文件中，用 EQU 定义的符号不能被重新定义，而等号=和 TEXTEQU 定义的符号可随时重新定义。
4）$运算符（当前地址计数器）

$运算符（当前地址计数器）返回当前程序语句的偏移量。在下例中，从当前地址计数器（$）中减去 list 的偏移量，计算得到 ListSize：

```
list BYTE 10,20,30,40
ListSize = ($ - list)
```

这是$运算符最常用的用法之一，即计算字符串或数组长度。
ListSize 必须紧跟在 list 的后面。且如果类型是 BYTE 长度的 n 倍，那相应的结果需要缩小 n 倍。

指令

一条指令有四个组成部分：

标号（可选）
指令助记符（必需）
操作数（通常是必需的）
注释（可选）

不同部分的位置安排如下所示：

```
(label:) mnemonic (operands) (;comment)
```

1）标号
标号（label）是一种标识符，是指令和数据的位置标记。标号位于指令的前端，表示指令的地址。同样，标号也位于变量的前端，表示变量的地址。标号有两种类型：数据标号和代码标号。
• 数据标号标识变量的位置，它提供了一种方便的手段在代码中引用该变量。比如，下面定义了一个名为 count 的变量（count 实则代表一个数据区的偏移位置）：

```
count DWORD 100
```

汇编器为每个标号分配一个数字地址。可以在一个标号后面定义多个数据项。在下面的例子中，array 定义了第一个数字（1024）的位置，其他数字在内存中的位置紧随其后：

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

上例也可以写成：（对于 LENGTHOF 运算符有区别）

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

• 程序代码区（指令所在区段）的标号必须用冒号（:）结束。这样的标号仅在当前过程（函数、方法、子程序）中有效，解除这个限制的方法是定义全局标号，即在名字后面加双冒号 (::)。代码标号用作跳转和循环指令的目标。例如，下面的 JMP 指令创建一个循环，将程序控制传递给标号 target 标识的位置：

```
target:
      mov ax,bx
      ...
      jmp target
```

代码标号可以与指令在同一行上，也可以自己独立一行。标号命名规则要求，只要每个标号在其封闭子程序页中是唯一的，那么就可以多次使用相同的标号。
2）指令助记符
3）操作数
操作数是指令输入输出的数值。汇编语言指令操作数的个数范围是 0～3 个，每个操作数可以是寄存器、内存操作数、整数表达式和输入输出端口。

生成内存操作数有不同的方法，比如，使用变量名、带方括号的寄存器等。变量名暗示了变量地址，并指示计算机使用给定地址的内存内容。
操作数有固有顺序。当指令有多个操作数时，通常第一个操作数被称为目的操作数，第二个及之后的操作数被称为源操作数（source operand）。一般源操作数用来计算，目的操作数用来修改和写入。当只有一个操作数时，可能是源操作数或目的操作数（如 PUSH 和 POP）。
4）注释
注释有两种指定方法：
单行注释，用分号（;）开始。汇编器将忽略在同一行上分号之后的所有字符。
块注释，用 COMMENT 伪指令和一个用户定义的符号开始。汇编器将忽略其后所有的文本行，直到相同的用户定义符号出现为止。
示例如下：

```
COMMENT !
        This line is a comment.
        This line is also a comment.
!
```

5）NOP（空操作）指令
最安全（也是最无用）的指令是 NOP（空操作）。它在程序空间中占有一个字节，但是不做任何操作。它有时被编译器和汇编器用于将代码对齐到有效的地址边界。

列表文件 listing file

列表文件 (listing file) 包括了程序源文件的副本，再加上行号、每条指令的数字地址、每条指令的机器代码字节（十六进制）以及符号表。符号表中包含了程序中所有标识符的名称、段和相关信息。
若想告诉 Visual Studio 生成列表文件，则在打开项目时按下述步骤操作：在 Project 菜单中选择 Properties，在 Configuration Properties 下，选择 Microsoft Macro Assemblero 然后选择 Listing File。在对话框中，设置 Generate Preprocessed Source Listing 为 Yes，设置 List All Available Information 为 Yes。

数据类型和数据定义

汇编语言中的数据类型，汇编器不检查格式，只是规限宽度。类型检查需要程序员自行完成，汇编器只负责按照声明类型进行解读一串数字而已。
• 下表给出了全部内部数据类型的列表：

类型	用法
BYTE	8 位无符号整数，B 代表字节
SBYTE	8 位有符号整数，S 代表有符号
WORD	16 位无符号整数
SWORD	16 位有符号整数
DWORD	32 位无符号整数，D 代表双（字）
SDWORD	32 位有符号整数，SD 代表有符号双（字）
FWORD	48 位整数（保护模式中的远指针）
QWORD	64 位整数，Q 代表四（字）
TBYTE	80 位（10 字节）整数，T 代表 10 字节
REAL4	32 位（4 字节）IEEE 短实数

REAL8	64 位（8 字节）IEEE 长实数
REAL10	80 位（10 字节）IEEE 扩展实数

• 此外，还有各种限定类型（qualified type），如指向现有类型的指针。限定类型还能够用 TYPEDEF 和 STRUCT 伪指令创建。下面是限定类型的例子：

```
PTR BYTE  PTR SBYTE
PTR WORD  PTR SWORD
PTR DWORD  PTR SDWORD
PTR QWORD  PTR TBYTE
```

1）数据定义语句

数据定义语句（data definition statement）在内存中为变量留出存储空间，并赋予一个可选的名字。数据定义语句根据内部数据类型（上表）定义变量。

数据定义语法如下所示：

```
[name] directive initializer [,initializer]...
```

• directive：该伪指令就是用来表示类型的。此外，它还可以是传统数据定义伪指令，如下表所示。

伪指令	用法	伪指令	用法
DB	8 位整数	DQ	64 位整数或实数
DW	16 位整数	DT	定义 80 位（10 字节）整数
DD	32 位整数或实数		

• initializer：数据定义中至少要有一个初始值，即使该值为 0。若声明后续其他初始值，用逗号分隔，视为数组声明。若数组声明中途换行，参考下例：

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

如果程序员希望不对变量进行初始化（随机分配数值），可以用符号 ？ 作为初始值。所有初始值，不论其格式，都由汇编器转换为二进制数据。

2）字符串

定义一个字符串实则为 BYTE 类型的数组（但是不必真的用逗号分隔每个字符那样定义），要用单引号或双引号将其括起来。最常见的字符串类型是用一个空字节（值为 0）作为结束标记，称为以空字节结束的字符串，很多编程语言中都使用这种类型的字符串：

```
greeting1 BYTE "Good afternoon",0
```

字符串可以换行/多行定义，相当于每一行声明一个字符串，行末不加逗号。

• DUP 操作符使用一个整数表达式作为计数器，为多个数据项分配存储空间。在为字符串或数组分配存储空间时，这个操作符非常有用，它可以使用初始化或非初始化数据：

```
BYTE 20 DUP ( 0 )       ;20 个字节，值都为 0
BYTE 20 DUP ( ? )       ;20 个字节，非初始化
BYTE 4 DUP ( "STACK" ) ; 20 个字节
```

另外，它还可以用于简单声明多维数组：

```
array2 WORD 5 DUP(3 DUP(?))
```

3）某些类型的特殊用法

• DWORD（32 位模式下）

还可以用于声明一种变量，这种变量包含的是另一个变量的数据区 32 位偏移量。如下所示，pVal 包含的就是 val3 的偏移量：

```
pVal DWORD (OFFSET) val3
```

pVal 可以理解为一种指针，而 DWORD 则是其宽度类型，和高级语言比较下也侧面说明了汇编语言中的类型含义。OFFSET 运算符可省略。

• QWORD

64 位模式下，QWORD 是偏移量（指针）的长度类型。用法与 DWORD 一样。

• TBYTE

该类型是把一个压缩的二进制编码的十进制（BCD, Binary Coded Decimal）整数存放在一个 10 字节的包中。每个字节（除了最高字节之外）包含两个十进制数字。在低 9 个存储字节中，每半个字节都存放了一个十进制数字。最高字节中，最高位表示该数的符号位。

数据操作相关运算符和指令

操作数类型

指令包含的操作数个数可以是：0 个，1 个，2 个或 3 个。如果有操作数，那第一个就是目的操作数，后续 0-2 个操作数都是源操作数。操作数有以下基本类型：

①立即数——用数字文本表达式

②寄存器操作数——使用 CPU 内已命名的寄存器

③内存操作数——引用内存位置，一般是数据区标号（变量）或偏移量表达式

• 直接-偏移量操作数——变量名加上一个位移。这样可以访问那些没有显式标记的内存位置。如：

```
arrayB BYTE 10h,20h,30h,40h,50h
mov al,[arrayB+1] ;AL = 20h,[]可以不加，但是习惯上加，因为 arrayB 是地址
```

• 间接操作数——寄存器名称加[]，寄存器内容为数据的地址：

```
[reg + n] ;寄存器内容加常数
```

• 变址操作数——寄存器加数据区标号产生一个内存位置，寄存器内容为相对偏移量：

```
data_name [reg * TYPE data_name] ;寄存器内容索引再乘数据宽度（比例因子）
data_name [reg] ;寄存器内容加数据区标号
[data_name + reg] ;两种写法都可以
```

• 基址-变址操作数——两个寄存器内容相加，构成一个地址

```
[reg1 + reg2 * TYPE data_name]
```

• 基址-变址-偏移量操作数——

```
[base + index + data_name]
data_name[base + index]
```

• 下表说明了标准操作数类型：

操作数	说明
reg8	8 位通用寄存器：AH、AL、BH、BL、CH、CL、DH、DL
reg16	16 位通用寄存器：AX、BX、CX、DX、SI、DI、SP、BP
reg32	32 位通用寄存器：EAX、EEX、ECX、EDX、ESI、EDI、ESP、EBP
reg	通用寄存器
sreg	16 位段寄存器：CS、DS、SS、ES、FS、GS
imm	8 位、16 位或 32 位立即数
imm8	8 位立即数，字节型数值
imm16	16 位立即数，字类型数值
imm32	32 位立即数，双字型数值

reg/mem8	8 位操作数，可以是 8 位通用寄存器或内存字节
reg/mem16	16 位立即数，可以是 16 位通用寄存器或内存字
reg/mem32	32 位立即数，可以是 32 位通用寄存器或内存双字
mem	8 位、16 位或 32 位内存操作数

MOV、MOVZX、MOVSX 指令

1）32 位

• MOV 指令将源操作数复制到目的操作数。作为数据传送（data transfer）指令，只要按照如下原则。

两个操作数必须是同样的大小。

两个操作数不能同时为内存操作数。

指令指针寄存器（IP、EIP 或 RIP）不能作为目标操作数。

• MOVZX 指令（进行全零扩展并传送）将源操作数复制到目的操作数，并把目的操作数 0 扩展到 16 位或 32 位。这条指令只用于无符号整数，且遵循以下原则。

源操作数不能是常数。

目的操作数只能是寄存器。

源操作数尺寸必须小于目的操作数。

• MOVSX 指令（进行符号扩展并传送）将源操作数内容复制到目的操作数，并把目的操作数符号扩展到 16 位或 32 位。这条指令只用于有符号整数。

2）64 位

64 位模式下的 MOV 指令与 32 位模式下的有很多共同点，只有几点区别：

①立即操作数（常数）——当一个 8、16 或 32 位常数送入 64 位寄存器时，目标操作数的高 32 位（位 32—位 63）被清除（等于 0）。

②内存操作数——传送一个 32 位内存操作数到 EAX（RAX 寄存器的低半部分），就会清除 RAX 的高 32 位。但是，如果是将 8 位或 16 位内存操作数送入 RAX 的低位，那么，目标寄存器的高位不受影响。

③MOVSXD 指令（符号扩展传送）——允许源操作数为 32 位寄存器或内存操作数，并符号扩展至 64 位。

④MOVZXD 指令？

LAHF、SAHF 指令

• LAHF（加载状态标志位到 AH）指令将 EFLAGS 寄存器的低字节复制到 AH。被复制的标志位包括：符号标志位、零标志位、辅助进位标志位、奇偶标志位和进位标志位。使用这条指令，可以方便地把标志位副本保管在变量中。

• SAHF（保存 AH 内容到状态标志位）指令将 AH 内容复制到 EFLAGS（或 RFLAGS）寄存器低字节。

• 两指令均无操作数。

XCHG 指令

XCHG（交换数据）指令交换两个操作数内容。除了 XCHG 指令不使用立即数作操作数之外，XCHG 指令操作数的要求与 MOV 指令操作数要求是一样的。

加减运算和相关指令

1）INC 和 DEC

INC（增加）和 DEC（减少）指令分别表示寄存器或内存操作数加 1 和减 1，只有一个操作数。注意 INC 和 DEC 指令不会影响进位标志位，但可能会影响其他标志位。

2）ADD 和 SUB

ADD 指令将长度相同的源操作数和目的操作数进行相加操作。SUB 指令从目的操作数中减去源操作数。语法如下：

```
ADD dest,source
SUB dest, source
```

在操作中，源操作数不能改变，相加之和存放在目的操作数中。该指令可以使用的操作数与 MOV 指令相同。

3）NEG

NEG（非）指令将操作数的符号取反，这是通过把寄存器或内存操作数转换为其二进制补码来完成的。在非零操作数上应用 NEG 指令总是会将进位标志位置 1。

4）标志位

汇编语言没有类型和数值有效范围检测，CPU 也只是按照特定逻辑设置标志位而不管实际运算类型。程序员需要根据运算类型自行设计和分析部分标志位，并忽略其他标志。

5）寄存器宽度范围的影响

执行计算时，需要时刻留意所使用的操作数的大小，当操作数只使用部分寄存器时，要注意寄存器的其他部分是没有被修改的。比如只用 AL 和 BL 进行加法运算，即便产生进位也不会改变 AX、EAX 和 RAX 等寄存器中的高位，即进位被忽略。

OFFSET 运算符

OFFSET 运算符返回数据标号的偏移量。这个偏移量按字节计算，表示的是该数据标号距离数据段起始地址的距离。OFFSET 运算符的优先级高于+-法。如

```
OFFSET data_seg_label_name ;若不加 OFFSET，则标号表示的是其存储内容
```

注意 OFFSET name + n 和 name + n 的区别，前者结果是偏移量（地址），后者是内容。这也是为何习惯写作[name + n]，因为括号内的是地址，其表达式结果是内容。

- 64 位模式下，OFFSET 运算符产生 64 位地址，必须用 64 位寄存器或变量来保存。

ALIGH 伪指令

ALIGN 伪指令将下一个变量对齐到字节边界、字边界、双字边界或段落边界。语法如下：

```
ALIGN bound
```

Bound 可取值有：1、2、4、8、16 等，代表字节位数，也可以直接使用类型代替 bound，如 DWORD 相当于 bound 取 4。当取值为 1 时，则下一个变量对齐于 1 字节边界（默认情况）。当取值为 2 时，则下一个变量对齐于偶数地址。当取值为 4 时，则下一个变量地址为 4 的倍数。当取值为 16 时，则下一个变量地址为 16 的倍数，即一个段落的边界。为了满足对齐要求，汇编器会在变量前插入一个或多个空字节。

PTR 运算符

PTR 运算符用以返回自某个内存位置开始的特定大小（类型）的数据内容。它可以用来重写一个已经被声明过的操作数的大小类型。只要试图用不同于汇编器设定的大小属性来访问操作数，那么这个运算符就是必需的。如：

```
type_name PTR data_seg_offset
```

- 该操作符返回从 offset 起始的以 type_name 为长度的内容。

• type_name：PTR 必须与一个标准汇编数据类型一起使用，这些类型包括：BYTE、SEYTE、WORD、SWORD、DWORD、SDWORD、FWORD、QWORD 或 TBYTE。重写类型可以比原类型大或小或相同（但起始于不同位置），当然范围和有效性检测需自行解决。
• data_seg_offset：数据区偏移量，可以是变量名、偏移量表达式（因为表达式必须先计算因而不能省略[]）
• 间接操作数作为偏移量执行某些指令时，汇编器不知道该数据的长度，进而需要 PTR 运算符指定。如：

```
inc BYTE PTR [esi] ;若不用 PTR 运算符，汇编器不知道对多长的数据加 1
```

TYPE 和 SIZEOF 运算符

TYPE 运算符返回变量单个元素的大小，这个大小是以字节为单位计算的。注意 TYPE 后接结构体类型名时，返回结构体的整体大小，其结果和 SIZEOF 伪指令一致。但其后接数组名称标识符时，仅返回单个元素的大小，而 SIZEOF 则返回数组整体大小。

LENGTHOF 运算符

LENGTHOF 运算符计算数组中元素的个数，元素个数是由数组标号同一行出现的数值来定义的。如果数组定义占据了多个程序行，每行都用 type_name 重新声明，那么 LENGTHOF 只针对第一行定义的数据。但若使用逗号换行，新行无 type_name 声明，则可返回全长。

LABEL 伪指令

LABEL 伪指令可以插入一个标号，并定义它的大小属性，但是不为这个标号分配存储空间。LABEL 中可以使用所有的标准大小属性，如 BYTE、WORD、DWORD、QWORD 或 TBYTE。LABEL 常见的用法是，为数据段中定义的下一个变量提供不同的名称和大小属性。
如下例所示，在变量 val32 前定义了一个变量，名称为 val16 属性为 WORD：

```
.data
val16 LABEL WORD
val32 DWORD 12345678h
.code
mov ax,val16          ; AX = 5678h
mov dx,[val16+2]      ; DX = 1234h
```

val16 与 val32 共享同一个内存位置。LABEL 伪指令自身不分配内存。另外，LABEL 修饰的标号类型可以大于后续的标号类型，即使用 LABEL 标号引用后续多个标号内容。

TYPEDEF 运算符

TYPEDEF 运算符可以创建用户定义类型，这些类型包含了定义变量时内置类型的所有状态。它是创建指针变量的理想工具。比如，下面声明创建的一个新数据类型 PBYTE 就是一个字节指针：

```
PBYTE TYPEDEF PTR BYTE
```

这个声明通常放在靠近程序开始的地方，在数据段之前。然后，变量就可以用 PBYTE 来定义：

```
.data
arrayB BYTE 10h,20h,30h,40h
ptr1 PBYTE ? ;未初始化
ptr2 PBYTE arrayB ;指向一个数组
```

汇编语言条件判断

布尔和比较指令

1）AND 指令

• AND 指令在两个操作数的对应位之间进行（按位）逻辑与（AND）操作，并将结果存放在目标操作数中（不能是立即数）：

```
AND destination,source
```

• 操作数可以是 8 位、16 位、32 位和 64 位，但是两个操作数必须是同样大小。源操作数中立即操作数不能超过 32 位。

• AND 指令总是清除溢出和进位标志位，并根据目标操作数的值来修改符号标志位、零标志位和奇偶标志位。

2）OR 指令

• OR 指令在两个操作数的对应位之间进行（按位）逻辑或（OR）操作，并将结果存放在目标操作数中：

```
OR destination, source
```

• OR 指令操作数组合与 AND 指令相同。

• OR 指令总是清除进位和溢出标志位，并根据目标操作数的值来修改符号标志位、零标志位和奇偶标志位。

3）XOR 指令

• XOR 指令在两个操作数的对应位之间进行（按位）逻辑异或（XOR）操作，并将结果存放在目标操作数中：

```
XOR destination, source
```

• XOR 指令操作数组合和大小与 AND 指令及 OR 指令相同。

• XOR 指令总是清除溢出和进位标志位，并根据目标操作数的值来修改符号标志位、零标志位和奇偶标志位。

4）64 位模式和操作数不等长时的现象

• 大多数情况下，64 位模式中的 64 位指令与 32 位模式中的操作是一样的。比如，如果源操作数是常数，长度小于 32 位，而目的操作数是一个 64 位寄存器或内存操作数，那么，目的操作数中所有的位都会受到影响。但是，如果源操作数是 32 位常数或寄存器，那么目的操作数中，只有低 32 位会受到影响。当目的操作数是内存操作数时，得到的结果是一样的。显然，32 位操作数是一个特殊的情况，需要与其他大小操作数的情况分开考虑。

5）NOT 指令

• NOT 指令触发（翻转）操作数中的所有位。其结果被称为反码。

• NOT 指令不影响标志位。

6）TEST 指令

• TEST 指令在两个操作数的对应位之间进行 AND 操作，并根据运算结果设置符号标志位、零标志位和奇偶标志位。

• TEST 与 AND 指令唯一不同的地方是，TEST 指令不修改目标操作数。TEST 指令允许的操作数组合与 AND 指令相同。在发现操作数中单个位是否置位时，TEST 指令非常有用。

• TEST 指令总是清除溢出和进位标志位，其修改符号标志位、零标志位和奇偶标志位的方法与 AND 指令相同。

7）CMP 指令

• x86 汇编语言用 CMP 指令比较整数。字符代码也是整数，因此可以用 CMP 指令。CMP（比较）指令执行从目的操作数中减去源操作数的隐含减法操作，并且不修改任何操作数：

```
CMP destination,source
```

• 标志位

当实际的减法发生时，CMP 指令按照计算结果修改溢出、符号、零、进位、辅助进位和奇偶标志位。

如果比较的是两个无符号数，则零标志位和进位标志位表示的两个操作数之间的关系如右表所示：

CMP 结果	ZF	CF
目的操作数 < 源操作数	0	1
目的操作数 > 源操作数	0	0
目的操作数 = 源操作数	1	0

如果比较的是两个有符号数，则符号标志位、零标志位和溢出标志位表示的两个操作数之间的关系如右表所示：

CMP 结果	标志位
目的操作数 < 源操作数	SF ≠ OF
目的操作数 > 源操作数	SF=OF
目的操作数 = 源操作数	ZF=1

检查奇偶标志

奇偶检查是在一个二进制数上实现的功能，计算该数中 1 的个数；如果计算结果为偶数，则说该数是偶校验；如果结果为奇数，则该数为奇校验。

x86 处理器中，当按位操作或算术操作的目标操作数最低字节为偶校验时，奇偶标志位置 1。反之，如果操作数为奇校验，则奇偶标志位清 0。一个既能检查数的奇偶性，又不会修改其数值的有效方法是，将该数与 0 进行异或运算：

```
mov al,10110101b              ;5 个 1,奇校验
xor al, 0                                ;奇偶标志位清 0 （奇）
mov al, 11001100b             ;4 个 1，偶校验
xor al, 0                                ;奇偶标志位置 1（偶）
```

Visual Studio 用 PE=1 表示偶校验，PE=0 表示奇校验。

• 16 位奇偶性

对 16 位整数来说，可以通过将其高字节和低字节进行异或运算来检测数的奇偶性：

```
mov ax,64C1h     ;0110 0100 1100 0001
xor ah, al               ;奇偶标志位置 1 （偶）
```

将每个寄存器中的置 1 位（等于 1 的位）想象为一个 8 位集合中的成员。XOR 指令把两个集合交集中的成员清 0，并形成了其余位的并集。这个并集的奇偶性与整个 16 位整数的奇偶性相同。

• 32 位奇偶性

如果将数值的字节进行编号，从 B_0 到 B_3 那么计算奇偶性的表达式为:B_0 XOR B_1 XOR B_2 XOR B_3。

置位和清除单个 CPU 标志

• 要将零标志位置 1，就把操作数与 0 进行 TEST 或 AND 操作；要将零标志位清零，就把操作数与 1 进行 OR 操作：

```
test al, 0            ;零标志位置 1
and al, 0            ;零标志位置 1
or al, 1               ;零标志位清零
```

• TEST 指令不修改目的操作数，而 AND 指令则会修改目的操作数。若要符号标志位置 1，将操作数的最高位和 1 进行 OR 操作；若要清除符号标志位，则将操作数最高位和 0 进行 AND 操作：

```
or al, 80h          ;符号标志位置 1
and al, 7Fh        ;符号标志位清零
```

• 若要进位标志位置 1，用 STC 指令；清除进位标志位，用 CLC 指令：

```
stc                    ;进位标志位置 1
clc                    ;进位标志位清零
```

• 若要溢出标志位置 1，就把两个正数相加使之产生负的和数；若要清除溢出标志位，则将操作数和 0 进行 OR 操作：

```
mov al,7Fh        ; AL = +127
inc al                  ; AL = 80h (-128), OF=1
or eax, 0             ; 溢出标志位清零
```

JMP 指令

JMP 指令无条件跳转到目标地址，该地址用代码标号来标识，并被汇编器转换为偏移量。语法如下所示：

```
JMP destination
```

当 CPU 执行一个无条件转移时，目标地址的偏移量被送入指令指针寄存器，从而导致迈从新地址开始继续执行。

条件跳转指令

x86 指令集包含大量的条件跳转指令。它们能比较有符号和无符号整数，并根据单个 CPU 标志位的值来执行操作。条件跳转指令可以分为四个类型：

1）基于特定标志位的值跳转

下表展示了基于零标志位、进位标志位、溢出标志位、奇偶标志位和符号标志位的跳转。

助记符	说明	标志位/寄存器	助记符	说明	标志位/寄存器
JZ	为零跳转	ZF=1	JNO	无溢出跳转	OF=0
JNZ	非零跳转	ZF=0	JS	有符号跳转	SF=1
JC	进位跳转	CF=1	JNS	无符号跳转	SF=0
JNC	无进位跳转	CF=0	JP	偶校验跳转	PF=1
JO	溢出跳转	OF=1	JNP	奇校验跳转	PF=0

2）基于两数是否相等，或是否等于（E）CX 的值跳转

下表列出了基于相等性评估的跳转指令。有些情况下，进行比较的是两个操作数；其他情况下，则是基于 CX、ECX 或 RCX 的值进行跳转。表中符号 leftOp 和 rightOp 分别指的是 CMP 指令中的左（目的）操作数和右（源）操 作数：

助记符	说明
JE	相等跳转 (leftOp=rightOp)

JNE	不相等跳转 (leftOp M rightOp)
JCXZ	CX=0 跳转
JECXZ	ECX=0 跳转
JRCXZ	RCX=0 跳转（64 位模式）

注：尽管 JE 指令相当于 JZ（为零跳转），JNE 指令相当于 JNZ（非零跳转），但是，最好是选择最能表明编程意图的助记符（JE 或 JZ），以便说明是比较两个操作数还是检查特定的状态标志位。

3）基于无符号操作数的比较跳转

基于无符号数比较的跳转如下表所示。操作数的名称反映了表达式中操作数的顺序（比如 leftOp < rightOp）。下表中的跳转仅在比较无符号数值时才有意义。有符号操作数使用不同的跳转指令。

助记符	说明	助记符	说明
JA	大于跳转（若 leftOp > rightOp）	JB	小于跳转（若 leftOp < rightOp）
JNBE	不小于或等于跳转（与 JA 相同）	JNAE	不大于或等于跳转（与 JB 相同）
JAE	大于或等于跳转（若 leftOp ≥ rightOp）	JBE	小于或等于跳转（若 leftOp ≤ rightOp）
JNB	不小于跳转（与 JAE 相同）	JNA	不大于跳转（与 JBE 相同）

4）基于有符号操作数的比较跳转

下表列出了基于有符号数比较的跳转。下面的指令序列展示了两个有符号数值的比较：

助记符	说明	助记符	说明
JG	大于跳转（若 leftOp > rightOp）	JL	小于跳转（若 leftOp < rightOp）
JNLE	不小于或等于跳转（与 JG 相同）	JNGE	不大于或等于跳转（与 JL 相同）
JGE	大于或等于跳转（若 leftOp ≥ rightOp）	JLE	小于或等于跳转（若 leftOp ≤ rightOp）
JNL	不小于跳转（与 JGE 相同）	JNG	不大于跳转（与 JLE 相同）

LOOP 相关指令

1）LOOP 指令

LOOP 指令，正式称为按照 ECX 计数器循环，将程序块重复特定次数。ECX 自动成为计数器，每循环一次计数值减 1。语法如下所示：

```
LOOP destination
```

循环目标必须距离当前地址计数器 -128 到 +127 字节范围内。LOOP 指令的执行有两个步骤：第一步，ECX 减 1；第二步，将 ECX 与 0 比较。

如果 ECX 不等于 0，则跳转到由目标给出的标号。否则，如果 ECX 等于 0，则不发生跳转，并将控制传递到循环后面的指令。

- 实地址模式中，CX 是 LOOP 指令的默认循环计数器。同时，LOOPD 指令使用 ECX 为循环计数器，LOOPW 指令使用 CX 为循环计数器。
- 64 位模式中，LOOP 指令用 RCX 作为循环计数器。

• 循环嵌套——当在一个循环中再创建一个循环时，就必须特别考虑外层循环的计数器 ECX，可以将它保存在一个变量中。作为一般规则，多于两重的循环嵌套难以编写。如果使用的算法需要多重循环，则将一些内层循环用子程序来实现。

2）LOOPZ 和 LOOPE 指令

• LOOPZ（为零跳转）指令的工作和 LOOP 指令相同，只是有一个附加条件：零标志位为 1 时控制转向目的标号。指令语法如下：

```
LOOPZ destination
```

• LOOPE（相等跳转）指令相当于 LOOPZ，它们有相同的操作码，只是为了清晰程序逻辑。这两条指令执行如下任务：

```
ECX = ECX - 1
if ECX > 0 and ZF = 1, jump to destination
```

否则，不发生跳转，并将控制传递到下一条指令。LOOPZ 和 LOOPE 不影响任何状态标志位。32 位模式下，ECX 是循环计数器；64 位模式下，RCX 是循环计数器。

3）LOOPNZ 和 LOOPNE 指令

• LOOPNZ（非零跳转）指令与 LOOPZ 相对应。当 ECX 中无符号数值大于零（减 1 操作之后）且零标志位等于零时，继续循环。指令语法如下：

```
LOOPNZ destination
```

• LOOPNE（不等跳转）指令相当于 LOOPNZ，它们有相同的操作码。这两条指令执行如 下任务：

```
ECX = ECX - 1
if ECX > 0 and ZF = 0, jump to destination
```

否则，不发生跳转，并将控制传递到下一条指令。

32 位条件控制流伪指令

32 位模式下，MASM 包含了一些高级条件控制流伪指令（conditional control flow directives），这有助于简化编写条件语句。遗憾的是，这些伪指令不能用于 64 位模式。

对程序进行汇编之前，汇编器执行的是预处理步骤。在这个步骤中，汇编器要识别伪指令，如：.CODE、.DATA，以及一些用于条件控制流的伪指令。下表列出了这些伪指令。

伪指令	说明
.BREAK	生成代码终止 .WHILE 或 .REPEAT 块
.CONTINUE	生成代码跳转到 .WHILE 或 .REPEAT 块的顶端
.ELSE	当 .IF 条件不满足时，开始执行的语句块
.ELSEIF condition	生成代码测试 condition，并执行其后的语句，直到碰到一个 .ENDIF 或另一个 .ELSEIF 伪指令
.ENDIF	终止 .IF、.ELSE 或 .ELSEIF 伪指令后面的语句块
.ENDW	终止 .WHILE 伪指令后面的语句块
.IF condition	如果 condition 为真，则生成代码执行语句块
.REPEAT	生成代码重复执行语句块，直到条件为真
.UNTIL condition	生成代码重复执行 .REPEAT 和 .UNTIL 伪指令之间的语句块，直到 condition 为真
.UNTILCXZ	生成代码重复执行 .REPEAT 和 .UNTILCXZ 伪指令之间的语句块，直到 CX 为零

.WHILE condition	当 condition 为真时，生成代码执行 .WHILE 和 .ENDW 伪指令之间的语句块

另外，MASM 也支持一些关系和逻辑运算符：

运算符	说明
expr1 == expr2	若 expr1 等于 expr2，则返回“真”
expr1 != expr2	若 expr1 不等于 expr2，则返回“真”
expr1 > expr2	若 expr1 大于 expr2，则返回"真”
expr1 ≥ expr2	若 expr1 大于等于 expr2，则返回“真”
expr1 < expr2	若 expr1 小于 expr2，则返回“真”
expr1 ≤ expr2	若 expr1 小于等于 expr2，则返回“真”
!expr1	若 expr 为假，则返回“真”
expr1expr2	对 expr1 和 expr2 执行逻辑 AND 运算
expr1 || expr2	对 1xpr1 和 expr2 执行逻辑 OR 运算
expr1 & expr2	对 expr1 和 expr2 执行按位 AND 运算
CARR1?	若进位标志位置 11 则返回“真”
OVERFLOW ?	若溢出标志位置 1，则返回“真”
PARITY ?	若奇偶标志位置 1，则返回“真”
SIGN ?	若符号标志位置 1，则返回“真”
ZERO ?	若零标志位置 1，则返回“真”

汇编语言整数运算

逻辑移位和算术移位

移动操作数的位有两种方法。第一种是逻辑移位 (logic shift)，空出来的位用 0 填充。另一种移位的方法是算术移位 (arithmetic shift)，空出来的位用原数据的符号位填充。

1）SHL 和 SHR 指令

• SHL（逻辑左移）指令使目的操作数逻辑左移一位，最低位用 0 填充。最高位移入进位标志位，而进位标志位中原来的数值被丢弃。

SHL 的第一个操作数是目的操作数，第二个操作数是移位次数：

```
SHL destination,count
```

该指令可用的操作数类型如下所示：

```
SHL reg, imm8
SHL mem, imm8
SHL reg, CL
SHL mem, CL
```

x86 处理器允许 imm8 为 0∽255 中的任何整数。另外，CL 寄存器包含的是移位计数。

上述格式同样适用于 SHR、SAL、SAR、ROR、ROL、RCR 和 RCL 指令。

• SHR（逻辑右移）指令使目的操作数逻辑右移一位，最高位用 0 填充。最低位复制到进位标志位，而进位标志位中原来的数值被丢弃。

2）SAL 和 SAR 指令

• SAL（算术左移）指令的操作与 SHL 指令一样。每次移动时，SAL 都将目的操作数中的每一位移动到下一个最高位上。最低位用 0 填充；最高位移入进位标志位，该标志位原来的值被丢弃。
• SAR（算术右移）指令将目的操作数进行算术右移，最高位由符号位（亦即其本身）补充。SAL 与 SAR 指令的操作数类型与 SHL 和 SHR 指令完全相同。移位可以重复执行，其次数由第二个操作数给出的计数器决定：

```
SAR destination, count
```

3）ROL 和 ROR 指令
以循环方式来移位即为位元循环（Bitwise Rotation）。一些操作中，从数的一端移出的位立即复制到该数的另一端。还有一种类型则是把进位标志位当作移动位的中间点。
• ROL（循环左移）指令把所有位都向左移。最高位复制到进位标志位和最低位。该指令格式与 SHL 指令相同。
• ROR（循环右移）指令把所有位都向右移，最低位复制到进位标志位和最高位。该指令格式与 SHL 指令相同。
※注：如果有符号数循环移动一位生成的结果超过了目的操作数的有符号数范围，则溢出标志位置 1。如果循环移动次数大于 1，则溢出标志位无定义。
4）RCL 和 RCR 指令
• RCL（带进位循环左移）指令把每一位都向左移，进位标志位复制到 LSB，而 MSB 复制到进位标志位。如果把进位标志位当作操作数最高位的附加位，那么 RCL 就成了循环左移操作。
• RCR（带进位循环右移）指令把每一位都向右移，进位标志位复制到 MSB，而 LSB 复制到进位标志位。RCL 指令将该整数转化成了一个 9 位值，进位标志位位于 LSB 的右边。
5）SHLD 和 SHRD 指令
• SHLD（双精度左移）指令将目的操作数向左移动指定位数。移动形成的空位由源操作数的高位填充。源操作数不变，但是符号标志位、零标志位、辅助进位标志位、奇偶标志位和进位标志位会受影响：

```
SHLD dest, source, count
```

• SHRD（双精度右移）指令将目的操作数向右移动指定位数。移动形成的空位由源操作数的低位填充：

```
SHRD dest, source, count
```

下面的指令格式既可以应用于 SHLD 也可以应用于 SHRD。目标操作数可以是寄存器或内存操作数；源操作数必须是寄存器；移位次数可以是 CL 寄存器或者 8 位立即数：

```
SHLD regl6, regl6, CL/imm8
SHLD meml6, regl6, CL/imm8
SHLD reg32, reg32, CL/imm8
SHLD mem32, reg32, CL/imm8
```

MUL 指令

32 位模式下，MUL（无符号数乘法）指令有三种类型：

第一种执行 8 位操作数与 AL 寄存器的乘法；
第二种执行 16 位操作数与 AX 寄存器的乘法；
第三种执行 32 位操作数与 EAX 寄存器的乘法。

乘数和被乘数的大小必须保持一致，乘积的大小则是它们的一倍。这三种类型都可以使用寄存器和内存操作数，但不能使用立即数：

```
MUL reg/mem8
```

MUL reg/mem16
MUL reg/mem32

MUL 指令中的单操作数是乘数。下表按照乘数的大小，列出了默认的被乘数和乘积。由于目的操作数是被乘数和乘数大小的两倍，因此不会发生溢出。

被乘数	乘数	乘积
AL	reg/mem8	AX
AX	reg/mem16	DX:AX
EAX	reg/mem32	EDX:EAX

如果乘积的高半部分不为零，则 MUL 会把进位标志位和溢出标志位置 1。有个很好的理由要求在执行 MUL 后检查进位标志位，即，确认忽略乘积的高半部分是否安全。

• 64 位模式下，MUL 指令可以使用 64 位操作数。一个 64 位寄存器或内存操作数与 RAX 相乘，产生的 128 位乘积存放到 RDX:RAX 寄存器中。

IMUL 指令

IMUL（有符号数乘法）指令执行有符号整数乘法。与 MUL 指令不同，IMUL 会保留乘 积的符号，实现的方法是，将乘积低半部分的最高位符号扩展到高半部分。

x86 指令集支持三种格式的 IMUL 指令：单操作数、双操作数和三操作数。单操作数格式中，乘数和被乘数大小相同，而乘积的大小是它们的两倍。

1）单操作数格式

单操作数格式将乘积存放在 AX、DX:AX 或 EDX:EAX 中：

```
IMUL reg/mem8     ; AX = AL * reg/mem8
IMUL reg/mem16     ; DX:AX = AX * reg/mem16
IMUL reg/mem32     ; EDX:EAX = EAX * reg/mem32
```

和 MUL 指令一样，其乘积的存储大小使得溢出不会发生。同时，如果乘积的高半部分不是其低半部分的符号扩展，则进位标志位和溢出标志位置 1。利用这个特点可以决定是否忽略乘积的高半部分。

2）双操作数格式（32 位模式）

32 位模式中的双操作数 IMUL 指令把乘积存放在第一个操作数中，这个操作数必须是寄存器。第二个操作数（乘数）可以是寄存器、内存操作数和立 即数。16 位格式如下所示：

```
IMUL reg16, reg/mem16
IMUL reg16, imm8
IMUL reg16, imm16
```

32 位操作数类型如下所示，乘数可以是 32 位寄存器、32 位内存操作数或立即数（8 位或 32 位）：

```
IMUL reg32, reg/mem32
IMUL reg32, inun8
IMUL reg32, imm32
```

双操作数格式会按照目的操作数的大小来截取乘积。如果被丢弃的是有效位，则溢出标志位和进位标志位置 1。因此，在执行了有两个操作数的 IMUL 操作后，必须检查这些标志位中的一个。

3）三操作数格式

32 位模式下的三操作数格式将乘积保存在第一个操作数中。第二个操作数可以是 16 位寄存器或内存操作数，它与第三个操作数相乘，该操作数是一个 8 位或 16 位立即数：

```
IMUL reg16, reg/mem16,imm8
```

IMUL reg16, reg/mem16, iirrn16
而 32 位寄存器或内存操作数可以与 8 位或 32 位立即数相乘：
IMUL reg32, reg/mem32, imm8
IMUL reg32, reg/mem32, imm32
IMUL 执行时，若乘积有效位被丢弃，则溢出标志位和进位标志位置 1。因此，在执行了有三个操作数的 IMUL 操作后，必须检查这些标志位中的一个。
4）64 位模式
在 64 位模式下，IMUL 指令可以使用 64 位操作数。在单操作数格式中，64 位寄存器或内存操作数与 RAX 相乘，产生一个 128 位且符号扩展的乘积存放到 RDX:RAX 寄存器中。三操作数格式也可以用于 64 位模式。

DIV 指令

32 位模式下，DIV（无符号除法）指令执行 8 位、16 位和 32 位无符号数除法。其中，单寄存器或内存操作数是除数。格式如下：

```
DIV reg/mem8
DIV reg/mem16
DIV reg/mem32
```

下表给出了被除数、除数、商和余数之间的关系：

被除数	除数	商	余数
AX	reg/mem8	AL	AH
DX:AX	reg/mem16	AX	DX
EDX:EAX	reg/mem32	EAX	EDX

64 位模式下，DIV 指令用 RDX:RAX 作被除数，用 64 位寄存器和内存操作数作除数，商存放到 RAX，余数存放在 RDX 中。

IDIV 指令

IDIV（有符号除法）指令执行有符号整数除法，其操作数与 DIV 指令相同。只有一个重要的区别：在执行除法之前，必须对被除数进行符号扩展。符号扩展是指将一个数的最高位复制到包含该数的变量或寄存器的所有高位中。余数的符号总是与被除数相同。

符号扩展指令

Intel 提供了三种符号扩展指令：CBW、CWD 和 CDQ。CBW（字节转字）指令将 AL 的符号位扩展到 AH，保留了数据的符号。
CWD（字转双字）指令将 AX 的符号位扩展到 DX。
CDQ（双字转四字）指令将 EAX 的符号位扩展到 EDX。

ADC 和 SBB 指令

• ADC（带进位加法）指令将源操作数和进位标志位的值都与目的操作数相加。该指令格式与 ADD 指令一样，且操作数大小必须相同：

```
ADC reg, reg
ADC mem, reg
ADC reg, mem
ADC mem, imm
ADC reg, imm
```

• SBB（带借位减法）指令从目的操作数中减去源操作数和进位标志位的值。允许使用的操作数与 ADC 指令相同。

汇编语言 ASCII 和非压缩十进制运算

非压缩十进制数是用每个字节表示一个十进制数位，因此每个字节的高 4 位总是 0。

1）AAA 指令

在 32 位模式下，AAA（ 加法后的 ASCII 调整 ）指令调整 ADD 或 ADC 指令的二进制运算结果。设两个 ASCII 数字相加，其二进制结果存放在 AL 中，则 AAA 将 AL 转换为两个非压缩十进制数字存入 AH 和 AL。一旦成为非压缩格式，通过将 AH 和 AL 与 30h 进行 OR 运算，很容易就能把它们转换为 ASCII 码。

2）AAS 指令

32 位模式下，AAS（减法后的 ASCII 调整）指令紧随 SUB 或 SBB 指令之后，这两条指令执行两个非压缩十进制数的减法，并将结果保存到 AL 中。AAS 指令将 AL 转换为 ASCII 码的数字形式。只有减法结果为负时，调整才是必需的。

3）AAM 指令

32 位模式下，MUL 执行非压缩十进制乘法，AAM（乘法后的 ASCII 调整）指令转换由其产生的二进制乘积。乘法只能使用非压缩十进制数。

4）AAD 指令

32 位模式下，AAD（除法之前的 ASCII 调整）指令将 AX 中的非压缩十进制被除数转换为二进制，为执行 DIV 指令做准备。

压缩十进制运算

压缩十进制数的每个字节存放两个十进制数字，每个数字用 4 位表示。由于数位个数必须为偶数，如果有效数字个数为奇数，则最高的半字节用零填充。

压缩十进制存储至少有两个优势：

数据几乎可以包含任何个数的有效数字。这使得以很高的精度执行计算成为可能。

实现压缩十进制数与 ASCII 码之间的相互转换相对简单。

1）DAA 指令

32 位模式下，ADD 或 ADC 指令在 AL 中生成二进制和数，DAA（加法后的十进制调整）指令将和数转换为压缩十进制格式。

2）DAS 指令

32 位模式下，SUB 或 SBB 指令在 AL 中生成二进制结果，DAS（减法后的十进制调整）指令将其转换为压缩十进制格式。

具体调整规则如下：

如果 AL 的低四位大于 9，或 AF=1，那么，AL=AL-06H，并置 AF=1；

如果 AL 的高四位大于 9，或 CF=1，那么，AL=AL-60H，并置 CF=1；

如果以上两点都不成立，则清除标志位 AF 和 CF。

经过调整后，AL 的值仍是压缩型 BCD 码，即：二个压缩型 BCD 码相减，并进行调整后，得到的结果还是压缩型 BCD 码。

汇编语言过程

PUSH 和 POP 及相关指令

• PUSH 指令首先减少 ESP 的值，再将源操作数复制到堆栈。操作数是 16 位的，则 ESP 减 2，操作数是 32 位的，则 ESP 减 4。操作数可以是寄存器、内存或立即数。
• POP 指令首先把 ESP 指向的堆栈元素内容复制到一个 16 位或 32 位目的操作数中，再增加 ESP 的值。如果操作数是 16 位的，ESP 加 2，如果操作数是 32 位的，ESP 加 4。操作数不能是立即数。
• PUSHFD 指令把 32 位 EFLAGS 寄存器内容压入堆栈，而 POPFD 指令则把栈顶单元内容弹出到 EFLAGS 寄存器。
• PUSHAD 指令按照 EAX、ECX、EDX、EBX、ESP（执行 PUSHAD 之前的值）、EBP、ESI 和 EDI 的顺序，将所有 32 位通用寄存器压入堆栈。POPAD 指令按照相反顺序将同样的寄存器弹出堆栈。
• PUSHA 指令按序（AX、CX、DX、BX、SP、BP、SI 和 DI）将 16 位通用寄存器压入堆栈。POPA 指令按照相反顺序将同样的寄存器弹出堆栈。在 16 位模式下，只能使用 PUSHA 和 POPA 指令。

CALL 和 RET 指令

CALL 指令调用一个过程，指挥处理器从新的内存地址开始执行。过程使用 RET（从过程返回）指令将处理器转回到该过程被调用的程序点上。
从物理上来说，CALL 指令将其返回地址压入堆栈，再把被调用过程的地址复制到指令指针寄存器。当过程准备返回时，它的 RET 指令从堆栈把返回地址弹回到指令指针寄存器。

```
CALL procedure_name
RET
```

USES 运算符

USES 运算符与 PROC 伪指令一起使用，让程序员列出在该过程中修改的所有寄存器名。USES 告诉汇编器做两件事情：第一，在过程开始时生成 PUSH 指令，将寄存器保存到堆栈；第二，在过程结束时生成 POP 指令，从堆栈恢复寄存器的值。实际上就是一个指导编译代码生成的伪指令。
USES 运算符紧跟在 PROC 之后，其后是位于同一行上的寄存器列表，表项之间用空格符或制表符（不是逗号）分隔。

```
procedure_name PROC USES reg1 [reg2] […]
```

汇编语言高级过程

调用规范

调用规范主要分两种：寄存器传递参数和堆栈传递参数。在 32 位模式下，堆栈参数总是由 Windows API 函数使用。然而在 64 位模式下，Windows 函数可以同时接收寄存器参数和堆栈参数。
寄存器传递参数理论上比较快捷，但是经常需要在调用开始前备份寄存器原始内容和返回前恢复寄存器内容的情况下，会使得寄存器传递参数非常不方便且性能优势不大。
1）x64 调用规范（寄存器参数）

Microsoft 在 64 位程序中使用统一模式来传递参数并调用过程，称为 Microsoft x64 调用规范。该规范由 C/C++ 编译器和 Windows 应用编程接口（API）使用。
程序员只有在调用 Windows API 的函数或用 C/C++ 编写的函数时，才会使用这个调用规范。该调用规范的一些基本特性如下所示：
• 前四个参数依序存入 RCX、RDX、R8 和 R9 寄存器，并传递给过程。如果只有一个参数，则将其放入 RCX。如果还有第二个参数，则将其放入 RDX，以此类推。其他参数，按照从左到右的顺序压入堆栈。长度不足 64 位的参数不进行零扩展，因此，其高位的值是不确定的。
• 寄存器 RAX、RCX、RDX、R8、R9、R10 和 R11 常常被子程序修改，因此，如果主调程序想要保存它们的值，就应在调用子程序之前将它们入栈，之后再从堆栈弹出。
• 寄存器 RBX、RBP、RDI、RSI、R12、R13、R14 和 R15 的值必须由子程序保存。
• 调用者的责任还包括在运行时堆栈分配至少 32 字节的影子空间（shadow space），这样，被调用的过程就可以选择将寄存器参数保存在这个区域中。
• 在调用子程序时，堆栈指针（RSP）必须进行 16 字节边界对齐（16 的倍数）。因为 CALL 指令将压入返回地址，并将 RSP（堆栈指针）寄存器减 8（因为地址是 64 位的），因此除了已经减去的影子空间的 32 之外，调用程序还必须从堆栈指针中减去 8。
• 被调用子程序执行结束后，主调程序需负责从运行时堆栈中移除所有的参数和影子空间。
• 如果返回值的长度小于或等于 64 位，那么它必须放在 RAX 寄存器中。大于 64 位的返回值存放于运行时堆栈，由 RCX 指出其位置。
2）x32 调用规范（堆栈参数）
——调用者：
• 按照参数列表逆序压入参数，被调用者可以用[EBP + n]访问
• CALL 指令压入被调用者要返回至的指令地址，将被调用者指令首地址赋给指 EIP
——被调用者：
• 压入原过程帧基址 EBP，并更新 EBP 值为当前 ESP 值
• 为局部变量保留空间，可以用[EBP - n]访问
• 压入寄存器的值以备份
注：最后两步根据实际情况都是可选的，且顺序可能在不同的具体规范中不固定，但是原则上一定是可以确定局部变量或寄存器备份的占用范围，进而可以使用 EBP 偏移量访问。
①C 调用规范
C 调用规范用于 C 和 C++ 语言。子程序的参数按逆序入栈。程序调用子程序时，在 CALL 指令的后面紧跟一条语句使堆栈指针（ESP）加上一个数，该数的值即为子程序参数所占堆栈空间的总和。因此，用 C/C++ 编写的程序在从子程序返回后，总是能把参数从堆栈中删除。
• C 语言说明符在外部过程名的前面添加前导下划线，如：_name。
②STDCALL 调用规范
给 RET 指令添加了一个整数参数，这使得程序在返回到调用过程时，ESP 会加上该整数。这个添加的整数必须与被调用过程参数占用的堆栈空间字节数相等。STDCALL 与 C 相似，参数是按逆序入栈的。通过在 RET 指令中添加参数，STDCALL 不仅减少了子程序调用产生的代码量（减少了一条指令），还保证了调用程序永远不会忘记清除堆栈。
• STDCALL 通过将输出（公共）过程名保存为如下格式来修改这些名称，并送给链接器，如_name@nn。前导下划线添加到过程名，@ 符号后面的整数指定了过程参数的字节数（向上舍入到 4 的倍数）。
③二者比较

C 调用规范则允许子程序声明不同数量的参数，主调程序可以决定传递多少个参数。C 编译器按逆序将参数入栈，被调用的函数负责确定要传递的实际参数的个数，然后依次访问参数。这种函数实现没有像给 RET 指令添加一个常数那样简便的方法来清除堆栈，因此，这个责任就留给了主调程序采用 C 调用规范解决。

LEA 指令

LEA 指令返回间接操作数的地址。由于间接操作数中包含一个或多个寄存器，因此会在运行时计算这些操作数的偏移量。如：

```
LEA reg, [reg + n]
```

• 显然 LEA 指令可以被其他方式替代，但是不包括 OFFSET，后者只适用于数据区标号

ENTER 和 LEAVE 指令

• ENTER 指令为被调用过程自动创建堆栈帧。它为局部变量保留堆栈空间，把 EBP 入栈。具体来说，它执行三个操作：

```
把 EBP 入栈 (push ebp)
把 EBP 设置为堆栈帧的基址 (mov ebp, esp)
为局部变量保留空间 (sub esp, numbytes)
```

ENTER 有两个操作数：第一个是常数，定义为局部变量保存的堆栈空间字节数；第二个定义了过程的词法嵌套级。

```
ENTER numbytes, nestinglevel
```

这两个操作数都是立即数。Numbytes 总是向上舍入为 4 的倍数，以便 ESP 对齐双字边界。Nestinglevel 确定了从主调过程堆栈帧复制到当前帧的堆栈帧指针的个数。

• LEAVE 指令结束一个过程的堆栈帧。它反转了之前的 ENTER 指令操作：恢复了过程被调用时 ESP 和 EBP 的值。

• 注意：这两个指令只用于被调用者内部，不负责参数传递和释放堆栈帧参数的任务。

LOCAL 伪指令

Microsoft 创建 LOCAL 伪指令是作为 ENTER 指令的高级替补。LOCAL 声明一个或多个变量名，并定义其大小属性。（另一方面，ENTER 则只为局部变量保留一块未命名的堆栈空间。）如果要使用 LOCAL 伪指令，它必须紧跟在 PROC 伪指令的后面。

其语法如下所示：

```
LOCAL varlist
```

varlist 是变量定义列表，用逗号分隔表项，可选为跨越多行。每个变量定义采用如下格式：

```
label:type
label[n]:type ;数组声明
```

其中，标号可以为任意有效标识符，类型既可以是标准类型（WORD、DWORD 等），也可以是用户定义类型。

• 在声明不同大小的局部变量时，LOCAL 为每个变量都按照其大小来分配空间：8 位的变量分配给下一个可用的字节，16 位的变量分配给下一个偶地址（字对齐），32 位变量分配给下一个双字对齐的地址。若最后一个局部变量为 8 位，则 ESP 减小 4 字节，而该局部变量只占用最高字节。

INVOKE 伪指令

INVOKE 伪指令，只用于 32 位模式，将参数入栈（按照 .MODEL 伪指令的语言说明符所指定的顺序）并调用过程。INVOKE 是 CALL 指令一个方便的替代品，因为，它用一行代码就能传递多个参数。常见语法如下：

```
INVOKE procedureName [, argumentList]
```

ArgumentList 是可选项，它用逗号分隔传递给过程的参数。

• 覆盖 EAX 和 EDX

如果向过程传递的参数小于 32 位，那么在将参数入栈之前，INVOKE 为了扩展参数常常会使得汇编器覆盖 EAX 和 EDX 的内容。有两种方法可以避免这种情况：

其一，传递给 INVOKE 的参数总是 32 位的；

其二，在过程调用之前保存 EAX 和 EDX，在过程调用之后再恢复它们的值。

ADDR 运算符

ADDR 运算符同样可用于 32 位模式，在使用 INVOKE 调用过程时，它可以传递指针参数。比如，下面的 INVOKE 语句给 FillArray 过程传递了 myArray 的地址：

```
INVOKE FillArray, ADDR myArray
```

传递给 ADDR 的参数必须是汇编时常数，就像 OFFSET 的要求一样，但是 ADDR 运算符只能与 INVOKE 一起使用。

• OFFSET 似乎可以替代 ADDR，但是二者都是伪指令，可能因汇编器而异。

PROC 和 ENDP 伪指令

汇编语言中，所有在当前文件中使用的过程（无论是自定义、库函数）都必须在文件开始处声明。32 位模式中，PROC 与 ENDP 伪指令用于定义子程序（也可以声明同时定义），其基本语法如下所示（主函数无需 RET）：

```
label PROC (attributes) (USES reglist), parameter_list
        （子程序内容）
label ENDP
```

• label 是由用户定义的标号。

• attributes 是指下述任一内容：

```
[distance] [langtype] [visibility] [prologuearg]
```

下表对这些属性进行了说明。

属性	说明
distance	NEAR 或 FAR。指定汇编器生成的 RET 指令（RET 或 RETF）类型
langtype	指定调用规范（参数传递规范），如 C、PASCAL 或 STDCALL。能覆盖由 .MODEL 伪指令指定的规范
visibility	指明本过程对其他模块的可见性。选项包括 PRIVATE、PUBLIC （默认项）和 EXPORT。若可见性为 EXPORT，则链接器把过程名放入分段可执行文件的导出表。EXPORT 也使之具有了 PUBLIC 可见性
prologuearg	指定会影响开始和结尾代码生成的参数

• parameter_list

PROC 伪指令允许在声明过程时，添加上用逗号分隔的参数名列表。代码实现可以用名称来引用参数，而不是计算堆栈偏移量。

每个参数的语法如下：

```
paramName: type
```

ParamName 是分配给参数的任意名称，其范围只限于当前过程（称为局部作用域（local scope））。同样的参数名可以用于多个过程，但却不能作为全局变量或代码标号的名称。
Type 可以在这些类型中选择：BYTE、SBYTE、WORD、SWORD、DWORD、SDWORD、FWORD、QWORD 或 TBYTE。此外，type 还可以是限定类型（qualified type），如指向现有类型的指针。
• 当 PROC 有一个或多个参数时，STDCALL 是默认调用规范。
1）隐藏和导出过程名
默认情况下，MASM 使所有的过程都是 public 属性，即允许它们能被同一程序中任何其他模块调用。使用限定词 PRIVATE 可以覆盖这个属性：

```
mySub PROC PRIVATE
```

使过程为 private 属性，可以利用封装原则将过程隐藏在模块中，如果其他模块有相同过程名，就还需避免潜在的重名冲突。
• 变量名和符号名的导出和隐藏方式为

```
PUBLIC var1,var2…
```

默认情况下，变量和符号对其包含模块是私有的（private）。
2）OPTION PROC:PRIVATE 伪指令
在源模块中隐藏过程的另一个方法是，把 OPTION PROC:PRIVATE 伪指令放在文件顶部。则所有的过程都默认为 private，然后用 PUBLIC 伪指令指明那些希望其可见的过程：

```
OPTION PROC:PRIVATE
PUBLIC mySub
```

PUBLIC 伪指令用逗号分隔过程名：

```
PUBLIC sub1, sub2, sub3
```

如果程序的启动模块使用了 OPTION PROC:PRIVATE，那么就应该将它（通常为 main）指定为 PUBLIC，否则操作系统加载器无法发现该启动模块。

PROTO 伪指令

用于声明子程序，不是定义。32 位模式下，PROTO 的用法格式类似 PROC。64 位模式下，PROTO 之后不能带有任何内容。有关于过程声明（原型）、过程调用、过程实现（定义）的规则和高级语言一致。
• 参数检查（MASM）：
MASM 会检测实际参数超过形式参数大小的错误，还有参数个数不匹配的错误。但是不会检测实参小于形参的情况，并会自动扩展实参。而且一如汇编语言的特点，也不会做指针和数据类型的检测。

EXTERN 伪指令

1）访问外部过程
调用当前模块之外的过程时使用 EXTERN 伪指令，它确定过程名和堆栈帧大小。

```
EXTERN procedure_name@n:PROC
```

当汇编器在源文件中发现一个缺失的过程时（由 CALL 指令指定），默认情况下它会产生错误消息。但是，EXTERN 伪指令告诉汇编器为该过程新建一个空地址。在链接器生成程序的可执行文件时再来确定这个空地址。
过程名的后缀 @n 确定了已声明参数占用的堆栈空间总量。如果使用的是基本 PROC 伪指令，没有声明参数，那么 EXTERN 中的每个过程名后缀都为 @0。若用扩展 PROC 伪指令声明一个过程，则每个参数占用 4 字节。

• 注意：在新文件中，过程名已经变为 procedure_name@n。
• 或者，也可以用 PROTO 伪指令声明来代替 EXTERN。
2）访问外部变量和符号
使用 EXTERN 伪指令可以访问在外部过程中定义的变量和符号：

```
EXTERN name:type
```

对符号（由 EQU 和 = 定义）而言，type 应为 ABS。对变量而言，type 是数据定义属性，如 BYTE、WORD、DWORD 和 SDWORD，可以包含 PTR。
3）EXTERNDEF 和 INCLUDE 文件
MASM 中一个很有用的伪指令 EXTERNDEF 可以代替 PUBLIC 和 EXTERN。它可以放在文本文件中，并用 INCLUDE 伪指令复制到每个程序模块。

Java 虚拟机 JVM 工作原理

• 字节码：实为 JVM 使用的汇编语言的二进制形式，每个指令一般只占 1 字节空间，且只执行一个操作。在运行时该汇编程序可以解释翻译为对应平台机器指令逐条执行，也成为实时编译 just-in-time compilation。
• 基于堆栈指令集：JVM 的汇编指令集使用堆栈实现数据传送、算术运算、比较和分支操作，而不是用寄存器和内存操作数。JVM 使用操作数栈进行工作，操作数区实际位于堆栈顶端，压入这个区域的数值可以作为算术和逻辑运算的操作数，设置局部变量，以及传递给类方法的参数。在局部变量被算术运算指令或比较指令使用之前，它们必须被压入堆栈帧的操作数区域（此时局部变量在整个堆栈中相当于有两个副本）。在运算结束后，其结果也会被压入栈中。这种基于堆栈的执行方式贯穿了从基本算术运算到方法调用的所有行为。
注：尽管 JVM 使用不同于 x86 的指令集，但是在 x86 平台上，Java 程序最终还是会转化为 x86 指令集结构。这也是 JVM 虚拟机的含义。

汇编语言字符串和数组

字符串基本指令

1）MOVSB、MOVSW 和 MOVSD 指令
MOVSB、MOVSW 和 MOVSD 指令将数据从 ESI 指向的内存位置复制到 EDI 指向的内存位置。（根据方向标志位的值）这两个寄存器自动地增加或减少：

MOVSB	传送（复制）字节
MOVSW	传送（复制）字
MOVSD	传送（复制）双字

MOVSB、MOVSW 和 MOVSD 可以使用重复前缀。方向标志位决定 ESI 和 EDI 是否增加或减少。
2）CMPSB、CMPSW 和 CMPSD 指令
CMPSB、CMPSW 和 CMPSD 指令比较 ESI 指向的内存操作数与 EDI 指向的内存操作数：

CMPSB	比较字节
CMPSW	比较字
CMPSD	比较双字

CMPSB、CMPSW 和 CMPSD 可以使用重复前缀。方向标志位决定 ESI 和 EDI 的增加或减少。
3）SCASB、SCASW 和 SCASD 指令

SCASB、SCASW 和 SCASD 指令分别将 AL/AX/EAX 中的值与 EDI 寻址的一个字节 / 字 / 双字进行比较。这些指令可用于在字符串或数组中寻找一个数值。结合 REPE（或 REPZ）前缀，当 ECX > 0 且 AL/AX/EAX 的值等于内存中每个连续的值时，不断扫描字符串或数组。
REPNE 前缀也能实现扫描，直到 AL/AX/EAX 与某个内存数值相等或者 ECX = 0。
4）STOSB、STOSW 和 STOSD 指令
STOSB、STOSW 和 STOSD 指令分别将 AL/AX/EAX 的内容存入由 EDI 中偏移量指向的内存位置。EDI 根据方向标志位的状态递增或递减。
与 REP 前缀组合使用时，这些指令实现用同一个值填充字符串或数组的全部元素。
5）LODSB、LODSW 和 LODSD 指令
LODSB、LODSW 和 LODSD 指令分别从 ESI 指向的内存地址加载一个字节或一个字到 AL/AX/EAX。ESI 按照方向标志位的状态递增或递减。
LODS 很少与 REP 前缀一起使用，原因是，加载到累加器的新值会覆盖其原来的内容。相对而言，LODS 常常被用于加载单个数值。
6）重复前缀
就其自身而言，字符串基本指令只能处理一个或一对内存数值。如果加上重复前缀，指令就可以用 ECX 作计数器重复执行。重复前缀使得单条指令能够处理整个数组。下面为可用的重复前缀：

REP	ECX > 0 时重复
REPZ、REPE	零标志位置 1 且 ECX > 0 时重复
REPNZ、REPNE	零标志位清零且 ECX > 0 时重复

7）方向标志位
根据方向标志位的状态，字符串基本青令增加或减少 ESI 和 EDI 如下表所示。可以用 CLD 和 STD 指令显式修改方向标志位：
CLD ;方向标志位清零（正向）
STD ;方向标志位置 1（反向）

方向标志位的值	对 ESI 和 EDI 的影响	地址顺序
0	增加	低到高
1	减少	高到低

汇编语言结构和宏
结构体
1）STRUCT 和 ENDS 伪指令
定义结构使用的是 STRUCT 和 ENDS 伪指令。在结构内，定义字段的语法与一般的变量定义是相同的。结构对其包含字段的数量几乎没有任何限制：

```
name STRUCT
    field-declarations
name ENDS
```

字段初始值若结构字段有初始值，那么在创建结构变量时就要进行赋值。字段初始值可以使用各种类型：
①无定义：运算符？使字段初始值为无定义。
②字符串文本：用引号括起的字符串。
③整数：整数常数和整数表达式。

④数组：DUP 运算符可以初始化数组元素。
为了获得最好的内存 I/O 性能，结构成员应按其数据类型进行地址对齐。
2）声明结构变量
• 结构变量可以被声明，并能选择为是否用特定值进行初始化。语法如下，
identifier structureType <initializer-list>
identifier 的命名规则与 MASM 中其他变量的规则相同。structureType 是已定义的结构体类型。initializer-list 为可选项，但是如果选择使用，则该项就是一个用逗号分隔的汇编时常数列表，需要与特定结构字段的数据类型相匹配（<>可以用{}代替）：
<initializer [, initializer] ...>
• 空括号 <> 使结构包含的是结构定义的默认字段值。此外，还可以在选定字段中插入新值。结构字段中的插入值顺序为从左到右，与结构声明中字段的顺序一致。若字符串字段初始值的长度少于字段的定义，则多出的位置用空格填充。空字节不会自动插到字符串字段的尾部。通过插入逗号作为位置标记可以跳过结构字段。
• DUP 可以用于声明结构体数组，如：
identifier structureType numOfElements DUP (<initializer-list>)
• 为了最好的处理器性能，结构变量在内存中的位置要与其最大结构成员的边界对齐。注意这里的结构体变量对齐和结构体定义时的内部成员对齐的区别。
• 结构体类型中也可以声明结构变量作为其成员，语法和一般声明结构变量无异。
3）引用成员
结构体类型和变量名都可以用点号“.”访问成员。当然仅在无需实例化的情况下使用类型名访问成员才合法，如使用 TYPE 等运算符计算大小。

联合 union

1）声明定义
• 结构中的每个字段都有相对于结构第一个字节的偏移量，而联合 (union) 中所有的字段则都起始于同一个偏移量。一个联合的存储大小即为其最大字段的长度。如果不是结构的组成部分，那么需要用 UNION 和 ENDS 伪指令来定义联合：

```
unionname UNION
    union-fields
unionname ENDS
```

• 如果联合嵌套在结构内，其语法会有一点不同：

```
structname STRUCT
    structure-fields
    UNION unionname
        union-fields
    ENDS
structname ENDS
```

除了其所有字段都必须使用同一个初始值外（如果在多个字段中声明多个不同的值，那第一个字段的初始值为默认，其他忽略），联合字段声明的规则与结构的规则相同。
2）声明变量
联合变量的声明和初始化方法与结构变量相同，只除了一个重要的差异：不允许初始值多于一个，可以采用默认值。

宏过程简述

宏过程 macro procedure 是一种没有返回值的宏，也称内联展开 inline expansion。另有一种宏函数 macro function 有返回值，通常所说的宏 macro 指前者。

1）定义

定义一个宏使用的是 MACRO 和 ENDM 伪指令，其语法如下所示：

```
macroname MACRO parameter-1, parameter-2...
    statement-list
ENDM
```

关于缩进没有硬性规定，但是还是建议对 macroname 和 ENDM 之间的语句进行缩进。 同时，还希望在宏名上使用前缀 m，形成易识别的名称，如 mPutChar，mWriteString 和 mGotoxy。

除非宏被调用，否则 MACRO 和 ENDM 伪指令之间的语句不会被汇编。宏定义中还可以有多个形参，参数之间用逗号隔开。

2）调用宏

调用宏的方法是把宏名插入到程序中，后面可能跟有宏的实参。宏调用语法如下：

```
macroname argument-1, argument-2,...
```

Macroname 必须是源代码中在此之前被定义宏的名称。每个实参都是文本值，用以替换宏的一个形参。实参的顺序要与形参一致，但是两者的数量不须相同。如果传递的实参数太多，则汇编器会发出警告。如果传递给宏的实参数太少，则未填充的形参保持为空。

3）其他特性

①规定形参

利用 REQ 限定符，可以指定必需的宏形参。如果被调用的宏没有实参与规定形参相匹配，那么汇编器将显示出错消息。如果一个宏有多个规定形参，则每个形参都要使用 REQ 限定符。下面是宏 mPutChar，形参 char 是必需的：

```
mPutchar MACRO char:REQ
```

②宏注释

宏定义中的注释行一般都出现在每次宏展开的时候。如果希望忽略宏展开时的注释，就在它们的前面添加双分号 (;;)。

③ECHO 伪指令

用于写一个字符串到标准输出，如：

```
ECHO xxxxxx
```

④LOCAL 伪指令

宏定义中常常包含了标号，并会在其代码中对这些标号进行自引用。由于汇编器不允许两个标号有相同的名字，因此若多次调用内部带有定义标号的宏，其结果会出现错误。同时，循环或重复语句中的标号也会有类似问题。

为了避免标号重命名带来的问题，可以对一个宏定义内的标号使用 LOCAL 伪指令。若标号被标记为 LOCAL，那么每次进行宏展开时，预处理程序就把标号名转换为唯一的标识符。如：

```
LOCAL xxx
```

汇编器生成的标号名使用了 ??nnnn 的形式，其中 nnnn 是具有唯一性的整数。

⑤默认参数初始值

宏可以有默认参数初始值。如果调用宏出现了宏参数缺失，那么就可以使用默认参数。其语法如下：

```
macroName MACRO paramName := < argument >
```

运算符前后的空格是可选的。

条件汇编伪指令简述

很多不同的条件汇编伪指令都可以和宏一起使用，这使得宏更加灵活。条件汇编伪指令常用语法如下所示，不同的伪指令语法结构类似，不同之处仅在于条件判断伪指令的逻辑。

```
IF condition
     statements
[ELSE
     statements]
ENDIF
```

下表列出了更多常用的条件汇编伪指令。若说明为该伪指令允许汇编，就意味着所有的后续语句都将被汇编，直到遇到下一个 ELSE 或 ENDIF 伪指令。必须强调的是，表中列出的伪指令是在汇编时而不是运行时计算。

伪指令	说明
IF expression	若 expression 为真（非零）则允许汇编。可能的关系运算符为 LT、GT、EQ、NE、LE 和 GE
IFB<argument>	若 argument 为空则允许汇编。实参名必须用尖括号（＜＞）括起来
IFNB<argument>	若 argument 为非空则允许汇编。实参名必须用尖括号（＜＞）括起来
IFIDN<arg1>,<arg2>	若两个实参相等（相同）则允许汇编。采用区分大小写的比较
IFIDNI<arg1 >,<arg2>	若两个实参相等（相同）则允许汇编。采用不区分大小写的比较
IFDIF<arg1 >,<arg2>	若两个实参不相等则允许汇编。采用区分大小写的比较
IFDIFI<arg1>,<arg2>	若两个实参不相等则允许汇编。采用不区分大小写的比较
IFDIF name	若 name 已定义则允许汇编
IFNDEF name	若 name 还未定义则允许汇编
ENDIF	结束用一个条件汇编伪指令开始的代码块
ELSE	若条件为真，则终止汇编之前的语句。若条件为假，ELSE 汇编语句直到遇到下一个 ENDIF
ELSEIF expression	若之前条件伪指令指定的条件为假，而当前表达式为真，则汇编全部语句直到出现 ENDIF
EXITM	立即退出宏，阻止所有后续宏语句的展开

• IF 伪指令的后面必须跟一个常量布尔表达式。该表达式可以包含整数常量、符号常量或者常量宏实参，但不能包含寄存器或变量名。

宏汇编运算符简介

1）替换运算符&

替换运算符（&）解析对宏参数名的有歧义的引用。比如若宏参数名出现在字符串文本“xxx”之中，一般不会被汇编器识别并替换，但若在其之前加&符号，则强制要求汇编器执行替换。

2）展开运算符%

展开运算符（%）有多个功能，可展开文本宏、将常量表达式计算后返回或转换为文本形式返回。

• 若使用的是 TEXTEQU，% 运算符就计算常量表达式，再把结果转换为文本。如

identifier TEXTEQU %(expr)
• 当展开运算符 (%) 是一行源代码的第一个字符时，它指示预处理程序展开该行上的所有文本宏和宏函数。尤其是和 ECHO 合用时，可以达到灵活输出的效果。
3）文字文本运算符<>
文字文本（literal-text）运算符（<>）把一个或多个字符和符号组合成一个文字文本，以防止预处理程序把列表中的成员解释为独立的参数。亦即，如果用作单个参数的文本内容中包含逗号等产生歧义的字符时，可以用<>包围全部内容以消歧义。这种歧义也是因为宏参数无类型及检查所致。
4）文字字符运算符!
构造文字字符（literal-character）运算符（!）的目的与文字文本运算符的几乎完全一样：强制预处理程序把预先定义的运算符当作普通的字符。如!>即可将>解析为普通字符（大于号）而不会和<>混淆。

宏函数

宏函数与宏过程有相似的地方，它也为汇编语言语句列表分配一个名称，声明方法和语法基本和宏过程完全一致。不同的地方在于，宏函数通过 EXITM 伪指令总是返回一个常量（整数或字符串），如 EXITM <xxx>。宏函数经常返回 0 和非 0 值作为逻辑条件判断和条件编译共用。
1）调用宏函数
调用宏函数时，它的实参列表必须用括号括起来。如 macroFunctionName(paraList)。

重复语句伪指令

MASM 有许多循环伪指令用于生成重复的语句块：WHILE、REPEAT、FOR 和 FORC。与 LOOP 指令不同，这些伪指令只在汇编时起作用，并使用常量值作为循环条件和计数器，从它们都有 ENDM 作为结尾可以看出其本质。
1）WHILE 伪指令
WHILE 伪指令重复一个语句块，直到特定的常量表达式为真。其语法如下：

```
WHILE constExpression
    statements
ENDM
```

2）REPEAT 伪指令
在汇编时，REPEAT 伪指令将一个语句块重复固定次数。其语法如下：

```
REPEAT constExpression
    statements
ENDM
```

constExpression 是一个无符号整数常量表达式，用于确定重复次数。
3）FOR 伪指令
FOR 伪指令通过迭代用逗号分隔的符号列表来重复一个语句块。列表中的每个符号都会引发循环的一次迭代过程。其语法如下：

```
FOR parameter,<arg1,arg2,arg3,...>
    statements
ENDM
```

第一次循环迭代时，parameter 取 arg1 的值，第二次循环迭代时，parameter 取 arg2 的值；以此类推，直到列表的最后一个实参。
4）FORC 伪指令

FORC 伪指令通过迭代字符串来重复一个语句块。字符串中的每个字符都会引发循环的一次迭代过程。其语法如下：

```
FORC parameter, <string>
    statements
ENDM
```

第一次循环迭代时，parameter 等于字符串的第一个字符，第二次循环迭代时，parameter 等于字符串的第二个字符；以此类推，直到最后一个字符。

浮点数处理与指令编码

FPU 浮点数计算单元简介

• FPU 不使用通用寄存器 (EAX、EBX 等等)。反之，它有自己的一组寄存器，称为寄存器栈 (register stack)。数值从内存加载到寄存器栈，然后执行计算，再将堆栈数值保存到内存。

• FPU 指令用后缀 (postfix) 形式计算算术表达式，这和惠普计算器的方法大致相同。在计算后缀表达式的过程中，用堆栈来保存中间结果。

1）FPU 数据寄存器

• FPU 有 8 个独立的、可寻址的 80 位数据寄存器 R0～R7，这些寄存器合称为寄存器栈。FPU 状态字中名为 TOP 的一个 3 位字段给出了当前处于栈顶的寄存器编号。在编写浮点指令时，这个位置也称为 ST(0)（或简写为 ST），最后一个寄存器为 ST(7)。ST(0) 总是表示栈顶，即随着 push 或 pop 操作变化。指令操作数不能直接引用寄存器编号。

• FPU 实际上用一组有限数量的寄存器实现循环式堆栈。

• 寄存器中浮点数使用的是 IEEE 10 字节扩展实数格式（也被称为临时实数（temporary real））。当 FPU 把算术运算结果存入内存时，它会把结果转换成如下格式之一：整数、长整数、单精度（短实数）、双精度（长实数），或者压缩二进制编码的十进制数（BCD）。

2）FPU 专用寄存器

共 6 个：

• 操作码寄存器：保存最后执行的非控制指令的操作码。

• 控制寄存器：执行运算时，控制精度以及 FPU 使用的舍入方法。还可以用这个寄存器来屏蔽（隐藏）单个浮点异常。

• 状态寄存器：包含栈顶指针、条件码和异常警告。

• 标识寄存器：指明 FPU 数据寄存器栈内每个寄存器的内容。其中，每个寄存器都用两位来表示该寄存器包含的是一个有效数、零、特殊数值 (NaN、无穷、非规格化，或不支持的格式)，还是为空。

• 最后指令指针寄存器：保存指向最后执行的非控制指令的指针。

• 最后数据（操作数）指针寄存器：保存指向数据操作数的指针，如果存在，那么该数被最后执行的指令所使用。

3）FPU 舍入

FPU 可以在四种舍入方法中进行选择，FPU 控制字用两位指明使用的舍入方法，这两位被称为 RC 字段：

1) 00——舍入到最接近的偶数 (round to nearest even):舍入结果最接近无限精确的结果。如果有两个值近似程度相同，则取偶数值 (LSB=0)。

2) 01——向 -∞ 舍入 (round down to -∞)：舍入结果小于或等于无限精确结果。

3) 10——向 +∞ 舍入 (round down to +∞)：舍入结果大于或等于无限精确结果。
4) 11——向 0 舍入 (round toward zero)：也被称为截断法，舍入结果的绝对值小于或等于无限精确结果。

浮点数异常

每个程序都可能出错，而 FPU 就需要处理这些结果。因而，它要识别并检测 6 种类型的异常条件：

无效操作（#I）
除零（#Z）
非规格化操作数（#D）
数字上溢（#O）
数字下溢（#U）
模糊精度（#P）。

前三个（#I、#Z 和 #D）在全部运算操作发生前进行检测，后三个（#O、#U 和 #P）则在操作发生后检测。
每种异常都有对应的标志位和屏蔽位。当检测到浮点异常时，处理器将与之匹配的标志位置 1。每个被处理器标记的异常都有两种可能的操作：

如果相应的屏蔽位置 1，那么处理器自动处理异常并继续执行程序。
如果相应的屏蔽位清 0，那么处理器将调用软件异常处理程序。

大多数程序普遍都可以接受处理器的屏蔽（自动）响应。如果应用程序需要特殊响应，那么可以使用自定义异常处理程序。一条指令能触发多个异常，因此处理器要持续保存自上一次异常清零后所发生的全部异常。完成一系列计算后，可以检测是否发生了异常。

浮点数指令集

1）指令集特点概述
• FPU 指令集有些复杂，因此这里只对其功能进行概述，指令集包括如下基本指令类型：

数据传送
基本算术运算
比较
超越函数
常数加载（仅对专门预定义的常数）
x87 FPU 控制
x87 FPU 和 SIMD 状态管理

• 浮点指令名用字母 F 开头，以区别于 CPU 指令。指令助记符的第二个字母（通常为 B 或 I）指明如何解释内存操作数：B 表示 BCD 操作数，I 表示二进制整数操作数。如果这两个字母都没有使用，则内存操作数将被认为是实数。这个规律基本适用所有的指令。
2）操作数概要
• 浮点指令可以包含零操作数、单操作数和双操作数。如果是双操作数，那么其中一个必然为浮点寄存器。指令中没有立即操作数，但是某些预定义常数（如 0.0，π 和 log210）可以加载到堆栈。
• 通用寄存器 EAX、EBX、ECX 和 EDX 不能作为操作数。（唯一的例外是 FSTSW，它将 FPU 状态字保存在 AX 中。）不允许内存-内存操作。
• 整数操作数从内存（不是从 CPU 寄存器）加载到 FPU，并自动转换为浮点格式。同样，将浮点数保存到整数内存操作数时，该数值也会被自动截断或舍入为整数。

FINIT 初始化

FINIT 指令对 FPU 进行初始化。将 FPU 控制字设置为 037Fh，即屏蔽（隐藏）了所有浮点异常；舍入模式设置为最近偶数，计算精度设置为 64 位。建议在程序开始时调用 FINIT，这样就可以了解处理器的起始状态。

FLD 加载浮点数值

• FLD（加载浮点数值）指令将浮点操作数复制到 FPU 堆栈栈顶（称为 ST(0)）。操作数可以是 32 位、64 位、80 位的内存操作数（REAL4、REAL8、REAL10）或另一个 FPU 寄存器。FLD 支持的内存操作数类型与 MOV 指令一样。

• 下面的指令将特定常数加载到堆栈。这些指令没有操作数：

FLD1 指令将 1.0 压入寄存器堆栈。
FLDL2T 指令将 log210 压入寄存器堆栈。
FLDL2E 指令将 log2e 压入寄存器堆栈。
FLDPI 指令将 π 压入寄存器堆栈。
FLDLG2 指令将 log102 压入寄存器堆栈。
FLDLN2 指令将 loge2 压入寄存器堆栈。
FLDZ（加载零）指令将 0.0 压入 FPU 堆栈。

FST，FSTP 保存浮点数值

• FST（保存浮点数值）指令将浮点操作数从 FPU 栈顶复制到内存，FST 不是弹出堆栈。FST 支持的内存操作数类型与 FLD 一致。操作数可以为 32 位、64 位、80 位内存操作数（REAL4、REAL8、 REAL10）或另一个 FPU 寄存器。

• FSTP（保存浮点值并将其出栈）指令将 ST(0) 的值复制到内存并将 ST(0) 弹出堆栈。

FCHS 和 FABS

FCHS(修改符号) 指令将 ST(0) 中浮点数值的符号取反。FABS (绝对值) 指令清除 ST(0) 中数值的符号，以得到它的绝对值。这两条指令都没有操作数。

FADD、FADDP、FIADD

①FADD

FADD（加法）指令格式如下，其中，m32fp 是 REAL4 内存操作数，m64fp 即是 REAL8 内存操作数，i 是寄存器编号：

```
FADD
FADD m32fp
FADD m64fp
FADD ST(0), ST(i)
FADD ST(i), ST(0)
```

• 无操作数

如果 FADD 没有操作数，则 ST(0)与 ST(1)相加，结果暂存在 ST(1)。然后 ST(0) 弹出堆栈，把加法结果保留在栈顶。

• 寄存器操作数

和 ADD 类似，第一个操作数也是结果的存储地址。

• 内存操作数

如果使用的是内存操作数，FADD 将操作数与 ST(0) 相加。
②FADDP
FADDP（相加并出栈）指令先执行加法操作，再将 ST(0) 弹出堆栈。MASM 支持如下格式：

```
FADDP ST(i),ST(0)
```

③FIADD
FIADD（整数加法）指令先将源操作数转换为扩展双精度浮点数，再与 ST(0) 相加。指令语法如下：

```
FIADD ml6int
FIADD m32int
```

FSUB、FSUBP、FISUB

①FSUB
FSUB 指令从目的操作数中减去源操作数，并把结果保存在目的操作数中。目的操作数总是一个 FPU 寄存器，源操作数可以是 FPU 寄存器或者内存操作数。该指令操作数类型与 FADD 指令一致：

```
FSUB
FSUB m32fp
FSUB m64fp
FSUB ST(0), ST(i)
FSUB ST(i), ST(0)
```

FSUB 的操作与 FADD 相似，只不过它进行的是减法而不是加法。比如，无参数 FSUB 实现 ST(1) - ST(0)，结果暂存于 ST(1)。然后 ST(0) 弹出堆栈，将减法结果留在栈顶。若 FSUB 使用内存操作数，则从 ST(0) 中减去内存操作数，且不再弹出堆栈。
②FSUBP
FSUBP（相减并出栈）指令先执行减法，再将 ST(0) 弹出堆栈。MASM 支持如下格式：

```
FSUBP ST(i),ST(0)
```

③FISUB
FISUB（整数减法）指令先把源操作数转换为扩展双精度浮点数，再从 ST(0) 中减去该操作数：

```
FISUB ml6int
FISUB m32int
```

FMUL、FMULP、FIMUL

①FMUL
•FMUL 指令将源操作数与目的操作数相乘，乘积保存在目的操作数中。目的操作数总是一个 FPU 寄存器，源操作数可以为寄存器或者内存操作数。其语法与 FADD 和 FSUB 相同：

```
FMUL
FMUL m32fp
FMUL m64fp
FMUL ST(0), ST(i)
FMUL ST(i), ST(0)
```

除了执行的是乘法而不是加法外，FMUL 的操作与 FADD 相同。比如，无参数 FMUL 将 ST(0) 与 ST(1) 相乘，乘积暂存于 ST(1)。然后 ST(0) 弹出堆栈，将乘积留在栈顶。同样，使用内存操作数的 FMUL 则将内存操作数与 ST(0) 相乘。

②FMULP

FMULP（相乘并出栈）指令先执行乘法，再将 ST(0) 弹出堆栈。MASM 支持如下格式：

```
FMULP ST(i),ST(0)
```

③FIMUL

FIMUL 与 FIADD 相同，只是它执行的是乘法而不是加法：

```
FIMUL ml6int
FIMUL m32int
```

FDIV、FDIVP、FIDIV

①FDIV

FDIV 指令执行目的操作数除以源操作数，被除数保存在目的操作数中。目的操作数总是一个寄存器，源操作数可以为寄存器或者内存操作数。其语法与 FADD 和 FSUB 相同：

```
FDIV
FDIV m32fp
FDIV m64fp
FDIV ST(0), ST(i)
FDIV ST(i), ST(0)
```

除了执行的是除法而不是加法外，FDIV 的操作与 FADD 相同。比如，无参数 FDIV 执行 ST(1) 除以 ST(0)。然后 ST(0) 弹出堆栈，将被除数留在栈顶。使用内存操作数的 FDIV 将 ST(0) 除以内存操作数。

②FDIVP

③FIDIV

FIDIV 指令先将整数源操作数转换为扩展双精度浮点数，再执行与 ST(0) 的除法。其语法如下：

```
FIDIV ml6int
FIDIV m32int
```

FCOM、FCOMP、FCOMPP

浮点数不能使用 CMP 指令进行比较，因为后者是通过整数减法来执行比较的。取而代之，必须使用 FCOM 指令。

FCOM（比较浮点数）指令将其源操作数与 ST(0) 进行比较。源操作数可以为内存操作数或 FPU 寄存器。其语法如下表所示：

```
FCOM   比较 ST(0) 与 ST(1)
FCOM m32fp    比较 ST(0) 与 m32fp
FCOM m64fp    比较 ST(0) 与 m64fp
FCOM ST(i)    比较 ST(0) 与 ST(i)
```

FCOMP 指令的操作数类型和执行的操作与 FCOM 指令相同，但是它要将 ST(0) 弹出堆栈。FCOMPP 指令与 FCOMP 相同，但是它有两次出栈操作。

1）条件码

FPU 条件码标识有 3 个，C3、C2 和 C0，用以说明浮点数比较的结果，C3、C2 和 C0 的功能分别与零标志位 (ZF)、奇偶标志位 (PF) 和进位标志位 (CF) 相同。

条件	C3（零标志位）	C2（奇偶标志位）	C0（进位标志位）	使用的条件跳转指令
ST(0) > SPC	0	0	0	JA.JNBE
ST(0) < SPC	0	0	1	JB.JNAE
ST(0) = SPC	1	0	0	JE.JZ
无序	1	1	1	（无）

如果出现无效算术运算操作数异常（无效操作数），且该异常被屏蔽，则 C3、C2 和 C0 按照标记为“无序”的行来设置。

在比较了两个数值并设置了 FPU 条件码之后，还需两个步骤：

用 FNSTSW 指令把 FPU 状态字送入 AX。

用 SAHF 指令把 AH 复制到 EFLAGS 寄存器。

条件码送入 EFLAGS 之后，就可以根据 ZF、PF 和 CF 进行条件跳转。

2）FCOMI

Intel P6 系列引入了 FCOMI 指令。该指令比较浮点数值，并直接设置 ZF、PF 和 CF。FCOMI 指令代替了之前代码段中的三条指令，但是增加了一条 FLD 指令。FCOMI 指令不使用内存操作数。

FWAIT 指令（WAIT）

整数 (CPU) 和 FPU 是相互独立的单元，因此，在执行整数和系统指令的同时可以执行浮点指令。这个功能被称为并行性 (concurrency)，当发生未屏蔽的浮点异常时，它可能是个潜在的问题。反之，已屏蔽异常则不成问题，因为，FPU 总是可以完成当前操作并保存结果。

发生未屏蔽异常时，中断当前的浮点指令，FPU 发异常事件信号。当下一条浮点指令或 FWAIT(WAIT) 指令将要执行时，FPU 检查待处理的异常。如果发现有这样的异常，FPU 就调用浮点异常处理程序（子程序）。

如果引发异常的浮点指令后面跟的是整数或系统指令，指令不会检查待处理异常，它们会立即由 CPU 执行。

设置 WAIT 和 FWAIT 指令是为了在执行下一条指令之前，强制处理器检查待处理且未屏蔽的浮点异常。这两条指令中的任一条都可以解决这种潜在的同步问题，直到异常处理程序结束或真的没有异常发生。

其他 FPU 指令

1）FSTCW 和 FLDCW

前者是将 FPU 控制字保存到内存，后者是将内存值加载为控制字。如：

```
FSTCW memoryWord
```

修改控制字可以修改一些舍入方式（RC 字段）、设定异常屏蔽等设定。控制字的含义如下：

位	说明	位	说明
0	无效操作异常屏蔽位	5	精度异常屏蔽位
1	非规格化操作数异常屏蔽位	8～9	精度控制位
2	除零异常屏蔽位	10～11	舍入控制位

3	上溢异常屏蔽位	12	无穷控制位
4	下溢异常屏蔽位		

2）FSQRT

单独使用（无操作数）时表示对 ST(0)取平方根，再存入 ST(0)

高级语言接口

.MODEL 伪指令

16 位和 32 位模式中，MASM 使用 .MODEL 伪指令确定若干重要的程序特性：内存模式类型、过程命名模式以及参数传递规则。若汇编代码被其他编程语言程序调用，那么后两者就尤其重要。

.MODEL 伪指令的语法如下：

```
.MODEL memorymodel [,modeloptions]
```

• MemoryModel

下表列出了 memorymodel 字段可选择的模式。除了平坦模式之外，其他所有模式都可以用于 16 位实地址编程。

模式	说明
微模式	一个既包含代码又包含数据的段。文件扩展名为 .com 的程序使用该模式
小模式	一个代码段和一个数据段。默认情况下，所有代码和数据都为近属性
中模式	多个代码段，一个数据段
紧凑模式	一个代码段，多个数据段
大模式	多个代码段和数据段
巨模式	与大模式相同，但是各个数据项可以大于单个段
平坦模式	保护模式。代码与数据使用 32 位偏移量。所有的数据和代码（包括系统资源）都在一个 32 位段内

• ModelOptions

.MODEL 伪指令中的 ModelOptions 字段可以包含一个语言说明符和一个栈距离。语言说明符指定过程与公共符号的调用和命名规范。栈距离可以是 NEARSTACK（默认值）或者 FARSTACK。

伪指令 .MODEL 有几种不同的可选语言说明符，其中的一些很少使用（比如 BASIC、FORTRAN 和 PASCAL）。反之，C 和 STDCALL 则十分常见。语言说明符 STDCALL 用于 Windows 系统函数调用。在链接汇编代码和 C 与 C++ 程序时，使用 C 语言说明符。

__asm 伪指令

在 Visual C++ 中，伪指令 __asm 可以放在一条语句之前，也可以放在一个汇编语句块（称为 asm 块）之前。注意在“asm”的前面有两个下划线。

• 编写内嵌汇编代码时允许：

使用 x86 指令集内的大多数指令。

使用寄存器名作为操作数。

通过名字引用函数参数。

引用在 asm 块之外定义的代码标号和变量。（这点很重要，因为局部函数变量必须在 asm 块的外面定义。）

使用包含在汇编风格或 C 风格基数表示法中的数字常数。比如，0A26h 和 0xA26 是等价的，且都能使用。
在语句中使用 PTR 运算符，比如 inc BYTE PTR[esi]。
使用 EVEN 和 ALIGN 伪指令。
•编写内嵌汇编代码时不允许：
使用数据定义伪指令，如 DB（BYTE）和 DW（WORD）。
使用汇编运算符（除了 PTR 之外）。
使用 STRUCT、RECORD, WIDTH 和 MASK。
使用宏伪指令，包括 MACRO、REPT、IRC、IRP 和 ENDM，以及宏运算符（<>、!、&、% 和 .TYPE）。
通过名字引用段。（但是，可以用段寄存器名作为操作数。）

部分汇编器特殊语法积累

MASM

ds:[00000000h] ;数据区任意位置的内容，常和 type PTR 运算符合用

Visual Studio 常用功能积累

查看 OBJ 文件中所有过程名

•要查看 OBJ 文件中所有的过程名，使用 Visual Studio 中的 DUMPBIN 工具，选项为 /SYMBOLS。

查看编译器生成的汇编语言代码

•用 Visual Studio 调试 C 和 C++ 程序时，若想查看汇编语言源代码，就在 Tools 菜单中选择 Options。再在 Debugging 标签的 General 分类中，选择 Enable address-level debugging。上述设置要在启动调试器之前完成。接着，在调试会话开始后，右键点击源代码窗口，从弹出菜单中选择 Go to Disassembly。
•另一种方法是，在 Project 菜单中选择 Properties，生成一个列表文件。在 Configuration Properties，选择 Microsoft Macro Assembler，再选择 Listing Fileo 在对话窗口中，将 Generate Preprocessed Source Listing 设置为 Yes，List All Available Information 也设置为 Yes。

将所有章节的 visual studio 搜索内容归此