FASM 1.67 程序员手册

From: xuyibo.org

Updated: 2008-04-20

FASM 1.67 程序员手册

这一章包含开始使用 FASM 前的所有必须知识，在使用 FASM 前应至少阅读此章。

1.1 编译器概述

FASM 是一个 x86 体系处理器下的汇编语言编译器，它可以通过多遍扫描来优化生成的机器码。

这篇文档还描述了用于 windows 系统的 IDE 版本，这个版本带有界面，并且有一个集成的编辑器。但从编译的角度，它和命令行版本是一样的。IDE 版本的可执行文件为 fasmw.exe，命令行的为 fasm.exe

1.1.1 系统需求

所有版本都需要 x86 平台 32 位处理器（至少 80386），虽然可以生成 x86 体系处理器 16 位程序。Windows 控制台版本需要任意 Win32 操作系统；GUI 版本需要 Win32 GUI 4.0 或更高版本，所以它可以运行在任何兼容 Windows 95 的系统上。

这个版本提供的 example 代码需要设置 INCLUDE 变量为 FASM 包目录下的 include 目录才能正确编译。 比如 FASM 包位置为 d:\fasm：右键点击【我的电脑】->【属性】->【高级】->【环境变量】，在弹出的对话框中，在下面的系统变量中，如果里面存在 INCLUDE 环境变量，那么双击其并将 d:\fasm\include 添加到变量值中，注意必须用分号分隔变量；如果不存在 INCLUDE 环境变量，点击添加 INCLUDE 环境变量。

如果你使用 FASMW 来编译，还有另一个方法，你可以在 d:\fasm\fasmw.ini 文件末尾添加下面的内容：

```
[Environment]
Include = c:\fasmw\include
```

如果不设置好 INCLUDE 环境变量，当 include 文件的时候，就必须提供完整的 include 文件路径。

1.1.2 编译器使用

开始使用 FASM，可以简单的双击 fasmw.exe 文件图标，或者拖拽一个源码文件到此图标。你也可以打开 fasmw.exe 后使用菜单【文件】->【打开】来打开源码文件，或者拖拽文件到编辑窗口。你可以一次编辑多个文件，每一个文件都在编辑窗口底部占用一 tab 按钮，点击相应的按钮就可以切换到该文件。FASM 默认将编译当前编辑的文件，但你可以通过右键点击该文件的 tab 按钮，让编译器来强制每次编译此文件。一次只能有一个文件可以指派给编译器。

当你的源码文件都准备好后，你可以执行运行菜单中的编译来执行此文件。编译成功后，编译器将显示编译过程总结;否则将显示发现的错误。编译总结包括编译了多少遍、消耗的时间、写入多少字节到目标文件。它还包含一个【显示】文本框，用来显示任何源码中的 display 指令。错误总结至少包含错误信息和一个显示文本框。如果错误和源码中的某些行有关，总结将包含指令段，用来显示预处理后导致错误的指令，和源码列表，显示和错误相关的源码行位置，如果你从列表中选择一行，那么编辑窗口也将选择相应的行。（如果此行的文件还没有加载，那么将自动加载。）

运行命令也调用执行编译器，并且在编译成功后如果此格式能在 Windows 环境下执行的话执行编译的程序；否则将弹出消息提示此类型文件不能执行。如果发生错误，编译器显示和编译命令相同的提示。

如果编译器运行超出内存，你可以在【选项】菜单中的【编译器设置】对话框中增加内存分配。你可以设置编译器应当使用多少 KB 字节，以及编译线程的优先级。

1.1.3 编辑器选项

在【选项】菜单中还包含一些编辑器选项，用来影响编辑器行为的开关。这一节中将描述此选项。

安全选择 - 当打开此选项的时候，当开始键入的时候，选择的文本将不会被删除。当你做任何文本修改操作时，选择部分将被撤销，不会影响任何选中的文本，并且之后会执行那个命令。当这个选项关闭的时候，当你键入的时候，选中的文本将被删除，Del 键也会删除选中的块（当安全选中开启时，你必须使用 Ctrl+Del 才能删除此文件）。

自动填充 - 当你键入任何开始括号的时候，编辑器将自动键入关闭括号。

自动缩进 - 当你键入回车开始新行时，光标停在和上一行第一个非空格所在的位置。当你分割行时，新的行也会开始在相同的缩进位置，任何新行后面的空白字符将被忽略掉。

智能制表键 - 当你按下 Tab 键的时候，将移动到上一行非空白字符开始处的下一个 tab 位置。如果在上一行没有找到相应的位置，将缩进 8 个字符。

保存优化 - 如果允许此选项，当保持文件的时候，空白区域将被优化的 tab 和空格填充来减少文件的大小。如果关闭此选项，空白区域将填充为空格（不保存最后一行的空格）。

Revive dead keys - left to do.

1.1.4 在命令行下执行编译器

在命令行下执行编译需要运行 fasm.exe。fasm 接受两个参数 - 第一个提供源码文件，第二个提供目标文件。如果没有给定第二个文件，输出文件名称将自动猜测一个。当显示简短的程序名称和版本后，编译器从源码文件中读取数据并且编译它。当编译成功，编译器将写入生成的文件到目标文件，并且显示编译过程总结；否则将显示发生的错误信息

源码文件必须是文本格式的，行结束符接受 DOS（CR+LF）和 Unix（LF）两种格式，tab 将被当做空格处理。

在命令行中你可以指定-m 选项用来指定 fasm 汇编器最大使用的内存（KB）。在 DOS 版本中，这个选项仅用来限定扩展内存的使用。-p 选项后面用来指定汇编器要执行的遍数。如果代码不能再指定的遍后生成，汇编器将结束并给出错误信息。最大值为 65536，默认值为 100。这个参数可以用来限制汇编器最多执行的遍数，-p 参数跟随一指定的最大遍数即可。

没有命令行参数来影响输出，flat 汇编器仅需要源码文件来包含真正需要的信息。例如，为了制定输出格式你可以在源码文件开头使用 format 指令。

1.1.5 命令行编译器消息

如上面描述的那样，当成功编译后，编译器将显示编译总结。它包含执行了多少遍，消耗的时间，以及写入了多少字节到目标文件。下面是一个编译总结例子：

```
flat assembler version 1.66
38 passes, 5.3 seconds, 77824 bytes.
```

当编译错误时，程序将显示错误信息。比如，当编译器找不到收入文件时，将显示下面的信息：

```
flat assembler version 1.66
error: source file not found.
```

如果错误和部分源码相关，导致错误的源码行将被显示。相应行的位置也会给出，以帮助你快速定位错误，比如：

```
flat assembler version 1.66
example.asm [3]:
```

```
mob ax,1
error: illegal instruction.
```

意思是 example.asm 的第三行编译时遇到了无法识别指令。如果导致错误行包含一个宏指令，生成错误指令的宏指令定义也将显示。比如：

```
flat assembler version 1.66
example.asm [6]:
stoschar 7
example.asm [3] stoschar [1]:
mob al,char
error: illegal instruction.
```

它的意思是 example.asm 的第六行宏指令生成了一个无法识别的指令，以及宏指令的第一行定义。

1.1.6 输出格式

当源码中没有 format 指令时，flat 简单的把生成的代码到输出文件中，创建 flat 二进制文件。默认生成的是 16 位代码，你可以通过 use16 或 use32 指令打开 16 位或者 32 位模式。一些选择一些输出格式时将切换到 32 位模式 - 更多你可以选择的格式可以参考 2.4 节。

输出文件的扩展名编译器将根据输出格式自动选择

所有输出代码顺序将和源码文件的顺序一样。

1.1.2 汇编语法

下面的信息主要是给使用过其他汇编器的汇编程序员看的。如果你是初学者，你应当寻找汇编编程的教程。

Flat 汇编器默认采用 Intel 的语法，虽然你可以使用预处理（宏指令和符号常量）来定制。它也包含一套伪指令 - 编译器的指令。

源码中定义的所有符号都区分大小写的。

操作符	位	字节
byte	8	1
word	16	2
dword	32	4
fword	48	6
pword	48	6
qword	64	8
tbyte	80	10
tword	80	10
dqword	128	16

表 1.1：size 操作符

类型	位数	
通用寄存器	8	al cl dl bl ah ch dh bh
	16	ax cx dx bx sp bp si di
	32	eax ecx edx ebx esp ebp esi edi
段寄存器	16	es cs ss ds fs gs
控制寄存器	32	cr0 cr2 cr3 cr4
调试寄存器	32	dr0 dr1 dr2 dr3 dr6 dr7
FPU	80	st0 st1 st2 st3 st4 st5 st6 st7
MMX	64	mm0 mm1 mm2 mm3 mm4 mm5 mm6 mm7
SSE	128	xmm0 xmm1 xmm2 xmm3 xmm4 xmm5 xmm6 xmm7

表 1.2:寄存器

1.2.1 指令语法

指令在汇编语言中是用行结束来分割的，一条指令占用一行文本。如果一行包含分号，除了双引号字符串中的分号外，这行剩余部分为一注释，编译器将忽略掉。如果一行为“\”字符（后面可能出现分号和注释），下一行将被连在“\”所在位置。

源码中每一行都是一些元素序列，其中可能为 3 中方式的一种。一种是符号字符，用来分割元素即使它们没有空格分开。任意的 +-*/=<>()[]{}:,|&~#‘ 为符号字符。其它字符，用空格或者符号字符串分割为符号。如果符号的第一个字符为单引号或双引号，其后的任

意字符序列，甚至特殊字符，将被当做引用字符串。不为符号字符和引用字符串，可以用作名称，也叫做名称符号。

每一个指令包含助记符和一些用逗号分隔的操作数。操作数可以为寄存器、立即数或者内存中的数据，在操作数之前可以跟着 size 操作符，用来定义或重写大小（表 1.1）。表 1.2 列出了可用的寄存器的名称，他们的大小是不能覆盖的。立即数可以指定为任意数值表达式。

当操作数为内存中数据是，数据的地址（也可以为任意数值表达式，但必须包含寄存器）必须用中括号括起来或者之前包含 ptr 运算符。例如：
指令 mov eax, 3 将把立即数 3 送到 eax 寄存器;
指令 mov eax, [7] 将把 32 位数据从地址 7 送到 eax；
指令 mov byte [7], 3 将把立即数 3 赋值给地址 7，也可以写成：mov byte ptr 7, 3。
为了指定寻址所用的段寄存器，段寄存器紧跟冒号，放在地址值前面（在中括号中或 ptr 运算符后面）。

1.2.2 数据定义

定义数据或保留空间，可以使用表 1.3 中的伪指令。数据定义伪指令必须跟着一个或多个逗号分隔的数值表达式。这些表达式定义的数据单元大小取决于使用的伪指令。例如：db 1,2,3 将分别定义 3 个字节数据 1, 2, 3。

db 和 du 指示符还接受任意长度的字符串。当使用 db 时，将被转换为字节序列；使用 du 的时候，将被转换为高字节为 0 的字序列。例如 db 'abc'将定义三个字节数据 61，62 和 63。

dp 指示符和其等价的 df 接受两个用冒号分隔的数字表达式为参数，第一个为高字，第二个将变成 far 指针值的低 DWORD。dd 也允许两个用冒号分隔的 word 指针，dt 只允许接受一个浮点参数并以扩展双精度浮点格式创建数据。

上面的任意伪指令都允许使用 dup 操作符来重复拷贝给定的值。重复次数必须在该操作符前面，后面为要重复的值 - 也可以为一串用逗号分隔的值，如果这样的话必须用括号将这些值括起来，如 db 5 dup(1,2)定义了五份给定两个字节序列的拷贝。

file 是特殊的伪指令，其语法也是不同的。这个伪指令包含来自文件的字节流，它后面必须跟着文件名，然后是可选的文件偏移数值表达式（前面有一冒号），然后也是可选的逗号和要包含多少字节的数值表达式（如果没有指定的话将包含文件中的所有数据）。例如：

file 'data.bin'将包含这个文件为二进制数据。
应该是 file 'data.bin':10h,4 将只包含从 10h 文件偏移开始后的 4 个字节。

大小（字节）	定义数据	保留数据
1	db file	rb
2	dw du	rw
4	dd	rd
6	dp df	rp rf
8	dp df	rp rf
10	dt	rt

表 1.3 数据伪指令

数据保留伪指令值允许跟着一个数值表达式，这个值定义了多少个指定大小单元空间将被保留。所有的数据定义伪指令都允许“?”值，意思是这个单元不应初始化为任何值，效果和数据保留伪指令相同。未初始化的数据可能没有存在于输出文件中，所以其值应当总是被认为是不可知的。

1.2.3 常数和标号

在数值表达式中你可以使用常数或者标号来替代数字。常数或标号定义应当使用特殊的伪指令。每一个标号只允许定义一次，它可以在源码中的任何地方使用（即使在定义前）。常数可以定义多次，此时它只能在定义后才能使用，而且其值总是等于使用位置前最后一次定义的值。当常数在源码中只定义了一次，那么和标号相同可以在源码中的任何位置使用。

常数定义包含常数名后面跟着“=”字符以及数值表达式，这个在常数定义时计算数据表达式的值将成为常数的值。例如你可以使用伪指令“count=17”定义 count 常数，然后再汇编指令中使用它，比如 mov cx, count - 编译时将变成 mov cx, 17。

有不同的几种方式来定义标号。最简单的是在标号名后面跟着冒号，这条伪指令同行的后面甚至可以跟着其他指令。它定义的标号的值为定义位置的偏移。这种方式通常用来定义代码中的标号。另一种方式是标号名（没有冒号）后面跟着一些数据伪指令。它定义的标号的值为定义数据起始位置的偏移，并作为一个标号记住这个数据，其单元大小由表 1.3 中的数据伪指令指定。

标号可以当作标记代码或数据位置偏移的常数值。例如当你使用标号伪指令“char db 224”定义了数据，为了将这个数据的偏移放到 bx 寄存器，你应当使用“mov bx, char”指令，为了将 char 处的字节数据移动到 dl 寄存器，你应当使用“mov dl,[char]”（或者“mov dl, ptr char”）。当当你试图汇编“mov ax, [char]”，将会产生错误，因为 FASM

会比较操作数的尺寸，以确保它们是相等的。你可以通过 size 覆盖来强制汇编那条指令：“mov ax, word [char]”，但记住这条指令将在 char 位置读取两个字节，而实际上 char 只定义了一个字节。

最后也是最灵活的定义标号的方式是使用 label 伪指令。这条伪指令后面为标号名，然后是可选的 size 操作符，后面是可选的 at 操作符，以及标号定义地址的数值表达式。例如：“label wchar word at char”将为 char 地址的 16 位数据定义一个新的标号。现在“mov ax, [wchar]”将和“mov ax, word[char]”编译后的结果相同。如果没有指定任何地址，label 伪指令就在当前位置定义标号。因此“mov [wchar], 57568”将拷贝两个字节，而“mov [char], 224”将拷贝一个字节到同一的地址。

以"."开头的标号被认为是局部标号，它附加在最后一个全局标号的后面（名称不以点开头的）来组成完整的标号名。所以你可以在定义另一个全局标号前使用这个标号的短名称（以点开头的），在其他位置你就必须使用完整的标号名。以两个点开头的“..”标号是个特例 - 它们如同全局变量，但它们不会成为局部标号的前缀。

@@为匿名标号，你可以在源码中多次定义它们。符号@b（或者等价于@r）引用最近的前面的匿名标号，符号@f 引用最近的后面的匿名标号。这些特殊标号都不区分大小写。

1.2.4 数值表达式

在上面的例子中所有的数值表达式的都是简单的数字、常数或标号。通过编译期间计算的算术或者逻辑操作符也可以变得更复杂些。所有这些操作符和他们的优先级都列在表 1.4 中。高优先级运算操作先计算，当然你可以通过将某部分表达式用括号括起来来改变它的优先级。+、-、*和/是标准的算术运算操作，mod 计算除操作后的余数。
and、or、xor、shl、shr 和 not 执行和汇编指令中同名指令相同的逻辑操作。rva 用来转换一个地址到重定位的偏移，特定于某些输出格式。（见 2.4）

优先级	操作符
0	+
	-
1	*
	/
2	mod
	and
3	or
	xor
4	shl
	shr
5	not
6	rva

表 1.4：算术和逻辑操作符优先级

表达式中的数字默认为十进制的，二进制数字可以在后面跟着字母 b，八进制的跟着字母 o，十六进制的以 0x 字母开头（如果 C 语言）或者以$开头（如果 Pascal 语言）或者以 h 字母结尾。当在表达式中遇到字符串时将被转换为数字 - 第一个字符将成为数字的最低位。

数值表达式用作地址可以用任意通用寄存器来寻址。它们可以加上或者乘以某个合适的值。

数值表达式中也可以使用一些特殊符号。第一是“$”，其值等于当前偏移值，而“$$”和当前地址空间的基地址相等。还有“%”，表示在某部分代码中使用特殊伪指令（见 2.2）时当前重复次数。还有%t 符号，等同于当前的时间戳。

任何数值表达式都可以包含科学计算法表示的单浮点数值（FASM 不允许编译期间的浮点运算），它们可以以字母 f 结尾，否则它们必须至少包含字符"."或"E"。所以"1.0"，"1E0"和"1f"定义了相同的浮点数据，而简单的"1"定义了一个整型值。

1.2.5 跳转和调用

任何的 jmp 和 call 指令操作数前面不仅放 size 操作符，也可以放跳转类型操作符：short，near 或 far。例如当汇编器在 16 位模式下，指令 jmp dword [0]将成为远跳转，而当在 32 位模式下，它将成为 near 跳转。为了强制这条指令来区分对待，可以使用 jmp near dword [0]或者 jmp far dword[0]格式。

当 near 跳转的操作数为立即数时，如果可能的话，汇编器将生成最短格式的跳转指令（但不要在 16 位模式下创建 32 位指令，也不要在 32 位模式下创建 16 位代码，除非它前面有 size 操作符）。通过指定跳转类型你可以强制生成长格式（例如“jmp near 0”）或者生成短格式并且如果不可能的话将产生错误（例如“jmp short 0”）。

1.2.6 操作数尺寸设置

当指令使用内存寻址时，如果寻址值在某个范围内，默认将使用短偏移来生成最短格式的指令。这可以通过在中括号中地址前面的 word 或 dword 操作符（或者 ptr 操作符后面）来重写，以强制使用长偏移。当地址不基于任何寄存器时，这些操作符将选择绝对寻址的合适模式。

指令 adc、add、cmp、or、sbb、sub 和 xor 第一个操作数为 16 位或者 32 位默认生成段的 8 位格式，如果第二个操作数为带符号字节范围内的立即数，将产生短格式的指令。可以通过在立即数前面放置 word 或 dword 操作符来重写。imul 指令简单的规则是最后一个操作数为立即数。

push 指令后面如果为立即数且没有 size 操作前缀的话，在 16 位模式下将当作一 word 值，在 32 位模式下将作为一 dword 值，如果可能将使用这条指令的短的 8 位格式，word 或者 dword size 操作符强制 push 指令生成指定大小的长格式。pushw 和 pushd 助记符强制汇编器生成 16 位或 32 位代码，而不是强制它使用长格式指令。

2.1 x86 体系指令

这一章讲述了汇编语言指令语法和功能。更多的技术信息可以阅读 Intel 软件开发者手册。

汇编指令有助记符（指令名称）和 0 到 3 个操作符组成。如果有大于两个的操作符，通常第一个为目的操作符第二个为源操作符。每个操作符都可以为寄存器，内存或立即数（操作符语法见 1.2 节）。每条指令描述后悔有操作符不同用法的例子。

一些指令用作前缀，可以和其他指令放在同行一起使用，一行也允许有多个前缀。段寄存器也是指令助记符前缀，但推荐在方括号中段重写来替代这些前缀。

2.1.1 数据传送指令

mov 从源操作符传送字节，字或双字道目的操作符。它可以在通用寄存器之间，通用寄存器到内存，或从内存到通用寄存器间传送数据，但不能在内存间传送数据。它也可以在立即数到通用寄存器或内存，段寄存器到通用寄存器或内存，通用寄存器或内存到段寄存器，控制或调试寄存器到通用寄存器以及通用寄存器到控制或调试寄存器间传送数据。只有当源操作符和目的操作符大小相同时 mov 才能被汇编。下面是一些例子：

```
mov bx,ax      ; 通用寄存器到通用寄存器
mov [char],al  ; 通用寄存器到内存
mov bl,[char]  ; 内存到通用寄存器
mov dl,32      ; 立即数到通用寄存器
mov [char],32  ; 立即数到内存
mov ax,ds      ; 段寄存器到通用寄存器
mov [bx],ds    ; 段寄存器到内存
mov ds,ax      ; 通用寄存器到段寄存器
mov ds,[bx]    ; 内存到段寄存器
mov eax,cr0    ; 控制寄存器到通用寄存器
mov cr3,ebx    ; 通用寄存器到控制寄存器
```

xchg 置换两个操作数内容。它可以用来置换两个字节、字或者双字操作数。操作数的顺序并不重要。操作数可以为两个通用寄存器，或者通用寄存器同内存。例如：

```
xchg ax,bx     ; 置换两个通用寄存器
xchg al,[char] ; 寄存器和内存置换
```

push 递减堆栈指针（esp 寄存器），然后传送操作数到 esp 执行的栈顶。操作数可以为内存，通用寄存器，段寄存器或字、双字立即数。如果操作数为没有指定大小的立即数时，汇编器在 16 位模式下默认将当作 16 位值，在 32 位模式下将当作 32 位值。pushw 和

pushd 助记符为 push 指令的变种，分别用来压入 16 位，32 位大小值到堆栈。如果同行后指定了更多参数（空格分隔，而非逗号），将汇编为一串 push 指令。下面为带有单一操作符的例子：

```
push ax      ; 压入通用寄存器到堆栈
push es      ; 压入段寄存器
pushw [bx]   ; 压入内存值
push 1000h   ; 压入立即数
```

pusha 压入 8 个通用寄存器的内容到堆栈，这条指令没有操作数。这条指令有两个版本，一个 16 位的和一个 32 位的，汇编器自动根据当前模式生成正确的版本，但也可以使用 pushaw 或 pushad 助记符重写为只为 16 位或 32 位版本。16 位版本的这条指令将以以下顺序压入通用寄存器：ax，cx，dx，bx，压入 ax 前的 sp 值，bp，si 和 di。32 为版本将以相同顺序压入等价的 32 位通用寄存器。

pop 传送当前栈顶的字或双字到目的操作符，然后递增 esp 指向新的栈顶。操作符可以为内存，通用寄存器或段寄存器。popw 和 popd 助记符为 pop 指令的变种，分别用来弹出字或双字。如果同行后指定了更多参数（空格分隔，而非逗号）将汇编为一串 pop 指令。

```
pop bx       ; 弹出栈顶数据到通用寄存器
pop ds       ; 弹出到段寄存器
popw [si]    ; 弹出到内存
```

popa 弹出堆栈中由 pusha 指令压入的寄存器，将忽略其中保存的 sp（或 esp）值。使用 popaw 或 popad 助记符来强制汇编 16 位或 32 位版本的这条指令。

2.1.2 类型转换指令

类型转换指令转换字节为字，字为双字，双字为四字。这些转换可以为符号扩展或零扩展的。符号扩展将用符号位来填充，而零扩展将使用 0 来填充。

cwd 和 cdq 分别用来扩展 ax 和 eax 大小，并将额外位存储到 dx 和 edx 中。转换将使用符号扩展。这些指令没有操作数。

cbw 符号扩展 al 的值到 ax，cwde 符号扩展 ax 到 eax。这些指令也没有操作数。

movsx 使用符号扩展将字节转换为字或双字，字转换为双字。movzx 类似，只是它使用 0 扩展。源操作数可以为通用寄存器或内存，目的操作数必须为通用寄存器。例如：

```
movsx ax,al         ; 字节寄存器转换为字寄存器
movsx edx,dl        ; 字节寄存器转换为双字寄存器
movsx eax,ax        ; 字寄存器转换为双字寄存器
movsx ax,byte [bx]  ; 字节内存之后为字寄存器
movsx edx,byte [bx] ; 字节内存转换为双字寄存器
movsx eax,word [bx] ; 字内存转换为双字寄存器
```

2.1.3 二进制算术指令

add 替换目的操作数的值为源操作数和目的操作数的和，并且在溢出时设置 CF 标志。操作数可以为字节，字或双字。目的操作数可以为通用寄存器或内存，源操作数可以为通用寄存器或立即数，如果目的操作数为寄存器也可以为内存。

```
add ax,bx     ; add 寄存器到寄存器
add ax,[si]   ; add 内存到寄存器
add [di],al   ; add 寄存器到内存
add al,48     ; add 立即数到寄存器
add [char],48 ; add 立即数到内存
```

adc 和 add 类似，只是如果设置 CF 的话结果还将递增 1。add 后跟着多个 adc 指令能用来计算大于 32 位值的和。

inc 将操作数值递增 1，它不影响 CF。操作数可以为通用寄存器或内存，操作数大小可以为字节，字或双字。

```
inc ax        ; 寄存器值递增 1
inc byte [bx] ; 内存值递增 1
```

sub 用目的操作数值减去源操作数，并且用结果替换目的操作数。如果需要借位，将设置 CF。操作数规则和 add 指令相同。

sbb 和 sub 类似，只是如果设置 CF 的话结构还将递减 1。操作数规则和 add 质量相同。sub 后跟着多个 sbb 指令能用来计算大于 32 位值的差。

dec 将操作数值递减 1，它不影响 CF。操作数规则和 inc 指令相同。

cmp 用目的操作数减去源操作数，类似 sub 指令更新标志值，但它不改变源和目的操作符。操作数规则和 sub 指令相同。

neg 用 0 减去带符号的整数操作数。这条指令的效果是将带符号的操作数从正数变为负数或者从负数变为正数。操作数规则和 inc 指令相同。

xadd 交换目的和源操作数，然后载入两个值的和到目的操作数。操作数规则和 add 指令相同。

所有上面的二进制算术指令都将更新 SF，ZF，PF 和 OF 标志。SF 被设置为结果符号位的值，ZF 当结果为 0 时设置为 1，PF 当低 8 位存在偶数个 1 时设置，OF 在结果对于正数太大或对于负数太小（超过符号位）以放到目的操作数中时设置。

mul 计算无符号操作数和累加器的积。如果为 8 位操作数，将和 al 计算积，16 位结果返回到 ah 和 al 中。如果 4，为 16 位操作数，将和 ax 计算积，32 位结果返回到 dx 和 ax 中。如果为 32 位操作数，将和 eax 计算积，64 位结果返回到 edx 和 eax 中。当结果高半部分不为零时将设置标志 CF 和 OF，否则将清除该标志。操作数规则和 inc 指令相同。

imul 执行符号乘法运算。这条指令有 3 种用法。第一种允许一个操作数，和 mul 指令类似。第二种有两个操作数，此时将计算目的操作数和源操作数的积，并将结果替换目的操作数。目的操作数可以为 16 位或 32 位通用寄存器，源操作数可以为通用寄存器，内存或立即数。第三种有 3 个操作数，目的操作数必须为 16 位或 32 位通用寄存器，源操作数可以为通用寄存器或内存，第三种操作数必须为立即数。源操作数乘以立即数并将结果保存到目的寄存器。所有上面三种形式都将计算出双倍大小的结构，并当结果高半部分不为零时设置标志 CF 和 OF。所以第二种和第三种形式也能用作无符号操作数，因为，无论操作数是否为有符号无符号，结果的低半部分是相同的。下面的所有三种形式的乘法指令使用例子：

```
imul bl         ；累加器和寄存器
imul word [si]  ；累加器和内存
imul bx,cx      ；寄存器和寄存器
imul bx,[si]    ；寄存器和内存
imul bx,10      ；寄存器和立即数
imul ax,bx,10   ；寄存器，立即数，值到寄存器
imul ax,[si],10 ；内存，立即数，值到寄存器
```

div 计算操作数和累加器无符号运算的商。被除数（累加器）为两倍大小的除数（操作数），商和余数和除数有相同尺寸。如果除数为 8 位，被除数为 ax，商和余数分别保存到 al 和 ah 中。如果除数为 16 位，被除数的商的高半部分从 dx 获取，低半部分从 ax 获取，

商和余数分别保存到 ax 和 dx 中。如果除数为 32 位，被除数的高半部分从 edx 获取，低半部分从 eax 获取，商和余数分别保存到 eax 和 edx 中。操作数规则和 mul 指令相同。

idiv 计算操作数和累加器有符号运算的商。它使用和 div 指令相同的寄存器，操作数的规则也是一样的。

2.1.4 十进制算术指令

十进制算术指令用来调整上一节的二进制算术操作以生成有效的压缩或未压缩十进制结果，或调整输入为一个二进制算术操作序列以使该操作能生产一个有效的压缩或解压缩十进制结果。

daa 调整 al 中两个有效压缩十进制操作数和的值。daa 必须跟着两对压缩十进制数（每半字节一个点）的和来得到一对有效压缩十进制数字结果。如果需要进位将设置 CF 标志。这条指令没有操作数。

das 调整 al 中两个有效压缩十进制操作数差的值。das 必须跟着两对压缩十进制数（每半个字节一个点）的差来得到一对有效压缩十进制数字结果。如果如要进位将设置 CF 标志。这条指令没有操作数。

aaa 修改 al 中的内容为有效的未压缩十进制数字，并将高四位清零。aaa 必须跟着 al 中两个有效未压缩十进制操作数和。如果需要进位将设置 CF 标志并递增 ah 的值。这条指令没有操作数。

aas 修改 al 的值为一个有效的未压缩十进制数据，并将高四位清零。aas 必须跟着 al 中两个有效未压缩十进制操作数差。如果需要进位将设置 CF 标志并递减 ah 的值。这条指令没有操作数。

aam 修正两个有效未压缩十进制数的积。aam 必须跟着两个十进制数字的积来生成一个有效的十进制结果。数字的高位在 ah 中，低位在 al 中。这条指令用来调整 ax 来生成两个任何基数的未压缩数字。标志版本的这条指令没有操作数，另一种有一个操作数 - 一个指定创建数字基数的立即数。

aad 修改 ah 保存的分子和 ah 和 ah 中两个有效未压缩十进制操作数的商，所以计算的商将为一个未压缩十进制数据。ah 为高位，al 为低位。这条指令调整 al 的值，结果也在 al 中，而 ah 内容为 0.这条指令用来调整任何基数的两个未压缩数字。操作数规则和 aam 质量相同。

2.1.5 逻辑指令

not 将指定操作数求反。它不影响标志。操作数规则和 inc 指令相同。

and，or 和 xor 质量执行标准的逻辑操作。它们更新标志 SF，ZF 和 PF。操作数规则和 add 指令相同。

bt，bts，btr 和 btc 指令只能处理一个在内存或寄存器中的位。该位位置由操作数的低位指定。偏移有第二个操作数指定，它可以为字节立即数或一个通用寄存器。这些指令首先将选择的位送到标志 CF。bt 指令不会做更多操作，bts 设置选择位为 1，btr 将选择为置为 0，btc 修改将改位值求反。第一个操作数可以为字或双字。

```
bt ax,15        ；测试寄存器中的位
bts word [bx],15；测试并设置内存值中的位
btr ax,cx       ；测试并重置寄存器中的位
btc word [bx],cx；测试并求反内存值中的位
```

bsf 和 bsr 质量扫描字或双字第一个为 1 的位，并将改为索引保存到必须为通用寄存器的目的操作数。源操作数可以为通用寄存器或内存。当整个串为 0 时设置 ZF 标志；否则将置为 0。如果没有找到为 1 的位，谜底寄存器的值为未定义的。bsf 从低位到高位扫描（从位索引 0 开始）。bsr 从高位到低位扫描（16 位时从第 15 位，32 位时从 31 位开始）。

```
bsf ax,bx       ；向前扫描寄存器
bsr ax,[si]     ；逆向扫描内存值
```

shl 左移目的操作数为第二个操作数指定的位数。目的操作数可以为字节，字，或双字通用寄存器或内存。第二个操作数可以为立即数或 cl 寄存器。左移的最后一位将被放到标志 CF 中。sal 和 shl 为相同指令。

```
shl al,1        ；左移寄存器一位
shl byte [bx],1 ；左移内存值一位
shl ax,cl       ；左移寄存器为 cl 中的值
shl word [bx],cl；左移内存值为 cl 中的值
```

shr 和 sar 右移目的操作数为第二个参数指定的位数。操作数规则和 shl 指令相同。shr 右移的最后一位将放到标志 CF 中。sar 保留操作数符号位，如果操作数为正数用 0 左移，否则用 1 左移。

shld 左移目的操作数（第二个操作数）为第三个操作数指定的位数，lefto do。目的操作数为字或双字通用寄存器或内存，源操作数必须为通用寄存器，第三个操作数可以为立即数或 cl 寄存器。

```
shld ax,bx,1      ; 左移寄存器 1 位
shld [di],bx,1    ; 左移内存值一位
shld ax,bx,cl     ; 左移寄存器为 cl 中的位数
shld [di],bx,cl   ; 左移内存值为 cl 中的位数
```

shrd 右移目的操作数，lefto to do。不修改源操作数。操作数规则和 shld 指令相同。

rol 和 rcl 左转字节，字或双字目的操作数为第二个操作数指定的位数。对于每次转动，左转出来的位数将成为这个值新的低位。rcl 指令还将把高位放到标志 CF 中。操作数规则和 shl 指令相同。

ror 和 rcr 右转字节，字或双字目的操作数为第二个操作数指定的位数。对于每次转东，右转出来的位数将成为这个值新的高位。rcr 指令还将把低位放到标志 CF 中。操作数规则和 shl 质量相同。

test 执行和 and 指令相同的操作，但它不会修改目的操作数的值，只更新标志。操作数规则和 and 指令相同。

bswap 翻转 32 位通用寄存器：0 到 7 位翻转为 23 到 31 位，8 到 15 位翻转为 16 位到 23 位。这条指令用来转换 little-endian 值为 big-endian 格式，反之亦然。

```
bswap edx         ; 翻转寄存器值
```

2.1.6 控制转移指令

jmp 无条件转移控制到目的位置。目的地址可以直接在指令中指定或间接通过寄存器或内存，允许的地址大小取决于跳转类型为 near 或 far（通过在操作数前指定 near 或 far 操作数来指定）以及指令是否为 16 位或 32 位。对于 16 位指令 near 跳转操作数为 16 位，32 位指令为 32 位。16 位 far 跳转操作数大小为 32 位，32 位指令为 64 位。一个直接 jmp 指令 包括作为指令一部分的目的地址（可以包含 short，near 或 far 操作符），指定地址的操作数对于 near 或短跳转为数值表达式，对于 far 跳转为两个用冒号分隔的数值表达式。第一个指定段选择子，第二个为段中偏移。pword 操作符可强制为 32 位 far 调用，dword 强制为 16 位 far 调用。间接 jmp 指令间接从寄存器或指针变量中获取目的地址，操作数应为通用寄存器或内存。细节见 1.2.5 节。

```
jmp 100h        ; 直接 near 跳转
jmp 0FFFFh:0    ; 直接 far 跳转
jmp ax          ; 间接 near 跳转
jmp pword [ebx]; 间接 far 跳转
```

call 转移控制到过程，保存 call 后指令地址到堆栈，稍后将被 ret（返回）指令使用。操作数规则和 jmp 指令相同，但 call 没有直接种类因此它不是最优的。

ret，retn 和 retf 指令结束过程执行将转移控制给堆栈中 call 指令压入的地址。ret 等价于 retn，retn 从 near 调用过程返回，而 retf 从 far 调用过程返回。这些指令默认地址大小和当前代码设置适合，但也可以使用 retw，retnw 和 retfw 助记符来强制大小为 16 位，使用 retd，retnd 和 retfd 助记符强制大小为 32 位。这些指令可可选的指定一个立即数操作数，用它和堆栈指针相加，它的作用是在执行 call 指令之前移除调用程序压入堆栈的参数。

iret 返回控制到中断过程。它不同于 ret 的地方是它还将弹出堆栈中的标志到标志寄存器。这个标志是由中断机制保存的。它默认返回地址为当前代码设置，但也可以使用 iretw 或 iretd 助记符来强制使用 16 位或 32 位。

条件转移指令根据指令执行时 CPU 标志状态来决定是否转移控制。条件跳转助记符可以通过 j 助记符后面跟着表格 2.1 列出的条件助记符来得到，例如 jc 指令在 CF 设置时转移控制。条件跳转可以为 short 或 near，仅能直接跳转，并且能优化（见 1.2.5），操作数为指定目的地址的立即数。

loop 指令为使用 cx（或 ecx）中指定软循环次数的条件跳转。所有 loop 指令自动递减 cx（或 ecx），并且在 cx（或 ecx）为 0 时结束循环。使用 cx 还是 ecx 取决于当前代码设置为 16 位还是 32 为，但也可以使用 loopw 助记符强制使用 cx 或使用 loopd 助记符强制使用 ecx。loope 和 loopz 是相同指令，用作标准 loop，但它也在 ZF 为 1 时结束循环。loopew 和 loopzw 强制使用 cx 寄存器，looped 和 loopzd 强制使用 ecx 寄存器。loopne 和 loopnz 是相同指令，用作标准 loop，但它也在 ZF 为 0 时结束循环。loopnew 和 loopnzw 助记符强制使用 cx 寄存器，loopned 和 loopnzd 强制使用 ecx 寄存器。每一个 loop 指令都需要一个立即数值来指定目的地址，它只能为短调整（跟在 loop 指令前 128 字节和指令后 127 字节范围）。

jcxz 在 cx 值为 0 时跳到指定标号，jecxz 类似，但在 ecx 为 0 时跳到指定标号。操作数规则和 loop 指令类似。

int 激活操作数指定的中断服务过程，中断号范围在 0 到 255 之间。中断服务过程以 iret 指令结束，返回控制给 int 后的指令。int3 助记符为短格式的调用中断 3 的指令。into 指令当 OF 为 1 的话调用中断。

bound 检查指定寄存器中的符号值是否在指定范围内。如果不在这个范围将产生中断 5。它需要两个参数，第一个操作数为要测试的寄存器，第二个操作数为范围。操作数大小为 word 或 dword。

```
bound ax,[bx] ; 检查 word 数据边界
boudn eax,[esi]; 检查 dword 数据边界
```

助记符	测试条件	描述
o	OF=1	溢出
no	OF=0	不溢出
c		进位
b	CF=1	小于
nae		不大于
nc		不进位
ae	CF=0	不大于
nb		不小于
e	ZF=1	相等
z		零
ne	ZF=0	不相等
nz		不为 0
be	CF 或 ZF=1	小于或等于
na		不大于
a	CF 或 ZF=0	大于
nbe		不小于不等于
s	SF=1	有符号
ns	SF=0	<无符号/td>
p	PF=1	偶校验
pe		
np	PF=0	奇校验
po		
l	SF 异或 OF=1	小于
nge		不大于不等于
ge	SF 异或 OF=0	大于或等于
nl		不小于
le	(SF 异或 OF)或	小于或等于
ng	ZF=1	不大于
g	(SF 异或 OF)或	大于
nle	ZF=0	不小于不等于

表 2.1：条件

2.1.7 I/O 指令

in 从输入端口传输字节，字或双字到 al，ax，或 eax。I/O 端口可以用与指令一起编码的字节立即数直接寻址，或间接使用 dx 寄存器。目的操作数为 al，ax 或 eax 寄存器。源操作数应当为 0 到 255 之间的立即数，或 dx 寄存器。

```
in al,20h      ; 从端口 20h 输入字节
in ax,dx       ; 从 dx 寻址的端口输入字
```

out 传送字节，字，或者双字到 al，ax，或 eax 指定的输出端口。程序可以使用与 in 指令相同的方法指定端口号。目的操作数应当为 0 到 255 之间的立即数，或 dx 寄存器。源操作数应为 al，ax 或 eax 寄存器。

```
out 20h,ax     ; 输出字道端口 20h
out dx,al      ; 输出字节到 dx 寻址的端口
```

2.1.8 字符串操作指令

字符串操作针对字符串的一个元素。字符串元素可以为字节，字或双字。字符串元素用 si 和 di（或 esi 和 edi）寻址。每次字符串操作后 si 和或 di（或 esi 和或 edi）自动更新指向字符串中后一个元素。如果 DF（方向标志位）为 0，将递增索引寄存器，否则将递减。取决于字符串元素的大小在递增或递减大小为 1，2 或 4。每一个字符串操作指令都有不使用任何操作数的简短格式，在 16 位下使用 si 或和 di，在 32 位下使用 esi 或和 edi。si 和 esi 默认从 ds 段中定位数据，di 和 edi 默认从 es 段中定位数据。当字符串操作助记符后跟着指定字符串元素大小的字母时将使用短格式，“b”为字节元素，“w”为字元素，“d”为双字元素。字符串操作完整格式需要指定尺寸操作符和内存地址的操作数，操作数可以为跟有任何段前缀的 si 或 esi，di 或 edi 通常和 es 段前缀使用。

movs 传送 si（或 esi）指向的字符串元素到 di（或 edi）指向的地址。操作数尺寸可以为 byte，word 或 dword。目的操作数应当为 di 或 edi 寻址的内存，源操作数应当为跟着任何段前缀 si 或 esi 寻址的内存。

```
movs byte [di],[si]      ; 传送字节
movs word [es:di],[ss:si] ; 传送字
movsd                    ; 传送双字
```

cmps 用目的字符串元素减去源字符串元素并更新标志 AF，SF，PF，CF 和 OF，但它不会修改任何比较元素。如果字符串元素相当，ZF 设置为 1，否则为 0。第一个操作数为带

有任何段前缀的 si 或 esi 定位的源字符串元素，第二个操作数为 di 或 edi 定位的目的字符串。

```
cmpsb                    ; 比较字节
cmps word [ds:si],[es:di] ; 比较字
cmps dword [fs:esi],[edi] ; 比较双字
```

scas 用 al，ax，或 eax（取决于字符串元素的尺寸）减去目的字符串元素并更新标志 AF，SF，ZF，PF，CF 和 OF。如果值相等，ZF 将设置为 1，否则为 0.操作数因为 di 或 edi 定位的目的字符串元素。

```
scas byte [es:di]        ; scan 字节
scasw                    ; scan 字
scas dword [es:edi]      ; scan 双字
```

lods 载入字符串元素到 al，ax，或 eax。操作数因为带有任何段前缀的 si 或 esi 寻址的字符串元素。

```
lods byte [ds:si]        ; load 字节
lods word [cs:si]        ; load 字
lodsd                    ; load 双字
```

stos 将 al，ax，或 eax 的值放到目的字符串元素。字符串规则和 scas 指令相同。

ins 从 dx 寻址的输入端口传送一个字节，字或者双字到目的字符串元素。目的操作数应当为 di 或 edi 寻址的内存，源操作数应当为 dx 寄存器。

```
insb                     ; input 字节
ins word [es:di],dx      ; input 字
ins dword [edi],dx       ; input 双字
```

outs 传送源字符串元素到 dx 寄存器寻址的输出端口。目的操作数应当 dx 寄存器，源操作数应当为带有可带有任何段前缀的 si 或 esi 寻址的内存。

```
outs dx,byte [si]        ; output 字节
outsw                    ; output 字
outs dx,dword [gs:esi]   ; output 双字
```

重复前缀 rep，repe/repz，和 repne/repnz 指定重复字符串操作。当一个字符串操作指令包含重复前缀时，操作将重复执行，每一次将使用不同的字符串元素。当前缀指定的一个条件满足时结束重复。每次操作后所有 3 个前缀自动递减 cx 或 ecx 寄存器（取决于是否字符串操作指令使用 16 位或 32 位寻址），并且重复指定的操作指导 cx 或 ecx 为 0。repe/repz 和 repne/repnz 仅和 scas 和 cmps 指令使用（下面讲述的）。当使用这

些前缀时，取决于 ZF 标志重复后面的指令，此外，当 ZF 为 0 时 repe 和 repz 结束执行，当 ZF 为 1 时，repne 和 repnz 结束执行。

```
rep movsd               ；传送多个双字
repe cmpsb              ；比较字节直到不相等
```

2.1.9 标志控制指令

标志控制指令用来直接修改标志寄存器中的状态位。这节讲述的所有指令都没有操作数。

stc 设置进位标志 CF 为 1，clc 清零 CF，cmc 逆反 CF 的值。std 设置方向标志 DF 为 1，cld 清零 DF，sti 设置中断标志 IF 为 1 以允许中的，cli 清零 IF 以禁止中断。

lahf 拷贝 SF，ZF，AF，PF，和 CF 到 ah 寄存器的位 7，6，4，2，和 0。其余位将不受影响。 标志位保持不变。

sahf 将 ah 的位 7，6，2，和 0 传送到 SF，ZF，AF，PF，和 CF。

pushf 将 esp 值递减 2 或 4，压入低 16 位或 32 位的符号寄存器到堆栈，压入数据大小取决于当前代码设置。pushfw 强制压入 16 位，pushfd 强制压入 32 位。

popf 从栈顶弹出 16 位或 32 位数据到符号寄存器，然后递减 esp 值为 2 或 4，递减大小取决于当前代码设置。popfw 强制弹出 16 位，popfd 强制弹出 32 位。

2.1.10 条件操作指令

这些指令有 set 助记符，条件助记符（见表 2.1）组成，如果条件为 true 设置一个字节为 1 否则为置为 0。操作数必须为 8 位的通用寄存器或内存中字节。

```
setne al      ；如果 ZF 为 0 设置 al
seto byte [bx]；如果溢出设置 byte
```

salc 指令当 CF 为 0 时设置 al 的所有位为 1，否则都置为 0.这条指令没有参数。

cmov 助记符后面跟着条件助记符组成的指令，仅当条件满足时传送通用寄存器中的 word 或 dword 到通用寄存器。目的操作数必须为通用寄存器，源操作数可以为通用寄存器或内存。

```
cmove ax,bx     ；当 ZF 为 1 时传送
cmovnc eax,[ebx]；当 CF 为 0 时传送
```

cmpxchg 比较 al，ax 或 eax 和目的操作数。如果两个值相等，源操作数载入到目的操作数，否则目的操作数载入到 al，ax 或 eax 寄存器。目的操作数可以为通用寄存器或内存，源操作数必须为通用寄存器。

```
cmpxchg dl,bl    ；和寄存器比较并交换
cmpxchg [bx],dx  ；和内存比较并交换
```

cmpxchg8b 比较 edx 和 eax 组成的 64 位值和目的操作数比较。如果值相等，ecx 和 ebx 中 64 位值将保存到目的操作数。否则目的操作数值保存到 edx 和 eax 寄存器。目的寄存器必须为内存中的 qword。

```
cmpxchg8b [bx]  ；比较并交换 8 字节
```

2.1.11 其他指令

nop 指令占用一个字节但除了指令指针外没有任何作用。这条指令没有操作数，不会执行任何操作。

ud2 指令生成一个无效的指令异常。这条指令用作软件测试来显式字生成一个无效指令。这条指令没有操作数。

xlat 替换 al 寄存器字节为 bx 或 ebx 寻址的转换表中 al 索引的字节。操作数必须为可带有任何段前缀的 bx 或 ebx 寻址的内存中一个字节。这条指令有一个没有任何操作数的短格式 xlatb，它使用 ds 段寄存器中 bx 或 ebx（取决于当前代码设置）中的地址。

lds 转移源操作数中的指针变量到 ds 和目的寄存器。源操作数必须为内存操作数，目的寄存器必须为通用寄存器。ds 寄存器接受段选择子，目的寄存器接受指针偏移部分。
les，lfs，lgs 和 lss 操作和 lds 类似，只是它们分别使用 es，fs，gs 和 ss 寄存器，而不是 ds 寄存器。

```
lds bx,[si]      ; 载入指针到 ds:bx
```

lea 传输源操作数偏移（而不是值）到目的寄存器。源操作数必须为内存操作数，目的寄存器必须为同一寄存器。

```
lea dx,[bx+si+1] ; 载入有效地址到 dx
```

cpuid 返回处理器标识和特性信息到 eax，ebx，ecx 和 edx 寄存器。指令执行前 eax 寄存器为参数。该指令没有操作数。

pause 指令延迟指定时间执行下一条指令。它可用来提高死等效率。这条指令没有操作数。

enter 创建堆栈框架，可用作实现块结构高级语言的范围规则。leave 指令在过程结束后和过程开头的 enter 一起用来简化堆栈管理，并用作嵌套过程中控制访问变量。enter 指令有两个参数。第一个指定堆栈中要分配的动态存储字节大小。第二个参数为相应嵌套层数，范围为 0 到 31。指定层数决定了多少堆栈框架指针从前面一个框架中拷贝新的堆栈框架。堆栈框架通常叫做显示。显示的第一个 word（当代码为 32 位时为 dword）为最后的堆栈框架。这个指针允许 leave 指令通过废弃上一个堆栈帧来逆向前面的 enter 指令动作。当 enter 为过程创建一个新的显示后，通过递减 esp 为指定字节来分配需要的动态存储空间。允许过程寻址显示，enter 保留 bp（后 ebp）指向新堆栈框架。如果嵌套层数为 0，enter 压入 bp（或 ebp），拷贝 sp 到 bp（或 esp 到 ebp），然后 esp 递减第一个操作数大小。对于嵌套层数大于 0 的，处理器在调整堆栈指针前压入额外的框架指针。

```
enter 2048,0
```

2.1.12 系统指令

lmsw 载入操作数到机器状态字（CR0 的 0 到 15 位），而 smsw 保存机器状态字到目的操作数。这两条指令操作数可以为 16 位通用寄存器，对于 smsw 还可以为 32 位通用寄存器。

```
lmsw ax     ；从寄存器载入机器状态字
smsw [bx]   ；载入机器状态字到内存
```

lgdt 和 lidt 指令分别用来载入操作数中的值到全局描述表寄存器和中断描述表寄存器。sgdt 和 sidt 分别用来保存全局描述表或中断描述表寄存器到目的操作数。操作数必须为内存中的 6 个字节。

```
lgdt [ebx]  ；载入全局描述表
```

lldt 载入操作数到局部描述表寄存器的选择子，sldt 保存局部描述表寄存器段选择子到操作数。ltr 载入操作数到任务寄存器段选择子，str 保存任务寄存器选择子到操作数。操作数规则和 lmsw，smsw 指令相同。

lar 载入源操作数指定的选择子对应的段描述符访问权限到目的操作数，并设置 ZF 标志。目的操作数可以为 16 位或 32 为通用寄存器。源操作数必须为 16 位通用寄存器或内存。

```
lar ax,[bx] ；载入访问权限到 word
lar eax,dx  ；载入访问权限到 dword
```

lsl 从源操作数选择子指定的段描述符的段限制到目的操作数并设置 ZF 标志。操作数规则和 lar 指令相同。

verr 和 verw 检查操作数指定代码或数据段是否能以当前特权级上读或写。操作数必须为 word，可以为通用寄存器或内存。如果可用段并且可读（对于 verr）或可写（对于 verw），将设置 ZF 为 1，否者 ZF 置为 0。操作数规则和 lldt 指令相同。

arpl 比较两个段选择子的 RPL（请求特权级）。第一个操作数包含一个段选择子，第二个包含另一个。如果目的操作数 RTL 小于源操作数的 RPL，ZF 置为 1，否则置为 0，这条指令不影响目的操作数。目的操作数可以为 16 位通用寄存器或内存，源操作数必须为通用寄存器。

```
arpl bx,ax  ；调整寄存器选择子 RPL
arpl [bx],ax；调整内存选择子 RPL
```

clts 清零 CR0 寄存器的任务切换 TS 位。这条指令没有操作数。

lock 前缀导致处理器在执行该指令期间断言总线锁定信号，总线锁定信号保证处理器在信号断言期间独占使用任何共享内存。lock 前缀只能用在以下指令，并且目的操作数为内存：add，adc，and，btc，btr，bts，cmpxchg，cmpxchg8b，dec，inc，neg，not，

or，sbb，sub，xor，xadd 和 xchg。如果 lock 前缀和上面其中一指令使用并且源操作数为内存，可能会产生未定义指令异常。一个未定义指令异常也可能在 lock 和不在上面列出的指令一起使用的时候产生。xchg 指令常用来断言总线锁定信号无论是否使用 lock 前缀。

invlpg 无效（写）操作数指定的转换后援缓冲项 TLB。处理器定位这些地址包含的页并为这些页回写 TLB 项。

rdmsr 载入 64 位 MSR（model specific register）ecx 中的地址到 edx 和 eax。wrmsr 写 edx 和 eax 到 ecx 寄存器指定的 64 位 MSR。rdtsc 从 64 位 MSR 载入当前处理器时间戳到 edx 和 eax 寄存器。处理器每一时钟周期递增 MSR 时间戳，每次处理器重置时重置时间戳为 0。rdpmc 载入 edx 寄存器指定的 40 位性能监视计数器到 edx 和 eax。这些指令没有操作数。

wbinvd 回写处理器内部缓冲中所有修改的缓冲行到主内存，并且无效内部缓冲。然后创建一个特殊函数总线周期来指导外部缓冲也回写外部修改数据以及另一个时钟周期来标识外部缓冲无效。这条指令没有操作数。

rsm 从系统管理模式返回到当处理器接受 SMM 中断时所处的模式。这条指令没有操作数。

sysenter 执行到 ring 0 系统过程的快速调用，sysexit 执行到 ring 3 的快速返回。这些指令是否可用有 MSR 相关位标识。这些指令没有操作数。

2.1.13 FPU 指令

浮点单元 FPU 指令操作三种格式的浮点数据：单精度（32 位），双精度（64 位）和扩展双精度（80 位）。FPU 寄存器构成一个堆栈，并且它们都是扩展双精度的。当从堆栈中压入或弹出一些值时，FPU 寄存器移动，所以 st0 一直为 FPU 堆栈栈顶的值，st1 为栈顶下的第一个值。st0 和 st 是同义词。

fld 压入浮点数据到 FPU 寄存器堆栈。操作数可以为 32 位，64 位或 80 位内存地址或 FPU 寄存器，其值将稍后载入到 FPU 寄存器堆栈栈顶（也就是 st0 寄存器），并且自动转换为扩展双精度格式。

```
fld dword [bx]      ; 从内存载入单精度浮值。
fld st2             ; 压入 st2 的值到寄存器堆栈
```

fld2，fldz，fldl2t，fldl2e，fldpi，fldlg2 和 fldln2 载入常用的常量到 FPU 寄存器堆栈。载入的常量分别为：+1.0，+0.0，log2|10，log2|e，pi，log10|2 和 ln2。这些指令没有操作数。

fild 转换一个源操作数整数为扩展双精度浮点格式，并将结果压入 FPI 寄存器堆栈。源操作数可以为 16 位，32 位，或 64 位内存地址。

```
fild qword [bx]     ; 从内存载入 64 位整数
```

fst 拷贝 st0 寄存器的值到目的操作数，目的操作数可以为 32 位或 64 位内存地址或另一个 FPU 寄存器。fstp 执行和 fst 相同的操作，只是它还将弹出寄存器堆栈。fstp 执行和 fst 相同的操作，只是它还将压入一个 80 位内存中的值。

```
fst st3             ; 拷贝 st0 值到 st3 寄存器
fstp tword [bx]     ; 存储内存中值并弹出堆栈
```

fist 转换 st0 值为一整数，并保存结果到目的操作数。操作数可以为 61 位或 32 位内存地址。fistp 执行相同操作，但很将弹出寄存器堆栈，并能存储值到 64 位内存，操作数规则和 fild 指令相同。

fbld 转换压缩 BCD 整数为扩展双精度浮点格式并压入值到 FPU 堆栈。fbstp 转换 st0 中的值为 18 个数字压缩 BCD 整数，保存结果到目的操作数并弹出寄存器堆栈。操作数应为 80 位内存地址。

fadd 计算目的和源操作数的和并保存结果到目的操作数。目的操作数一直为 FPU 寄存器，如果源操作数为内存地址，目的操作数为 st0 寄存器并且只指定源操作数。内存操作数可以为 32 位或 64 位值。

```
fadd qword [bx]     ; 计算扩展双精度和 st0 的和
```

```
fadd st2,st0        ; 计算 st0 和 st2 的和
```

faddp 计算目的和源操作数的和，并保持结果到目的位置，然后弹出堆栈。目的操作数必须为 FPU 寄存器，源操作数必须为 st0。当没有指定操作数时，将使用 st1 作为目的操作数。

```
faddp               ; 计算 st0 和 st1 的和并弹出堆栈
faddp st2,st0       ; 计算 st0 和 st2 的和并弹出堆栈
```

fiadd 指令转换源操作数整数为扩展双精度浮点数，并和目的操作数相加。操作数必须为 16 位或 32 位内存地址。

```
fiadd word[bx]      ; word 整数和 st0 相加
```

fsub，fsubr，fmul，fdiv，fdivr 指令和 fadd 类似，操作数规则和 fadd 相同。fsub 计算目的操作数和源操作数的差，fsubr 计算源操作数和目的操作数的差，fmul 将目的和源操作数相乘。fdivr 计算目的操作数和源操作数的差，fdivr 计算源操作数和目的操作数的差。fsubp，fsubrp，fmulp，fdivp，fidivr 在转换源操作数整数为浮点数据后执行这些操作，它们操作数的规则和 fiadd 指令相同。

fsqrt 计算 st0 寄存器中值的开方。fsin 计算值的 sin，fabs 清除符号位来得到绝对值，frndint 根据当前四舍五入模式来得到最接近的整数值。f2xm1 计算 2 的以 st0 为幂的指数，并减去 1.0，st0 的值的范围必须在-1.0 和+1.0 之间。所有这些指令保存结果到 st0 并且没有操作数。

fsincos 计算 st0 值的 sin 和 cos，保存 sin 结果到 st0，压入 cos 值到 FPU 寄存器堆栈。fptan 计算 st0 的 tag 值，保存结果到 st0，并压入值 1.0 堆栈。fpatan 计算 st1 的 arctag，并和 st0 相除，保存结果到 st1 并弹出寄存器堆栈。fyl2x 计算 st0 的二进制算术结果，乘以 st1 值，保存结果到 st1，然后弹出 FPU 寄存器堆栈。fyl2xp1 执行相同操作，但它在计算对数前和 1.0 相加，保存结果到 st0。fprem 计算 st0 和 st1 相除的余数到 st1，结果到 st0。fprem1 执行和 fprem 相同的操作，但它计算 IEEE 标志 754 指定的余数。fscale 截去 st1 的值并和 st0 值相加。fxtract 分隔 st0 值为指数和有效数字，保存指数到 st0，压入有效数字到寄存器堆栈。fnop 不执行任何操作。这些指令没有操作数。

fxch 交换 st0 和另一个 FPU 寄存器的内容。这个操作数必须为 FPU 寄存器，不用指定操作数，st0 和 st1 内容将被交换。

fcom 和 fcomp 比较 st0 和源操作数，并格局结构设置 FPU 状态字标志。fcomp 执行操作后还将弹出寄存器堆栈。操作数可以为内存中单精度或双精度浮点或 FPU 寄存器。当没有指定源操作数时将使用 st1.

```
ficom word [bx]     ; 16 位整数和 st0 比较
```

fcomi，fcomip，fucomi，fucomip 用 st0 和另一个 FPU 寄存器比较并根据结果设置标志 ZF，PF 和 CF。fcomip 和 fucomip 还将在执行操作后弹出寄存器堆栈。fcmov 助记符后面跟着表 2.2 列出的 FPU 条件助记符组成的指令如果给定测试条件为 true 时传送指定 FPU 寄存器到 st0 寄存器。这些指令有两种不同语法，一种是跟着指定源 FPU 寄存器的单一操作数，另一种带有两个操作数，此时目的操作数为 st0，第二个操作数为源 FPU 寄存器。

```
fcomi st2           ; 比较 st0 和 st2 并设置标志
fcmovb st0,st2      ; 如果小于传送 st2 到 st0
```

助记符	测试条件	描述
b	CF=1	小于
e	ZF=1	等于
be	CF 或 ZF=1	不大于
u	PF=1	无序的
nb	CF=0	不小于
ne	ZF=0	不相等
nbe	CF 且 ZF=0	大于
nu	PF=0	有序的

表 2.2：FPU 条件

ftst 比较 st0 和 0.0 并根据结果设置 FPU 状态字标志。fxam 检查 st0 内容并设置 FPU 状态字来标识该寄存器值类型。这些指令没有操作数。

fstsw 和 fnstsw 保存当前 FPU 状态字到目的位置。目的操作数可以为 16 位内容或 ax 寄存器。fstsw 在保持状态字前检查未知的没有屏蔽的 FPU 异常，而 fnstsw 不这么做。

fstcw 和 fnstcw 保存当前 FPU 状态字到指定的内存中目的地址。fstcw 在保持状态字前检查未知没有屏蔽的 FPU 异常，而 fnstcw 不这样。fldcw 载入操作数到 FPU 控制字。操作数必须为 16 位内存地址。

fstenv 和 fnstenv 保存当前 FPU 操作环境到目的操作数指定的内存地址，然后屏蔽所有 FPU 异常。fstenv 在处理前检查待处理的未屏蔽的 FPU 异常，fnstenv 将不检查。flden 从内存中载入完整的操作环境到 FPU。fsave 和 fnsave 保存当前 FPU 状态（操作环境和寄存器堆栈）到内存中定制的目的地址并重新初始化 FPU。fsave 在处理前检查待处理的非屏蔽 FPU 异常，fnsave 不检查。frstor 从指定内存位置载入 FPU 状态。所有这些指令都需要一个内存位置操作数。

finit 和 fninit 设置 FPU 操作环境到默认状态。finit 在处理前检查待处理非屏蔽 FPU 异常，而 fninit 不检查。fclex 和 fnclex 清除 FPU 状态字中 FPU 异常标志。fclex 在处理前检查

待处理非屏蔽 FPU 异常，fnclex 不检查。wait 和 fwait 为相同指令，将导致处理器检查待处理的非屏蔽 FPU 异常并在处理前处理它们。这些指令没有操作数。

ffree 设置和指定 FPU 寄存器相关的 tag 为 0。操作数必须为一个 FPU 寄存器。

fincstp 和 fdecstp 翻转 FPU 堆栈为 1 或栈顶指针减 1.这些指令没有操作数。

2.1.14 MMX 指令

MMX 指令 操作压缩整数或 MMX 寄存器，MMX 寄存器为 80 位 FPU 寄存器的低 64 位。因为此 MMX 指令不能和 FPU 指令一起使用。他们可以操作压缩字节（八个 8 位整数），压缩字（四个 16 位整数）或压缩双字（两个 32 位整数），使用压缩格式允许一次对多个数据执行操作。

movq 从源操作数拷贝 8 字节到目的操作数。至少一个操作数必须为 MMX 寄存器，第二个可以为 MMX 寄存器或 64 位内存地址。

```
movq mm0,mm1      ; 寄存器到寄存器移动 8 字节
movq mm2,[ebx]    ; 内存到寄存器移动 8 字节
```

movd 从源操作数移动双字到目的操作数。其中一个操作数必须为 MMX 寄存器，第二个可以为通用寄存器或 32 位内存地址。只使用 MMX 寄存器的低双字。

所有通用 MMX 操作有两个操作数，目的操作数应当为 MMX 寄存器，源操作数可以为 MMX 寄存器或 64 位内存地址。对源和目的操作数执行相应操作并保存数据单元到目的操作数。paddb，paddw 和 paddd 计算压缩字节，压缩字，压缩双字的和。paddsb，paddsw，psubsb 和 psubsw 执行压缩字节或压缩字的带符号 saturation 的和。paddusb，paddusw，psubusb，psubusw 类似，但将计算无符号 saturation。pmulhw 和 pmullw 符号乘压缩字，保存结果的高位或低位到目的操作数。pmaddwd 乘压缩字，加上四个立即双字对来生成压缩双字结果。pand，por 和 pxor 执行 qword 逻辑操作。pcmpeqb，pcmpeqw 和 pcmpeqd 比较压缩字节，压缩字或压缩双字是否相等。如果某对数据元素相等，目的操作数中相应数据元素将填充为 1，否则填充 0.pcmpgtb，pcmpgtw 和 pcmpgtd 执行相同操作，但它们用来检查是否目的操作数中数据元素大于源操作数中数据元素。packsswb 转换带压缩符号字为压缩带符号字节，使用 saturation 来处理溢出。packuswb 转换压缩符号字道压缩无符号字节。源操作数中转换后的数据单元存储到目的操作数的低部分，目的操作数中转换后的数据单元存储到高半部分。punpckhbw，punpckhwd 和 punpckhdq 从源操作数和目的操作数高半部分插入数据单元并保持结果到目的操作数。punpcklbw，punpcklwd 和 punpckldq 执行相同操作，但它们使用源和目的操作数的低半部分。

```
paddsb mm0,[esi]   ; 计算压缩字节符号 saturation 和
pcmpeqw mm3,mm7    ; 比较压缩字是否相当
```

psllw，pslld 和 psllq 对压缩字，压缩双字或目的操作数中的一个 qword 执行逻辑左移，左移位数由源操作数指定。pswlw，psrld 和 psrlq 对压缩字，压缩双字或目的操作数中的一个 qword 执行逻辑右移。psraw 和 psrad 对压缩字或双字执行算术右移。目的操作数因为 MMX 寄存器，而源操作数可以为 MMX 寄存器，64 位内存为孩子，或 8 位立即数。

```
psslw mm2,mm4       ; 逻辑左移 word
psrad mm4,[ebx]     ; 算术右移 dword
```

emms 是得 FPU 寄存器可用。如果使用了 MMX 指令，它必须在使用 FPU 指令前使用。

2.1.15 SSE 指令

SSE 扩展增加了更多 MMX 指令，并且能操作压缩单精度浮点数。128 位压缩单浮点格式由 4 个单精度浮点数组成。128 位 SSE 寄存器设计用来操作这种数据类型。

movapshemovups 传送源操作数中一个包含单进度值的双 qword 操作数到目的操作数。至少一个操作数必须为 SSE 寄存器，第二个操作数可以为 SSE 寄存器或 128 位内存地址。movaps 指令的内存操作数必须对齐在 16 位字节边界，movups 指令操作数不需要对齐。

```
movups xmm0,[ebx]   ; 传送未对其双 qword
```

movlps 在内存和 SSE 寄存器低 qword 之间移动两个压缩单精度数据。movhps 在内存和 SSE 寄存器高 qword 之间移动两个压缩单精度数据。其中一个操作数必须为 SSE 寄存器，另一个必须为 64 位内存地址。

```
movlps xmm0,[ebx]   ; 移动内存到 xmm0 低 qword
movhps [esi],xmm7   ; 移动 xmm7 高 qword 到内存
```

movlhps 从源寄存器的低 qword 移动压缩的两个浮点数据到目的寄存器。movhlps 从源寄存器高 qword 移动两个压缩单浮点数到目的寄存器的低 qword。这两个操作数都必须为 SSE 寄存器。

movmskps 传送 SSE 寄存器中 4 个单浮点数据的最高位到一个通用寄存器的低 4 位。源操作数必须为 SSE 寄存器，目的操作数必须为通用寄存器。

movss 在源和目的操作数（只传送低 dword）传送单浮点数据。至少一个操作数必须为 SSE 寄存器，第二个操作数可以为 SSE 寄存器或 32 位内存地址。

```
movss [edi],xmm3    ; 移动 xmm3 低 dword 到内存
```

每一个 SSE 算术操作都有两种。当助记符以 ps 结尾时，源操作数可以为 128 位内存地址或 SSE 寄存器，目的操作数必须为 SSE 寄存器，操作压缩的四个浮点数据，对于对应数据元素对，结果保存在目的寄存器。当助记符以 ss 结尾时，源操作数可以为 32 位内存地址或 SSE 寄存器，目的操作数必须为 SSE 寄存器，操作于单浮点数据，此时只适用 SSE 寄存器的低 dword。addps 和 addss 计算和，mulps 和 mulss 计算积，divps 和 divss 计算目的值和源值的商，rcpps 和 rcpss 计算源操作数的近似倒数，sqrtps 和 sqrtss 计算源

操作数的开放，rsqrtps 和 rsqrtss 计算源值的开方的近似倒数，maxps 和 maxss 比较源和目的值并返回大的值，minps 和 minss 计算源和目的值并返回小的值。

```
mulss xmm0,[ebx]   ；乘以单浮点数据
addps xmm3,xmm7    ；加上压缩单浮点数据
```

andps，andnps，orps 和 xorps 对压缩单精度数据执行逻辑操作。源操作数可以为 128 位内存地址或 SSE 寄存器，目的操作数必须为 SSE 寄存器。

cmpps 比较压缩单精度数并返回结果掩码到目的操作数，目的操作数只能为 SSE 寄存器。源操作数可以为 128 位诶从地址或 SSE 寄存器，第三个参数必须为表 2.3 中列出的 8 个比较条件操作数立即数。cmpss 对单浮点数据执行相同的操作，但它只影响目的寄存器的低 dowrd，此时源操作数可以为 32 位内存地址或 SSE 寄存器。这两个指令也包含只有两个操作数和条件编码的助记符。这些助记符有 cmp 助记符后跟着表 2.3 列出的助记符，以及 ps 或 ss 构成。

```
cmpps xmm2,xmm4,0  ；比较压缩单精度值
cmpltss xmm0,[ebx] ；比较单精度数据
```

comiss 和 ucomiss 比较单精度并设置标志 ZF，PF 和 CF 来表示结果。目的操作数必须为 SSE 寄存器，源操作数可以为 32 位内存地址或 SSE 寄存器。

代码	助记符	描述
0	eq	相等
1	lt	小于
2	le	小于后等于
3	unord	无序
4	neq	不等
5	nlt	不小于
6	nle	不小于不等于
7	ord	有序

shufps 从目的操作数移动任何两个四单精度数据到目的操作数的低 qword，源操作数中 4 个值的任何两个到目的操作数的高 qword。目的操作数必须为 SSE 寄存器，源操作数可以为 128 位内存地址或 SSE 寄存器，第三个操作数必须 8 位立即数来指定选择移动那些数据到目的操作数。位 0 和 1 选择移动目的操作数到结果的低 dword，位 2 和 3 移动目的操作数到第二个 dword，位 4 和 5 移动源操作数的到结果的第三个 dword，位 6 和 7 移动源操作数到结果的高 dword。

```
shufps xmm0,xmm0,10010011b ; 搅乱 dword
```

unpckhps 执行从源和目的操作数高部分插入的未压缩数据，并保存结果到目的操作数。源操作数可以为 128 位内存地址或 SSE 寄存器。unpcklps 执行从源和目的操作数低部分插入的未压缩数据，并保持结果到目的操作数，操作数规则相同。

cvtpi2ps 转换压缩的 2dword 整数到压缩的 2 单浮点数据，并保存结果到目的操作数的低 qword，目的操作数应为 SSE 寄存器。源操作数可以为 64 位内存地址或 MMX 寄存器。

```
cvtpi2ps xmm0,mm0          ; 整合为单精度数
```

cvtsi2ss 转换 dword 整数位单精度浮点数并保存结果到目的操作数的低 dword，目的操作数必须为 SSE 寄存器。源操作数可以为 32 位内存地址或 32 位通用寄存器。

```
ctsi2ss xmm0,eax           ; 整合为但精度数
```

cvtps2pi 转换 2 单精度浮点数为压缩 2dword 整数并保存结果到目的操作数，目的操作数必须为通用寄存器。源操作数可以为 64 位内存地址或 SSE 寄存器，只适用 SSE 寄存器的低 qword。cvttps2pi 操作结果类似，除了截去近似为整数，操作数规则相同。

```
cvtps2pi mm0,xmm0          ; 单精度数到整数
```

cvtss2si 转换 2 单精度浮点数为压缩 2dword 整数并保存结果到目的操作数，目的操作数必须为 32 位通用寄存器。源操作数可以为 32 位内存地址或 SSE 寄存器，只适用 SSE 寄存器的低 qword。cvttss2pi 操作结果类似，除了截去近似为整数，操作数规则相同。

```
cvtss2pi eax,xmm0          ; 单精度数到整数
```

pextrw 拷贝第三个操作数指定的源操作数 word 到目的操作数。源操作数必须为 MMX 寄存器，目的操作数必须为 32 位通用寄存器（仅影响低 word），第三个操作数必须为 8 位立即数。

```
pextrw eax,mm0,1           ; 取 word 到 eax
```

pinsrw 插入第三个操作数指定的 word 到目的操作数中第三个操作数指定的位置，第三个操作数必须为 8 位立即数。目的操作数必须为 MMX 寄存器，源操作数可以为 16 位内存地址或 32 位通用寄存器（只适用寄存器的低 word）。

```
pinsrw mm1,ebx,2           ; 从 ebx 插入 word
```

pavgb 和 pavgw 计算压缩字节或字平均值。pmaxub 返回压缩无符号字节的最大值，pminub 返回压缩无符号字节的最小值，pmaxsw 返回压缩无符号字的最大值，pminsw 返回压缩无符号字的最小值。pmulhuw 执行无符号压缩字乘法并保存结果到目的操作数的高 word。psadbw 计算压缩无符号字节绝对差别，汇总不同点，并保存汇总到目的操作数的低 word。所有这些指令操作数规则和上一节讲述的 MMX 操作相同。

pmovmskb 创建源操作数每一个字节的自高位掩码，并保存结果到目的操作数的低 byte。源操作数必须为 MMX 寄存器，目的操作数必须为 32 位通用寄存器。

pshufw 插入 word 源操作数到目的操作数中第三个操作数指定的位置。目的操作数必须为 MMX 寄存器，源操作数可以为 64 位内存地址或 MMX 寄存器，第三个操作数必须 8 位立即数用来选择那些值将移动到目的操作数，和 shufps 指令第三个操作数相同方式。

ovntq 使用非临时缓冲提示以最小缓冲损失方式从源操作数移动 qword 到内存。源操作数必须为 MX 寄存器，目的寄存器应为 64 位内存地址。movntps 使用非临时提示从 SSE 寄存器保存压缩单精度数据到内存。源操作数必须为 SSE 寄存器，目的操作数必须为 128 位内存地址。maskmovq 使用非临时提示方式保存第一个操作数指定的字节到 64 位内存地址。两个操作数都必须为 MMX 寄存器，第二个操作数决定源操作数中那些字节将写到内存中。内存地址由 DS 段中 DI（或 EDI）寄存器指向。

prefetcht0，prefetcht1，prefetcht2 和 prefetchnta 获取操作数指定的字节所处内存数据行到指定位置。操作数应为 8 位内存地址。

sfence 同步所有在它之前所有创建指令操作。这条指令没有操作数。

ldmxcsr 载入 32 位内存操作数到 MXCSR 寄存器。stmxcsr 保存 MXCSR 内容到 32 位寄存器。

fxsave 保存 FPU，MXCSR 寄存器当前状态，和所有 FPU 和 SSE 寄存器内容到 512 字节目的操作数指定的内存地址。fxrstor 重新载入前面用 fxsave 指令保存的 512 字节内存地址。这两条指令内存操作数必须对齐在 16 字节边界上，它不能声明任何指定大小操作数。

2.1.16 SSE2 指令

SSE2 扩展用来操作压缩双精度浮点数据，扩展 MMX 指令语法，并且增加了新的指令。

movapd 和 movupd 从源操作数传送包含压缩双精度数据的双 qword 操作数到目的操作数。这些指令类似 movaps 和 movups，操作数规则也相同。

movmskpd 传送 SSE 寄存器两个双精度数最高位到通用寄存器低两位。这条指令和 movmskps 类似并有相同操作数规则。

movsd 在源和目的操作数之间传送双精度数（值传送低 qword）。其中至少一个操作数为 SSE 寄存器，第二个可以为 SSE 寄存器或 64 位内存地址。

双精度值算术操作有：
addpd，addsd，subpd，subsd，mulpd，mulsd，divpd，divsd，sqrtpd，sqrtsd，maxpd，maxsd，minpd，minsd，它们和上一节讲述的单浮点算术操作类似。当助记符以 pd 而不是 ps 结尾时，操作针对于压缩的 2 双精度数，但操作数规则相同。当助记符以 sd 而不是 ss 结尾时，源操作数可以为 64 位内存地址或 SSE 寄存器，目的寄存器必须为 SSE 寄存器并且操作于双精度数，此时只适用 SSE 寄存器的低 qword。

andpd，andnpd，orpd 和 xorpd 对压缩双精度值执行逻辑操作。它们和针对单精度的逻辑操作类似并且有相同的操作数规则。

cmppd 比较压缩双精度数并返回掩码结果到目的操作数。这条指令和 cmpps 类似，并且有相同的操作数规则。cmpsd 对双精度数据执行相同操作，但它只影响目的寄存器的低 qword。接受两个操作数的指令由 cmp 助记符，表 2.3 列出的条件助记符和 pd 或 sd 组成。

comisd 和 ucomisd 比较双精度操作数并设置标志 ZF，PF 和 CF 来表示结果。目的操作数必须为 SSE 寄存器，源操作数可以为 128 位内存地址或 SSE 寄存器。

shufpd 从目的操作数移动任何两个双精度数到目的操作数的低 qword，源操作数任何两个值值到目的寄存器的高 qword。这条指令和 shufps 类似并且有相同的操作数规则。第三个操作数位 0 指定将移动到目的操作数的值，位 1 选择将从源操作数移动的值，其他位为保留的必须为 0。

unpckhpd 在源和目的操作数之间执行压缩高 qword，unpcklpd 在源和目的操作数之间执行未压缩低 qword。它们是 unpckhps 和 unpcklps 类似，并且有相同的操作数规则。

cvtps2pd 转换压缩 2 单精度浮点数据为两个压缩的双精度浮点数据，目的操作数必须为 SSE 寄存器，源操作数可以为 64 位内存地址或 SSE 寄存器。 cvtpd2ps 转换压缩 2 扩展双精度浮点数据为压缩 2 单精度浮点数据，目的操作数必须为 SSE 寄存器，源操作数可以为 128 位内存地址或 SSE 寄存器。cvtss2sd 转换单精度浮点数据为双精度浮点数据，目

的操作数必须为 SSE 寄存器，源操作数可以为 32 位内存地址或 SSE 寄存器。cvtsd2ss 转换扩展双精度数据为单精度浮点数据，目的操作数必须为 SSE 寄存器，源操作数可以为 64 位内存地址或 SSE 寄存器。

cvtpi2pd 转换压缩 2 dword 整数位压缩双精度浮点数据，目的操作数必须为 SSE 寄存器，源操作数可以为 64 位内存地址或 MMX 寄存器。cvtsi2sd 转换一个 dword 整数为双精度浮点数据，目的操作数必须为 SSE 寄存器，源操作数可以为 32 位内存地址或 32 位通用寄存器。cvtpd2pi 转换压缩双精度浮点数据为压缩 2 dword 整数，目的操作数应为 MMX 寄存器，源操作数可以为 128 位内存地址或 SSE 寄存器。cvttpd2pi 执行类似操作，除了它将源值截断到整数，操作数规则也一样。cvtsd2si 转换双精度浮点数据为 dword 整数，目的操作数应为 32 位通用寄存器，源操作数可以为 64 为内存地址或 SSE 寄存器。cvttsd2si 执行相同吃哦啊在，除了将源值截为整数，操作数规则也一样。

cvtps2dq 和 cvttps2dq 转换压缩单精度浮点数据为压缩 4 dword 整数，保存它们的值到目的操作数。cvtpd2dq 和 cvttpd2dq 转换压缩双精度浮点数据为压缩 2 dword 整数，保存结果到目的操作数的低 qword。cvtdq2ps 转换压缩 4dword 帧数为压缩单精度浮点数据。cvtdq2pd 从源操作数低 qword 转换压缩 2 dword 整数为压缩双精度浮点数据。所有这些指令目的操作数必须为 SSE 寄存器，源操作数可以为 128 位内存地址或 SSE 寄存器。

movdqa 和 movdqu 传送源操作数中双 qword 大小的压缩整数为目的操作数。至少其中一个操作数必须为 SSE 寄存器，第二个可以为 SSE 寄存器或 128 位内存地址。movdqa 指令内存操作数必须 16 字节对齐，movdqu 指令操作数不需要对齐。

movq2dq 移动 MMX 源寄存器内容到目的 SSE 寄存器低 qword。movdq2q 传送源 SSE 寄存器低 qword 到目的 MMX 寄存器。

```
movq2dq xmm0,mm1    ; MMX 寄存器传送到 SSE 寄存器
movdq2q mm0,xmm1    ; SSE 寄存器传送到 MMX 寄存器
```

所有 MMX 指令操作的 64 位压缩整数（用 p 开头的助记符）扩展能操作 SSE 寄存器中 128 位压缩整数。left to do。pshufw 指令另外，它不需要扩展语法，但有两种新的变种：pshufhw 和 pshuflw，他们只允许扩展语法，并且分别针对操作数高或低 qword 执行和 pshufw 相同操作。此外 pshufd 为新增指令，用来执行和 pshufw 相同的操作，但它操作 dword 而不是 word，它只允许扩展语法。

```
psubb xmm0,[esi]    ; 减 16 压缩字节
pextrw eax,xmm0,7   ; 提取最高 word 到 eax
```

paddq 执行两个压缩 qword 的和，psubq 执行两个压缩 qword 的差，puludq 执行无符号乘法每一个对应 qword 的低 dword，并返回结果到压缩 qword。这些指令和 2.1.14 讲述的通用 MMX 操作有相同规则。

pslldq 和 psrldq 执行逻辑左或右移双 dqword 目的操作数，移动位数由源操作数指定。目的操作数必须为 SSE 寄存器，源操作数应为 8 位立即数。

punpckhqdq 源操作数高 qword 和目的操作数高 qword，并将结果写到目的 SSE 寄存器中。punpcklqdq 插入源操作数低 qword 和目的操作数低 qword，并将结果写到目的 SSE 寄存器中。源操作数可以为 128 位内存地址或 SSE 寄存器。

movntdq 使用非临时提示从 SSE 寄存器保存压缩整数数据到内存。源操作数应为 SSE 寄存器，目的操作数应为 128 位内存地址。movntpd 使用非临时提示从 SSE 寄存器保存压缩双精度数据到内存。源操作数应为 32 位通用寄存器，目的操作数应为 32 位内存地址。maskmovdqu 使用非临时提示从第一个参数保存选择位到 128 位内存地址。这两条指令操作数都应为 SSE 寄存器，第二个操作数选择了那些字节将从源操作数写到目的操作数。内存地址有 DS 段中 DI（或 EDI）寄存器指定，不需要对齐。

clflush 写并且无效指定操作数地址字节的缓冲行，指定操作数必须为 8 位内存位置。

lfence 执行载入同步。mfence 执行访问同步。所以它组合了 sfence（上一节讲述的）和 lfence 指令功能。这些指令没有任何操作数。

2.1.17 SSE3 指令

Prescott 技术发明了新的指令来提高 SSE 和 SSE2 性能 - 称为 SSE3。

fisttp 行为和 fistp 指令相似，并且允许相同操作数，唯一区别是它总是会截操作，无论当前的舍入模式。

movshdup 载入目的操作数为原值同样尺寸用两个重复的高 dword 填充每一个 qword 的 128 位值。movsldup 执行相同动作，除了它拷贝低 dword。目的操作数应为 SSE 寄存器，源操作数可以为 SSE 寄存器或 128 位内存地址。

movddup 载入 64 为源值，赋值它到目的操作数的高和低 qword。目的操作数应当为 SSE 寄存器，源操作数可以为 SSE 寄存器或 64 位内存地址。

lddqu 是和 movdqu 执行等价功能的指令，但能在源操作数跨缓冲行边界时提高性能。目的操作数必须为 SSE 寄存器，源操作数必须为 128 位内存地址。

addsubps 执行第二和第四组单精度和，第一和第三组单精度差。addsupd 执行第二组双精度和，第一组双精度差。haddps 执行源和目的操作数每个 qword 的两个单精度和，保存结果到目的操作数低 qword，源操作数结果到目的操作数高 qword。haddpd 对每个操作数执行两个双精度值和，并保存目的操作数中结果到目的操作数的低 qword，源操作数结果到目的操作数高 qword。所有这些指令都需要 SSE 寄存器为目的操作数，源操作数可以为 SSE 寄存器或 128 位内存地址。

monitor 创建回写地址行监视。它需要三个有顺序的操作数 EAX，ECX 和 EDX。mwait 等待回写到 monitor 创建的地址区域。它使用带有额外参数的两个操作数，第一个为 EAX，第二个为 EDX。

2.1.18 AMD 3DNOW!指令

3DNow!扩展增加新的 2.1.14 列出 MMX 指令，并且能操作 64 位压缩浮点数据，每一个有两个单精度浮点数据组成。

这些指令规则和通用 MMX 操作相同，目的操作数必须为 MMX 寄存器，源操作数可以为 MMX 寄存器或 64 位内存地址。pawgusb 计算压缩无符号字节平均值。pmulhrw 执行带符号压缩字的乘积，舍入每一个 dword 结果高半部分到目的操作数。pi2fd 转换压缩 dword 整数为压缩浮点数。pf2id 使用舍入转换压缩浮点数据为压缩 dword 整数。pi2fw 转换压缩字整数到压缩浮点数，只是哟个源操作数中每个 dword 的低 word。pf2iw 转换压缩浮点数据为压缩 word 整数，结果使用符号扩展扩展为压缩浮点数。pfadd 计算压缩浮点数的和。pfsub 和 pfsubr 计算压缩浮点数的差，第一个用目的值减去源值，第二个用源值减去目的值。pfmul 计算压缩浮点数的积。pfacc 计算目的操作数地和高浮点数的和，保存结果到目的操作数的低 dword，并且计算源操作数的低和高 dword 的和，保存结果到目的寄存器的高 dword。pfnacc 用目的操作数的高浮点数减去低浮点数，保存结果到目的操作数的低 dword，并结算源操作数的高和低 dword 的差，保存结果到目的操作数的高 dword。pfpnacc 用目的操作数的高浮点数据减去低浮点数据，保存结果到目的操作数的低 dword，并计算源操作数低和高浮点数的和，保存结果到目的操作数的高 dword。pfmax 和 pfmin 计算浮点数的最大和最小值。pswapd 翻转源操作数的高低 dword。pfrcp 返回原操作数的近似浮点值倒数。pfrsqrt 返回原操作数的开方的近似倒数。pfrcpit1 执行第一步 Newton-Raphson 迭代开方。pfrcpit2 计算第二步 Newton-Raphson 迭代开发。pfcmpeq，pfcmpge 和 pfcmpgt 比较压缩浮点数并根据比较结果设置目的操作数中相应数据元素的位（全部置为 1 或置为 0），第一个检查值是否相等，第二个检查目的值是否大于或等于源之，第三个检查是否目的值大于源值。

prefetch 和 prefetchw 从内存载入 包含操作数指定字节的数据行，prefetchw 指令必须当缓冲行中数据被修改时使用，否则应使用 prefetch 指令。操作数必须为 8 位内存地址。

femms 执行快速清除 MMX 状态。它没有操作数。

2.1.19 x86-64 长模式指令

AMD64 和 EM64T 体系（我们将使用 x86-64 作为通用名称）扩展 x86 指令集以用作 64 位处理。而原始和兼容模式使用相同的寄存器和指令集。新的长模式扩展 x86 操作 64 位，并且发明了一个新的寄存器。你可以使用 use64 伪指令来生成这个模式的代码。

每一个通用寄存器都被扩展为 64 位，增加了 8 个新的通用寄存器和 8 个新的 SSE 寄存器。表 2.4 列出了新增的这些寄存器。通用寄存器的小的尺寸为大的值的低部分。你仍然可以在长模式下访问 ah，bh，ch 和 dh 寄存器，但你不能在新的指令中使用任何新的寄存器。

通常 x86 体系中任何指令，允许 16 位或 32 位操作数尺寸，在长模式下还允许 64 位操作数。64 位操作数必须在长模式下寻址，也允许 32 位寻址，但不能使用基于 16 位寄存器的地址。下面为长模式中的 mov 指令例子：

```
mov rax,r8          ; 传送 64 位通用寄存器
mov al,[rbx]        ; 传送通过 64 位寄存器寻址的内存
```

类型	通用寄存器				SSE
位	8	16	32	64	128
				rax	
				rcx	
				rdx	
				rbx	
	spl			rsp	
	bpl			rbp	
	sil			rsi	
	dil			rdi	
	r8b	r8w	r8d	r8	xmm8
	r9b	r9w	r9d	r9	xmm9
	r10b	r10w	r10d	r10	xmm10
	r11b	r11w	r11d	r11	xmm11
	r12b	r12w	r12d	r12	xmm12
	r13b	r13w	r13d	r13	xmm13
	r14b	r14w	r14d	r14	xmm14
	r15b	r15w	r15d	r15	xmm15

表 2.4：长模式新寄存器

长模式也使用基于地址的指令指针，你可以手动用 RIP 指定，但这些地址也能自动由 FASM 生成，因此在长模式下没有 64 位绝对地址。你可以通过中括号中 dword 尺寸重写地址来强制汇编器使用 32 位绝对地址。也有一个使用 64 位绝对寻址的例外，它为 mov 后跟着其中一个为累加器，第二个为内存操作数的情况。使用 qword 来强制汇编器使用 64 位绝对寻址。当没有指定尺寸操作符时，汇编器自动生成最佳格式。

```
mov [qword 0],rax    ; 绝对 64 位寻址
mov [dword 0],r15d   ; 绝对 32 位寻址
mov [0],rsi          ; 自动 RIP 相对寻址
mov [rip+3],sil      ; 手动 RIP 相对寻址
```

作为 64 位操作立即数只可能为 32 位数，唯一例外是带有 64 位通用寄存器目的操作数的 mov 指令。试图其他指令使用 64 位立即数将导致错误。

如果在长模式下操作 32 位通用寄存器的指令，64 位寄存器的高 32 位填充为 0.这不同于 16 位或 32 位那些指令操作，它们保留高位。

新增三条类型转换指令。cdqe 符号扩展 EAX 中 dword 到 qword 并保持结果到 RAX。cqo 符号扩展 RAX qword 为双 qword 并保存额外位到 RDX 寄存器。这些指令没有操作数。movsxd 符号扩展 dword 源操作数到 64 位目的操作数，源操作数可以为 32 为寄存器或内存，目的操作数必须为寄存器。没有零扩展类似指令，因为它自动由 32 位寄存器完成，上一段中说明的那样。movzx 和 movsx 指令遵守通常规则，可以使用 64 位目的操作数，允许扩展字节或字到 qword。

所有二进制算术和逻辑指令提升以允许在长模式下操作 64 位操作数。在长模式下禁止使用十进制算术指令。

堆栈操作，比如 push 和 pop 在长模式下默认为 64 位操作数，不能使用 32 位操作数。pusha 和 popa 在长模式下不可用。

间接 near 调整和调用在长模式下默认为 64 位操作数，它不能使用 32 位操作数。另外，间接 far 调整和调用允许任何 x86 体系允许的操作数，也允许使用 80 位内存操作数（仅 EM64T 实现了），80 位内存操作数有第一个定义偏移的 8 字节和指定选择子的最后两个字节组成。长模式下不允许直接 far 调整和调用。

I/O 指令，in，out，ins 和 outs 为例外指令，它们不允许在长模式下操作 qword 操作数。但其他串操作可以。他们有新的段格式 movsq，cmpsq，scasq，lodsq 和 stosq。RSI 和 RDI 寄存器默认用来寻址这些串元素。

lfs，lgs 和 lss 用来扩展以接受 80 位元内存操作数和 64 位目的寄存器（仅 EM64T 实现了）。lds 和 les 不能在长模式下使用。

系统指令，比如需要 48 位内存操作数的 lgdt，在长模式下需要 80 为内存操作数。

cmpxchg16b 为 64 位的 cmpxchg8b 等价指令，它使用双 qword 内存操作数和 64 为寄存器来执行类似操作。

swapgs 为新增指令，它置换 GS 寄存器和 KernelGSbase MSR 寄存器的内容（MSR 地址为 0C0000102h）。

syscall 和 sysret 为新增指令，用来在长模式下提供和 sysenter 和 sysexit 相似功能的指令，而 sysenter 和 sysexit 在长模式下不允许使用。

2.2 控制伪指令

这章将讲述控制汇编流程的伪指令，它们在汇编时处理，并可能会引起某些指令块的不同汇编或完全不被汇编。

2.2.1 数值常量

=伪指令可以用来定义数值常量。它前面为常量名，后面为提供值的数值表达式。常量值可以为数字或地址，但不同于标号，数值常量不允许拥有基于寄存器的地址。除了这点不同外，数值常量和标号非常类似，它们甚至可以前置引用（在它们的定义前使用它们）。

当你用已经定义的数值常量定义数值常量时，汇编器把常量当作汇编期间的变量，并且允许赋给新值，但不允许前置引用（很显然的原因）。让我们看一个例子：

```
dd sum
x = 1
x = x+2
sum = x
```

这里 x 为汇编期间的变量，每次访问它的值为最后一次使用的值。因此如果在 x 定义前访问它，比如将“dd sum”指令改为“dd x”，将导致错误。当通过指令“x = x + 2” 重新定义 x 时，将用先前 x 的值来计算新的 x 值。所以当常量 sum 定义时，x 值为 3 并将赋给 sum。因为 sum 在源码中只定义了一次，它是标准的数值常量，能够前置引用。所以“dd sum”将被汇编为“dd 3”。可阅读 2.2.6 节来了解汇编器是如何做到解析的。

数值常量值前可以放尺寸操作符，用来确保值在某个给定范围内，并能影响数值表达式中的这些计算是怎样执行的。例如：

```
c8 = byte -1
c32 = dword -1
```

上面的语句将定义两个不同的常量，第一个存放 8 位，第二个存放于 32 位。

当你用地址值（可以基于寄存器寻址）定义常量时，可以使用 label 伪指令的扩展语法（已在 1.2.3 节中描述）。例如：

```
label myaddr at ebp+4
```

上面的语句声明一个 ebp+4 地址处的标号。然而标号不同于数值常量，不能成为汇编期间的变量。

2.2.2 条件汇编

if 伪指令导致跟着的指令块仅在特定条件下汇编。它必须跟着指定条件的逻辑表达式，当条件满足时汇编后一行的指令，否则跳过。可选的 else if 伪指令后面跟着指定条件的逻辑表达式，当前面的条件不满且这个条件满足时汇编后面的指令块。可选的 else 伪指令开始的指令块，当所有条件都不满足时才汇编。end if 伪指令结束最后的指令块。

应当注意的是 if 伪指令是在汇编阶段处理的，因此它不会影响任何预处理伪指令，比如符号常量和宏指令的定义 - 当汇编器识别出 if 伪指令时，所有的预处理已经完成。

逻辑表达式由逻辑值和逻辑操作符组成。逻辑操作符有“~”逻辑非，“&”逻辑与，“|”逻辑或。逻辑非操作优先级最高。逻辑值可以为数值表达式，如果等于 0 为 false，否则为 true。两个数值表达式可以使用以下操作符做比较来生成逻辑值：=（等于），<（小于），>（大于），<=（小于或等于），>=（大于或等于），<>（不相等）。

跟着符号名的 used 操作符为逻辑值，用来检查给定符号是否在某些地方使用过（即使在检查位置后使用它也能返回正确的结果）。defined 操作符后面可以跟着任何表达式，通常为一符号名；它检查只包含唯一符号的给定表达式是否在源码中已定义并能在当前位置访问。

下面例子中使用的 count 常量必须在源码中某处定义：

```
if count>0
  mov cx,count
  rep movsb
end if
```

当 count 大于 0 时将汇编其中的两条指令。下面为更加复杂的条件结构：

```
if count & ~ count mod 4
  mov cx,count/4
  rep movsd
else if count>4
  mov cx,count/4
  rep movsd
  mov cx,count mod 4
  rep movsb
else
  mov cx,count
  rep movsb
end if
```

当 count 不为零且能被 4 整除时将汇编第一个指令块，如果不满足，将评估跟在 else if 后面的逻辑表达式，如果为 true 将汇编第二个指令块，否则汇编 else 中的指令块。

还有用来比较符号串操作符。eq 比较两个值是否精确相等。in 操作符检查给定值是否在其后面给的列表中，列表必须放在中括号中，列表项以逗号分隔。对于汇编器来说，当符号们有相同含义时被认为相同 - 例如对于汇编器 pword 和 fword 是相同的。“16 eq 10h”条件为 true，而“16 eq 10+4”条件为 false。

eqtype 操作符检查比较的两个值结构是否相同，以及结构的元素是否为相同类型。

常见的类型有数值表达式，单独的字符串，浮点数，地址表达式（在中括号内或者前面有 ptr 操作符），指令助记符，寄存器，size 操作符，跳转类型和代码类型操作符。并且每一个用作分隔符的特殊字符，如逗号或冒号，用来分隔类型。例如，两个由寄存器名和逗号和数值表达式构成的值，将被认为是相同类型，无论使用了什么寄存器和复杂的数值表达式；除非字符串和浮点数，它们为特殊类型的数值表达式，被认为为不同的类型。所以“eax,16 eqtype fs,3+7”条件为 true，但“eax,16 eqtype eax,1.6”为 false。

2.2.3 重复块指令

times 伪指令重复一条指令到指定次数。它后面必须跟着重复次数数值表达式和重复指令（可选的冒号用来分隔数字和指令）。当在指令中使用特殊符号“%”时，其值等于当前的重复次数。例如“times 5 db %”将定义五字节的数据 1，2，3，4，5。也允许递归使用 times 伪指令，所以“times 3 times % db %”将定义 6 字节数据 1，1，2，1，2，3。

repeat 伪指令重复整个指令块。它后面必须跟着重复次数数值表达式。重复的指令出现在另一行，最后以 end repeat 伪指令结束，例如：

```
repeat 8
  mov byte [bx],%
  inc bx
end repeat
```

上面的语句生成的代码将把从 1 到 8 存储到 BX 寻址的内存单元内。

重复次数可以为 0，此时不会汇编任何指令。

break 伪指令停止前面的重复，继续汇编 end repeat 后面的第一行。和 if 伪指令一起使用，可以在某些特殊条件下停止重复，如：

```
s = x/2
repeat 100
  if x/s=s
    break
  end if
  s = (s+x/s)/2
end repeat
```

当 while 伪指令后面的逻辑表达式为真时重复指令块。重复指令块必须以 end while 伪指令结束。在每一次重复前逻辑表达式都将评估，当值为 false 时，将汇编 end while 后第一行。此时“%”也用来保存当前重复值。break 伪指令也可以像 repeat 指令那样结束循环。前面的例子可以用 while 指令重写为：

```
s = x/2
while x/s <> s
  s = (s+x/s)/2
  if % = 100
    break
  end if
end while
```

if，repeat 和 while 定义的块可以任意嵌套，但它们必须以它们开始的顺序依次结束。Break 伪指令常用来停止处理 repeat 或 while 开始的块。

2.2.4 地址空间

org 伪指令设置后面代码将被期望出现在内存中的地址。它必须跟着指定地址的数值表达式。这个伪指令开始新的地址空间，后面的代码并不会做任何移动，但将影响其中定义的所有标号和$符号，如同被放到给定地址一样。然而程序员必须负责将代码放到执行时正确的位置。

load 伪指令用来从已汇编代码的二进制值中定义常量。这条伪指令必须跟着常量名，可选的尺寸操作符，from 操作符和指定了当前地址空间中有效地址的数值表达式。尺寸操作符此时有特殊含义 - 它声明了多少字节（最多为 8）将被加载组成常量的二进制值。如果没有指定尺寸操作符，将载入一个字节（因此值的范围为 0 到 255）。载入的数据不能超过当前的偏移。

store 伪指令通过替换前面生成的代码为给定数值表达式来修改已生成的代码。这个表达式可以放置可选的尺寸操作符来指定要定义多大的值。如果没有定义尺寸操作符默认为一个字节。然后 at 操作符和数值表达式定义了当前地址代码空间中的有效地址。这是个高级伪指令，应当小心使用。

load 和 store 伪指令操作都限制在当前地址空间中。$$总是等于当前地址空间的基地址，$为当前地址空间的当前地址，因此这两个值定义了 load 和 store 能够操作的区域界限。

load 和 store 一起使用可以为已生成的代码编码。例如在当前地址空间中编码生成的所有代码可以使用这样的伪指令块：

```
repeat $-$$
  load a byte from $$+%-1
  store byte a xor c at $$+%-1
end repeat
```

其中代码中每个字节将和常量 c 异或。

virtual 在指定位置定义虚拟数据。这个数据不会包含到输出文件中，但定义的标号可以在源码中使用。这条伪指令可以跟着 at 操作符和指定虚拟数据地址的数值表达式（如果没有指定地址将使用当前地址，和“virtual at $”相同）。数据定义指令在另一行，最后以 end virtual 伪指令结束。virtual 指令本身是一个独立的地址空间，当它结束后，将恢复前面的地址空间上下文。

virtual 伪指令可以用来创建一些变量的联合，例如：

```
GDTR dp ?
virtual at GDTR
  GDT_limit dw ?
  GDT_address dd ?
 end virtual
```

它在 GDTR 地址处为 48 位的变量定义了 2 个标号。

它也可以为寄存器寻址的结构定义标号：

```
virtual at bx
  LDT_limit dw ?
  LDT_address dd ?
end virtual
```

有了上面的定义后，指令“mov ax, [LDT_limit]”将被汇编为“mov ax,[bx]”。

在虚拟块中声明已定义的数据值或指令也是有用的，因为 load 伪指令可以从虚拟的生成代码中载入数据到常量。这条伪指令必须在载入的代码后，虚拟块结束前使用，因为它只能从当前地址空间载入数据。例如：

```
virtual at 0
  xor eax,eax
  and edx,eax
  load zeroq dword from 0
end virtual
```

上面的代码将定义 zeroq 常量，包含 4 字节定义在虚拟块中的机器码指令。也可以用这个方法从外部文件载入二进制数据。例如：

```
virtual at 0
  file ’a.txt’:10h,1
  load char from 0
end virtual
```

上面的语句从文件 a.txt 10h 文件偏移处载入一字节到 char 常量。

2.4 中描述的任何 section 伪指令都将开始一新地址空间。

2.2.5 其他伪指令

align 伪指令对齐代码或数据到指定边界。它必须跟着对齐大小的数值表达式。边界值必须为 2 的指数。

为了实现对齐，align 伪指令将把跳过的字节用 nop 指令填充，同时标识这段区域为已初始化数据，所以它和其他未初始化数据一起将不会出现在输出文件中，对齐字节将执行相同方式。如果你需要填充对齐区域为其他值，可以用 align 和需要的对齐值 virtual 自己创建对齐，例如：

```
virtual
  align 16
  a = $ - $$
end virtual
db a dup 0
```

常量 a 在对齐后地址和 virtual 块地址之间定义不同（见上一部分），所以它等于需要对齐空间的大小。

display 伪指令在汇编期间显示信息。它必须跟着字符串或者字节数据，用逗号分隔。它也可以用来显示常量，例如：

```
bits = 16
display 'Current offset is 0x'
repeat bits/4
  d = '0' + $ shr (bits-%*4) and 0Fh
  if d > '9'
    d = d + 'A'-'9'-1
  end if
  display d
end repeat
display 13,10
```

这个伪指令计算 4 个 16 位值的 16 进制数字，并且以字符显示。注意如果当前地址空间为可重定位的是不可用（比如可能出现在 PE 或 COFF 输出格式中），此时只能使用绝对的值。绝对的值可以通过相对地址的计算得到，如同$-$$，或 rva $（Pe 格式中）。

2.2.6 多遍扫描

因为汇编器允许在标号或者常量实际定义前引用它们，所以它必须预测这些标号的值。只有有一种情形下怀疑预测失败，它将再次执行扫描，汇编所有源码，基于上遍扫描中得到的标号值来做更好的预测。

标号值的变化将引起一些指令有不同长度的编码，并能再次引起标号值的变化。既然在表达式中可以使用影响控制伪指令的标号和常量，整个源码在新的一遍扫描中可能完全不同的处理，每一次都试图做更好的预测，直到所有的值都预测正确。它使用各种方法来预测值，对于大多数程序通过几遍扫描就能找到的最短路径。

一些错误，如值不能符合要求的边界，不能在中间扫描中提示，它可能在某些值预测好后会才出现。如果汇编器遇到一些非法语法或未知指令，它将立即停止。此外多次定义一个标号也将导致错误，因为它会引起无效的预测。

只有 display 伪指令创建的消息能在最后一次扫描中显示。当汇编器因为错误停止时，这些消息将返回还没有解析正确的预测值。

扫描遍数有个上限，当汇编器达到上限后，它将停止并提示无法生成正确代码的消息。看看下面的例子：

```
if ~ defined alpha
  alpha:
end if
```

当 defined 操作符后的表达式在此处能计算时将传给 true 值，此处意思为 alpha 标号在某处已定义。但上面的块仅当 defined 操作符给定的值为 false 时才会定义，将导致一个 xx 和是其不能解析那些代码。当处理 if 伪指令，汇编器必须预测标号 alpha 是否在某处定义（如果在这遍扫描前已定义就不需要预测了），无论预测出什么，总会发生相反的。一次汇编器将会失败，除非标号 alpha 在上面指令块之前某地定义 - 此时，正如已经提到的，不需要预测并且块将被跳过。

上面的例子也可以写成当它没有定义时定义标号。如果失败，因为 defined 操作符将检查标号是否已定义，它包括在条件处理块中的定义。然后增加些条件就能让它解析:

```
if ~ defined alpha | defined @f
  alpha:
  @@:
end if
```

@f 总和源码后面最近的@@符号有相同的标号，所以上面的例子中，如果使用了任何唯一名称而不是匿名标号，意思是相同的。当 alpha 没有在源码其他地方定义时，唯一的可能是当这个块定义并且此时他不会导致 XX，因为你们标号标号将会是这个块自行创建。为了更好的理解这点，看看下面这个除了自我创建外没有任何东西的块：

```
if defined @f
  @@:
end if
```

这是一个可以有多个方案的源码，当这个块被处理或没有所有这些情形都正确。取决于汇编器算法 - 预测算法。回到前面的例子，当 alpha 没有在任何地方定义，if 条件块不为 false，所以你只有一种可能方案，我们希望汇编器能够到达。灵位，当 alpha 在某些地方定义时，我们将再次得到两种可能方案，但其中将导致 alpha 定义两次，那样的错误将导致汇编器立即终止汇编，这是一个极深的扰乱解析过程的错误种类，我们得到将取决于汇编器内部的选择。

然而这些选择中存在某些确定的事实。当汇编器需要检查给定符号是否定义并且它已经在当前扫描中定义时，不需要任何预测 - 这已经在前面说明。当给定符号从未在以前定义，包括所有已经完成的扫描，汇编器将预测它没有定义。知道这点，我们可以期待如同上面的简单的自我创建块不被汇编，上个例子中在外面的条件块前当 alpha 在某地定义时将被正确解析，当它在前面没有定义时，它家那个定义 alpha，因此潜在的导致错误，因为如果 alpha 在后面某处定义将有两个定义。

used 操作符类似，然而任何预测都不是简单的，你不应依赖它们。

2.3 预处理伪指令

所有预处理指令在主汇编过程前处理，因此不受控制伪指令的影响。此时所有的注释都已被去除。

2.3.1 包含源文件

include 伪指令在其使用位置包含指定源文件。它后面必须跟着要包含的文件名，例如：

```
include 'macros.inc'
```

整个包含文件会在下一行被预处理前预处理。要包含的文件无个数限制。

文件路径可以包含“%”括起的环境变量，它们将被替换为实际的值，“\”和“/”都允许作为地址分隔符。如果没有指定文件的绝对地址，将首先在包含它的那个文件路径下搜索该文件，没有找到的话，将在主源码文件目录搜索（命令行中指定的）。这些规则也适用与 file 伪指令。

2.3.2 符号常量

符号常量不同于数值常量，在汇编过程前所有符号常量都被它们的值替代，它们的值可以为任何东西。

符号常量定义由常量名后面跟着 equ 伪指令组成。这条伪指令后面跟着的任何东西都将成为常量的值。如果符号常量的值包含其他符号常量，他们将在赋值给新的常量值前会先替换它们的值。例如：

```
d equ dword
NULL equ d 0
d equ edx
```

在这三个定义后，NULL 常量的值为 dword 0，d 的值为 edx。所以“push NULL”将被汇编为“push dword 0”，“push d”将被汇编为“push edx”。然后下面的行：

```
d equ d,eax
```

常量 d 将得到新的值“edx,eax”。这种方式可以用来定义增长的符号列表。

restore 伪指令用来恢复上次定义的常量的值。它后面应当跟着一个或多个用逗号分隔的符号常量。所以在上面定义后“restore d”将给常量 d 恢复到值 edx，再来一次“restore

d”将恢复至 dword，再次恢复 d 将变成初始时的含义（常量 d 没有定义）。如果没有定义给定名称的常量，restore 不会导致错误，它将忽略掉。

符号常量可以用来调节汇编器语法。例如下面的定义为尺寸操作符提供了方便的快捷方式：

```
b equ byte
w equ word
d equ dword
p equ pword
f equ fword
q equ qword
t equ tword
x equ dqword
```

因为符号常量允许为空值，可以在任何地址值前使用 offset 语法：

```
offset equ
```

这样定义后“mov ax,offset char”将拷贝变量 char 的偏移到寄存器 ax，因为 offset 被替换为空值，因此将被忽略掉。

符号常量也可以用 fix 伪指令来定义，它和 equ 有相同的语法，但定义常量的优先级更高 - 它们甚至在处理预处理伪指令和宏指令前被替换，唯一例外的是 fix 伪指令本身，它有最高可能的优先级，这点可以用来重新定义常量。

fix 伪指令可以用来调整预处理器伪指令的语法，这通常不能用 equ 伪指令实现。例如：

```
incl fix include
```

上面的语句将为 include 伪指令定义一个短名称，而同样的 equ 伪指令定义不能得到那样的结果，因为标准的符号常量是在搜索预处理伪指令之后才进行替换的。

2.3.3 宏指令

macro 伪指令允许定义自己的复杂指令，称为宏指令。宏指令极大的简化编程过程。最简单的形式类似符号常量的定义。例如，下面为指令“test al,0xFF”定义快捷方式。

```
macro tst {test al,0xFF}
```

macro 伪指令之后为宏指令名和以"{" 和"}"括起来的内容。在这个定义后你可以在任何地方使用 tst 指令，它将被汇编为“test al,0xFF”。定义符号常量 tst 将有相同的效果，但不同的是宏指令名仅作为指令助记符识别。此外，宏指令在符号常量替换之前被替换相应代码。所以如果你定义了同名的宏指令和符号常量，并且当作指令助记符使用，其内容将被替换为宏指令的，但如果在操作数使用将被替换为符号常量的值。

宏指令的定义可以有多行组成，因为字符“{”和“}”不必和 macro 宏指令在同一行，例如：

```
macro stos0
{
  xor al,al
  stosb
}
```

任何使用宏指令 stos0 的地方将被两条汇编指令替换。

如同指令需要一些操作数一样，宏指令也可定义为接受一些一些用逗号分隔的参数。需要的参数名称必须跟在同行 macro 宏指令后面。宏指令中出现的任何参数名都将被替换为宏指令使用时的对应值。下面是一个用于二进制输出格式数据对齐的宏指令：

```
macro align value { rb (value-1)-($+value-1) mod value }
```

这条宏指令定义后，当发现“align 4”指令后，value 将为 4，所以结果为“rb (4-1)-($+4-1) mod 4”。

如果宏指令定义中出现了同名的指令，那么将使用前面定义的含义。这可以用来重定义宏指令，例如：

```
macro mov op1,op2
{
  if op1 in  & op2 in
    push op2
    pop op1
  else
    mov op1,op2
  end if
}
```

这条宏指令扩展了 mov 指令的语法，允许两个操作数都为段寄存器。例如“mov ds,es”将被汇编为“push es”和“pop ds”。其他情形下将使用标准的 mov 指令。这个 mov 的语法还能进一步的扩展，比如：

```
macro mov op1,op2,op3
{
 if op3 eq
   mov op1,op2
 else
   mov op1,op2
   mov op2,op3
 end if
}
```

它允许 mov 指令接受三个操作数，但仍然允许两个操作数，因为当宏指令传入比实际需要少的参数时，剩余的参数将为空值。当三个参数都给定时，这条宏指令将变成两个前面定义的宏指令，所以“mov es,ds,dx”将被汇编为“push ds, pop es”和“mov ds,dx”。

参数名后带有字符“*”表明这个参数是必须的-预处理器不允许这个参数为空值。例如上面的宏指令可以这样声明：“macro mov op1*,op2*,op3”来确保头两个参数必须给定非空值。

当传给宏指令包含逗号的参数时，参数必须用尖括号括起来。如果包含不止一个字符“<”，应当使用同样数目的字符“>”来结束参数值。

purge 伪指令消除上次定义的宏指令。它必须跟着一个或多个逗号分隔的宏指令名。如果这个宏指令没有定义，将不会产生任何错误。例如上面定义的扩展 mov 宏指令后，你可以通过“purge mov”宏指令来删除三个操作符的宏指令。另外“purge mov”将删除 2 个操作符的宏指令，以及所有这些伪指令将没有任何效果。

如果 macro 伪指令后存在一些方括号括起的参数名，那么当使用这条宏指令时它允许为这个参数给定多个值。这个宏指令将会依次展开给定参数。这也是为什么中括号后不年能再有任何参数的原因。宏指令的内容将被参数组分别处理。一个简单的包含一个在中括号中的参数例子：

```
macro stoschar [char]
{
 mov al,char
 stosb
}
```

这条宏指令接受无限数目的参数，并且每一个将分别处理成两天指令。例如 stoschar 1,2,3 将被汇编出以下指令：

```
mov al,1
stosb
mov al,2
stosb
mov al,3
stosb
```

有一些只能在宏指令定义中使用特殊的宏指令。local 宏指令定义局部名称，每次宏指令使用时将被替换为唯一值。它必须跟着用逗号分隔的名称。如果参数名以“.”或“..”开头，由每一个计算的宏指令生成的唯一标号将有相同的属性。这通常用来定义在宏指令内部使用的常量和标号。例如：

```
macro movstr
{
  local move
move:
  lodsb
  stosb
  test al,al
  jnz move
}
```

每次宏指令使用，mov 在其指令中将变成唯一的名称，因此不会产生标号重复定义错误。

forward，reverse 和 common 伪指令把宏指令分成块，每块都在前面的块完成后处理。它们的不同点仅在当宏指令接受多个参数组时。forward 伪指令后的指令块将依次处理每个参数组，从第一个到最后一个 - 类似默认的块（前面没有任何这些伪指令）。跟在 reverse 伪指令的块将以相反的顺序处理 - 从最后一个到第一个。跟在 common 伪指令的块只处理一次，通常是对于所有的参数组。在处理同组参数时，在某块中定义的局部名字对其他块而言是可见的。当然 common 块中定义的局部标号对所有块都是可见的，与处理的是哪个参数组无关。

下面是一个在字符串之前创建字符地址数组的例子：：

```
macro strtbl name,[string]
{
  common
  label name dword
  forward
  local label
  dd label
  forward
  label db string,0
```

```
}
```

这个宏指令的第一个参数将成为地址表的标号，后面的参数应当为字符串。第一个块仅处理一次，它定义了标号。第二个块为每个字符串声明局部名，并且定义一个表项来存放字符串地址。第三个块用相应的标号名为每个字符串定义数据。

开始块的伪指令可以和其后的代码放在同一行，比如：

```
macro stdcall proc,[arg]
{
  reverse push arg
  common call proc
}
```

这条宏指令用 STDCALL 约定来调用过程，堆栈以相反的方向压栈。例如“stdcall foo,1,2,3”将被汇编为：

```
push 3
push 2
push 1
call foo
```

如果宏指令中一些名称有不同的值（既可以为方括号中的参数，也可以为跟在 forward 或 reverse 伪指令块中定义的局部名称）并且用在 common 伪指令块中，它将被替换为以逗号分隔的所有的值。例如下面的例子把所有参数传递给前面定义的 stdcall 宏：

```
macro invoke proc,[arg]
{ common stdcall [proc],arg }
```

它可以用来 STDCALL 方式间接调用（通过内存指针）过程。

在宏指令内部也可以使用特殊的#操作符。这个操作符将两个名称连接在一起。由于连接是在所有的参数和局部变量都用真实值代替之后，所以有时可能会很有用。下面的宏指令将根据 cond 参数生成条件跳转：

```
macro jif op1,cond,op2,label
{
  cmp op1,op2
  j#cond label
}
```

例如“jif ax,ae,10h,exit”将被汇编为“cmp ax,10h”和“jae exit”指令。

#操作符也能合并两个字符串。宏指令中也可以用'操作符将名称转换为字符串。它转换后面的名称为字符串，但注意，当它后面跟着多个符号宏参数时，只有第一个被转换，也就是说'操作符值转换后面紧跟着的那个符号。下面是使用这两个特性的一个例子：

```
macro label name
{
  label name
  if ~ used name
    display ‘name # " is defined but not used.",13,10
  end if
}
```

以这种方式定义的宏在源文件中未使用的时候，宏就会发出警告信息，通知哪个标号未使用。

为使宏能够根据参数类型的不同其行为有所不同，可使用"eqtype"比较操作符。下面是一个区分字符串和其他参数的宏指令：

```
macro message arg
{
  if arg eqtype ""
    local str
    jmp @f
    str db arg,0Dh,0Ah,24h
    @@:
    mov dx,str
  else
    mov dx,arg
  end if
  mov ah,9
  int 21h
}
```

上面的宏用于 DOS 程序中显示信息。当这个宏的参数为一些数字，标号，或变量，将会显示该地址的字符串，但当参数为字符串时，创建的代码将显示后面包含回车和换行字符的字符串。

宏指令中也可以在另一个宏指令声明，这样可以使用一个宏定义另外一个宏，但这样做存在一个问题宏指令中不能出现字符“}”，因为它表示宏定义的结束。为了解决这个问题，可以在宏指令内部使用转义字符。通过在任何符号（甚至特殊字符）前放置一个或多个字符“\”。预处理将它们作为一个单一的符号处理，每次在处理宏指令时遇到这些符号，都将取出前面的字符“\”。例如“\}”被当作一单一符号，当处理宏指令时变成了符号“}”。这允许在一个宏指令定义中嵌套另一个：

```
macro ext instr
{
  macro instr op1,op2,op3
  \{
    if op3 eq
      instr op1,op2
    else
      instr op1,op2
      instr op2,op3
    end if
  \}
}
ext add
ext sub
```

ext 宏定义正确，当使用它时，“\{”和“\}”变成了符号“{”和“}”。所以当处理 ext add 时，宏的内容变成有效的宏指令定义，add 宏将被定义。同样 ext sub 定义了 sub 宏。这里使用符号“\{”并不是必须的，但这样做可以让定义更清晰些。

如果某些只用于宏指令的伪指令，比如 local 或 common 也需要在以这种方式嵌套在宏中，它们也可以以同种方式转义。用多于一个的“\”转义符号也是允许的，这允许用来定义多层嵌套的宏。

另一种在一个宏指令中定义另一个的技术是 fix 伪指令，当某些宏指令仅以另一个的定义开始时，而没有结束它时将变得很有用处。例如：

```
macro tmacro [params]
{
common macro params {
}
MACRO fix tmacro
ENDM fix }
```

定义了另一种定义宏指令的语法，如同：

```
MACRO stoschar char
  mov al,char
  stosb
ENDM
```

注意，这样个性化的定义必须使用"fix"来定义，因为只有更高优先级的常量才能在预处理器在定义宏时查找"}"字符之前处理。这对于那些想在定义结束之后执行一些额外操作的时候可能是一个问题，但还有一个特性可能有助于这个问题的处理。也就是说可以将任何伪

指令、指令或宏指令放在结束宏的"}"字符之后，这就会使得他像被放到了下一行一样进行处理。

2.3.4 结构

struc 伪指令是用于定义数据的"macro"的变例。使用"struc" 定义的宏在使用时前面必需加一个标号（和数据定义伪指令一样），这个标号会附加在所有以"."开头的元素名称前面。以"struc"定义的宏和使用"macro"定义的宏名字可以相同，不会相互影响。所有适用于"macro"的规则也同样适用于"struc"。以下是一个结构宏指令的例子

下面为一个结构宏指令例子：

```
struc point x,y
{
  .x dw x
  .y dw y
}
```

例如 my point 7,11 将定义结构标号 my，包含两个值为 7 的变量 my.x 和值为 11 的变量 my.y。

如果在结构定义中出现字符“.”，它将被替换为结构的标号名，而且此时不会自动定义该标号，以允许完全的手动定义。下面为利用这个特性来扩展数据定义伪指令 db 以能够计算定义数据的大小例子：

```
struc db [data]
{
  common
  . db data
  .size = $ - .
}
```

这样定义后，“msg db 'Hello!',13,10”还将定义“msg.size”常量，其值等于定义数据占用的字节大小。

定义寄存器或绝对值寻址的数据结构应当使用和结构宏指令一起使用的 virtual 伪指令（见 2.2.5）。

restruc 伪指令消除上次的结构定义，如同 purge 处理宏指令，restore 处理符号常量一样。它也有相同的语法 - 跟着一个或多个逗号分隔的结构宏指令。

2.3.5 重复宏指令

rept 伪指令是特殊种类的宏指令，用来重复用括号括起来的块到指定数目。rept 伪指令的基本语法是跟着数字（不能为表达式，因为预处理器不做计算，如果需要重复的值基于汇编器计算，使用汇编器处理的代码重复指令，见 2.2.3），然后是字符“{”和“}”之间的源码块。一个简单的例子：

```
rept 5 {in al, dx}
```

将变成 5 份重复的 in al,dx 行。和可以在宏指令内部使用的标准宏指令和任何特殊操作符和伪指令同样方式定义的指令块也允许使用。当重复次数为 0 时，块将被简单的跳过，如果你定义宏指令但从不使用它。重复次数可以跟着重复次数符号名称，它将被替换为当前生成的重复次数。如同：

```
rept 3 counter
{
  byte#counter db counter
}
```

将生成：

```
byte1 db 1
byte2 db 2
byte3 db 3
```

应用于 rept 块的重复机制和宏指令中处理多个参数组的相同，所以像 forward，common 和 reverse 伪指令都可以使用。因此伪指令：

```
rept 7 num { reverse display ‘num }
```

将显示数字 7 到 1 为文本。local 伪指令的行为和带有多个参数组的伪指令中的相同，所以：

```
rept 21
{
  local label
  label: loop label
}
```

将为每次重复生成唯一的标号。

重复次数符号默认次数为 1，你可以在计数器名称后面跟着冒号和起始次数来改变默认的起始次数。例如：

```
rept 8 n:0 { pxor xmm#n,xmm#n }
```

生成的代码将 清除 8 个 SSE 寄存器的值。你可以用逗号来定义多个计数器，每一个都可以有不同的起始计数。

irp 伪指令通过给定的列表参数来逐一重复。irp 后面跟着参数名，然后是逗号和一串参数。参数的指定类似宏指令中的方式，所以他们可以以逗号分隔并且每一个都可以包含在“<”和“>”之间。另外 参数名称也可以跟着字符“*”来说明某个参数不能传空值。这样的块：

```
irp value, 2,3,5
{ db value }
```

将生成：

```
db 2
db 3
db 5
```

irps 伪指令从给定列表符号中重复，它必须跟着参数名，然后逗号和任何符号序列。序列中的每一个符号，无论是否为名称符号，符号字符或字符串，都将成为一次重复的参数。如果逗号后没有任何符号，将不会执行任何重复。例如：

```
irps reg, al bx ecx
{ xor reg,reg }
```

将生成：

```
xor al,al
xor bx,bx
xor ecx,ecx
```

irp 和 irps 定义的块将和任何宏指令以相同方式处理，所以只用于宏指令的操作符和伪指令此时也可以自由使用。

2.3.6 条件宏指令

match 伪指令将引起一些块被预处理，而且仅当给定符号序列匹配指定模式时传给汇编器。模式在前，以逗号结束，然后是执行匹配操作的符号，最后是在“{”和“}”之间的块源码。

有几条规则用来创建匹配表达式，第一是任何符号字符和字符串应当精确匹配。例如：

```
match +,+ { include ’first.inc’ }
match +,- { include ’second.inc’ }
```

第一个文件将被包含，因为逗号后的“+”匹配模式“+”，第二个文件将不会包含，因为不匹配。

为了照字面意思匹配，模板前面必须放置字符“=”，为了匹配字符“=”，或“,”，必须使用“==”和“=,”。例如模板“=a==”就匹配“a=”序列。

如果在模板中放置一些名称符号，它匹配其中至少一个符号的任何序列，并且这个名称将被替换为匹配序列，类似宏指令的参数。例如：

```
match a-b, 0-7
{ dw a,b-a }
```

将生成 dw 0,7-0 指令。每一个名称都将匹配为尽可能小的符号，留下剩余做为后面的一个，所以在这个例子中：

```
match a b, 1+2+3 {db a}
```

名称 a 匹配符号 1，剩下+2+3 序列来匹配 b。而：

```
match a b, 1 {db a}
```

将不会为 b 剩下任何东西匹配，所以这个块将不会处理任何事情。

match 定义的源码块将和宏指令以相同方式处理，所以宏指令的任何操作符也都能在这里使用。

使得 match 伪指令更加有用的事实是执行匹配前在匹配序列符号中（逗号后源码块开始前的任意地方）它替换符号常量为它们的值。这样可以用来在一些符号常量给定了值的某些条件下处理源码块：

```
match =TRUE, DEBUG { include ’debug.inc’ }
```

当定义了符号常量 DEBUG 为 TRUE 时将包含文件。

2.3.7 处理顺序

当组合各种不同的预处理器特性时，知道它们处理的先后顺序是很重要的。正如上面已经提到的，最高优先级为 fix 伪指令和用它替换定义的。在做任何预处理前完成的，一次下面的源码：

```
V fix {
macro empty
V
V fix }
V
```

成为一个有效的空宏指令定义。它可以被解析为：fix 伪指令和有优先级的的符号常量在单独阶段处理，所有其他的预处理将在此后的源码中完成。

标准的预处理在后面，在每个识别为别的第一个符号开头。它从检查预处理伪指令开始，当没有发现时，预处理将检查是否第一个符号为伪指令。如果没有发现伪指令，它移动到第二行的符号，再一次的开始检查伪指令，此时只有 equ 伪指令，如同在第二行符号只有一个。如果没有伪指令，第二个符号将被检查结构宏指令，当所有这些都没有检查出正值时，符号常量将被替换为他们的值，并且这些行将传给汇编器。

为了在例子中验证这点，假设有存在一个已定义的宏指令 foo 和结构宏指令 bar。下面的行：

```
foo equ
foo bar
```

将都被解析为宏指令 foo 调用，因为第一个符号的意思将覆盖第二的。

宏指令从它们的定义块生成新的行，用参数的值替换参数，然后处理符号中的转义字符“#”“‘”。转换操作符比连接操作符有更高优先级，如果它们包含转义字符，在完成处理前转义字符将被忽略。当这完成时，新生成的行将执行上面描述的标准预处理。

虽然符号常见通常仅在行中替换，当即没有预处理伪指令也没有宏指令发现时，在这里包含伪指令的替换位置有一些特例。第一是符号常量的定义，在 equ 关键词后的任何地方完成的的替换，结果将被赋给新的常量（见 2.3.2）。第二中情形是 match 伪指令，替换将在匹配模板前逗号后面的符号完成。这些特性可以用来维护列表，如下面的定义：

```
list equ
macro append item
{
  match any, list \{ list equ list,item \}
  match , list \{ list equ item \}
```

```
}
```

list 常量将被初始化为空值，append 宏指令可以增加新的以逗号分隔的项到列表。这个宏指令第一个匹配仅发生在当它们列表值不为空（见 2.3.6），此时新的值为上一次值并且新的值将被添加到后面。第二个匹配只当列表不为空时发生，并且此时列表定义中只包含新值。所以从空表开始，append 1 将定义 list equ 1，后面跟着的 append 2 将定义 list equ 1,2。作为某些伪指令的参数也可以使用。但它不能在直接使用 - 如果 foo 为伪指令，foo list 将作为宏参数传给符号 list，因为符号常量此阶段还没有回滚。

```
match params, list {foo params}
```

如果 list 值不为空，和 params 匹配，当有括号块中定义的新行将会稍后被替换为匹配值。所以如果 list 含有值 1,2，上面的行将生产包含 foo 1,2 的行，这些都会稍后进入标准处理。

还有一个特殊情形 - 当预处理检查行中第二个参数为冒号（如同标号定义那样由汇编器解析），它在当前位置停止并完成第一个符号的预处理（所以如果它是符号常量它将回滚），如果它仍为标号，它将从标号后位置执行标准预处理。这允许在标号后放置预处理伪指令和宏指令，类似汇编器处理指令和伪指令，例如：

```
start: include 'start.inc'
```

然而如果在预处理时断掉（例如当为空的符号常量时），只剩余行中的符号常量将继续。

它必须记住，预处理执行的工作首先为文本符号，他们在主汇编过程前有一个简单的单趟处理。预处理结果后的文本将传给汇编器，然后汇编器将进行多趟操作。因此仅有汇编器识别和处理的控制伪指令 - 它们取决于数值可能在不同趟中改变 - 不能有汇编器以任何方式扫描并且影响预处理。考虑下面的例子：

```
if 0
 a = 1
 b equ 2
end if
dd b
```

当预处理上面的语句，唯一有预处理社别的伪指令为 equ，它定义了符号常量 b，稍后再源码中符号 b 的值被替换为 2.除了这个替换外，对于汇编器其他航没有变化。当预处理后，上面源码将变成：

```
if 0
 a = 1
end if
dd 2
```

此时当汇编器传入它是，if 的条件为 false，常量 a 没有定义。然后符号常量 b 将做普通处理，即使它的定义放在 a 之后。所以每次你必须非常小心的混合伪指令和汇编器特性 – 经常想象你的源码在预处理后将变成什么，汇编器将看到的，以及它的多遍扫描。

2.4 格式伪指令

格式伪指令实际上一些控制伪指令，其目的是控制生成代码的格式。

format 伪指令用来选择输出格式。这条伪指令必须放在源码任何开头。默认的输出格式为 flat 二进制文件，也可以通过使用“format binary”伪指令来选择。

use16 和 use32 伪指令强制汇编器生成 16 位或者 32 位代码，忽略输出格式的默认设置。use64 能够为 x86-64 处理器的长模式生成代码。

下面描述了不同的输出格式以及指定这些格式的伪指令。

2.4.1 MZ 格式

使用“format MZ”伪指令来选择 MZ 输出格式。此格式代码设置默认为 16 位。

segment 伪指令定义定义新段，后面必须跟着标号，其值将成为定义段的数字。可选的 use16 或 use32 可以跟在后面用来指定这个段中的代码为 16 位还是 32 位。段是 16 字节对齐的。所有定义的标号都有一个相对于段开始位置的值。

entry 伪指令用来设置 MZ 可执行文件的入口，它必须指向该入口的远地址（段名，冒号和段中的偏移）。

stack 伪指令为 MZ 可执行文件创建堆栈。它后面可以跟着指定堆栈（将自动创建）大小的数值表达式或者初始堆栈的远地址（如果你想手动设置堆栈）。当没有定义堆栈时，将创建默认的 4096 大小的堆栈。

heap 伪指令应当跟着 16 位定义额外堆最大值（这个堆除堆栈和未定义数据外）。使用“heap 0”用来分配程序实际需要的内存。堆默认大小为 65535。

2.4.2 PE 格式

使用“format PE”伪指令来选择 PE 输出格式。它可以跟着额外的格式设置：使用 console，GUI 或者 native 操作符来选择目标子系统（可以指定子系统版本的浮点数），DLL 表示输出文件为一动态库。然后也可以跟着 at 操作符和指定 PE 文件基地址的数值表达式，然后可选的跟着 on 操作符以及自定义 PE 程序 MZ 头的文件名（如果指定的文件不是 MZ 可执行文件，它将被当做 flat 二进制可执行文件并转化为 MZ 格式）。这个格式默认的代码设置为 32 位。一个所有特性的 PE 格式声明如下：

```
format PE GUI 4.0 DLL at 7000000h on 'stub.exe'
```

创建 x86-64 体系的 PE 文件，使用 PE64 关键词而不是 PE，这种情况下某人生成长模式的代码。

section 伪指令定义一个新的段，它后面必须跟着段名，然后是一个或多个段标志。可能的标志为：
code，data，readable，writeable，executable，shareable，discardable，notpageable。段是页对齐的（4096 字节）。一个声明 PE 段的例子：

```
section '.text' code readable executable
```

除了这些标志也可以指定一些特殊的 PE 数据标识符来标识整个段为特殊数据，可能的标识符有：export，import，resource 和 fixups。如果一个段标识为包含 fixups，重定位数据将自动创建，不需要在这个段中定义任何数据。资源数据也可以从资源文件中自动创建，它可以通过 resource 标识符、from 操作符和文件名来实现。下面是包含特殊 PE 数据段的例子：

```
section '.reloc' data discardable fixups
section '.rsrc' data readable resource from 'my.res'
```

entry 伪指令用来设置 PE 的入口函数，后面跟着入口函数值。

stack 伪指令设置 PE 的堆栈大小，后面跟着保留的堆栈大小，后面可以跟着可选的用逗号隔开的提交堆栈大小。当没有定义堆栈时，默认大小将为 4096 字节。

heap 伪指令用来选择 PE 堆大小，后面应当跟着保留的堆大小，后面可以跟着可选的用逗号隔开的提交堆大小。当没有定义堆时，默认大小将为 65535 字节。当没有指定提交的堆大小时，默认大小为 0。

data 伪指令用来定义特殊的 PE 数据，它后面必须跟着一个或多个数据标识符（export，import，resource 或 fixups）或者 PE 头中的数据项个数。数据必须在另一行定义，并且以“end data”伪指令结束。当选择了 fixups 数据时，它们将自动创建不需

要定义任何额外数据。同样的规则也适用于 resource 标识符后面跟着 from 和文件名 - 此时数据将从给定的资源文件名获取。

rva 操作符可以在数值表达式中使用，用来通过它提供的值获取指定项的 RVA。

2.4.3 COFF 格式

使用“format COFF”（创建简单的 COFF 文件）或“format MS COFF”（创建微软 COFF 文件）伪指令来选择 COFF 格式。这种格式默认的代码为 32 位。创建 x86-64 体系下的微软 COFF 格式需要使用“format MS64 COFF”，此时将生成长模式代码。

section 伪指令定义一个新段，它后面必须跟着段名，然后为一个或多个段标志。两种 COFF 格式的段都可用的标志为 code 和 data，而 readable，writeable，executable，shareable，discardable，notpageable，linkremove 和 linkinfo 只用于微软 COFF 格式。

默认段是 4 字节对齐的，微软的 COFF 格式可以用 align 操作符后面跟着对齐的大小来指定其他对齐（任意不超过 8192 的 2 的指数）。

extrn 伪指令定义外部符号，后面必须跟着符号名和可选的 size 操作数以指定这个符号数据标号的大小。符号名可以包含外部符号名和 as 操作符。可以这样声明一些外部符号：

```
extrn exit
extrn '__imp__MessageBoxA@16' as MessageBox:dword
```

public 伪指令将存在的符号声明为公共的，它必须跟着符号名，可选的 as 操作符和这个符号外部可用的名称。下面是一些公共符号的声明：

```
public main
public start as '_start'
```

2.4.4 ELF 格式

可以使用“format ELF”伪指令选择 ELF 输出格式。这个模式下默认代码设置为 32 位。使用“format ELF64”伪指令创建 x86-64 体系下的 ELF 文件，此时默认将生成长模式代码。

section 伪指令定义一个新段，它必须跟着段名，人啊后是一个或多个 executable 和 writeable 标志，可选的 align 以及指定段对齐数字是可选的（2 的指数），如果没有指定对齐，将使用默认的 4 或 8，取决于选择的格式。

extrn 和 public 伪指令和 COFF 输出格式有着同样的意思和语法（将上一节中的描述）。

这个格式下也可以使用 as 操作符（当目标体系不为 x86-64 时），它将地址转换为相对于 GOT 表格的偏移。还有一个特殊的 plt 操作符，它允许通过过程链接表格来调用外部函数。你甚至可以为外部函数创建别名，使其总能通过 PLT 来调用，如同这样的代码：

```
extrn ’printf’ as _printf
printf = PLT _printf
```

使用跟有 executable 关键词的 format 伪指令来创建可执行文件。它允许使用 entry 伪指令来设置程序入口。另外 extrn 和 public 伪指令将不可用，而且只能使用 segment 伪指令而不是 section，segment 后面跟着一个或多个允许的标志。段是页对齐的（4096 字节），可用的标志有：readable，writeable 和 executable。

FASM 第三章 - Windows 编程

Windows 版本的 FASM 包含一个开发包用来协助开发 Windows 环境下的程序。这个开发包中，根目录下包含头文件、子目录为一些特殊用途的文件。通常 Win32 头文件已经为你包含必须的文件。

有六个 Win32 头文件可供你选择，它们都是以 WIN32 打头后面跟着字母 A（ASCII 编码）或者字母 W(WideChar 编码)，其中：
WIN32A.INC、WIN32W.INC 为基本头文件。
WIN32AX.INC、WIN32WX.INC 为扩展头文件，提供了更多的宏指令，这些扩展将被分别讨论。
WIN32AXP.INC、WIN32WXP.INC 是包含过程调用参数个数检查的扩展头文件。

你可以以任何你喜欢的方式包含这些头文件，全路径或者使用自定义的环境变量，但最简单的方式就是定义 INCLUDE 环境变量指向头文件所在目录，然后就可以这些使用它们：

```
include 'win32a.inc'
```

必须注意的是所有的宏指令和内部伪指令不同，它们都是区分大小写的，并且大多数情况下都使用小写。如果想使用默认外的，必须使用 fix 伪指令来做适当的调整。

3.1 基本头文件

基本头文件 WIN32A.INC 和 WIN32W.INC 包含 Win32 常数和结构定义，还提供了一些标准的宏指令。

3.1.1 结构

所有的头文件都允许使用 struct 宏指令，以比 struc 伪指令更简单的类似其他汇编器的方式来定义结构。结构的定义必须以 struct 宏指令开始，然后是结构名，最后以 ends 结束。在中间值只允许数据定义伪指令，其中的标号将成为结构字段名。

```
struct POINT
  x dd ?
  y dd ?
ends
```

这样定义了 POINT 结构后，就可以这样定义一个 point1 变量：

```
point1 POINT
```

上面这条语句将声明一个包含 point1.x 和 point1.y 的 point1 结构，初始化它们默认的值 - 结构定义中提供的值（在这里默认值都是未初始化的值）。定义结构也可以包含参数，参数个数和结构字段个数相同，指定的参数将覆盖结构定义中的默认值。例如：

```
point2 POINT 10,20
```

上面这条语句将初始化 point2.x 值为 10，point2.y 的值为 20。

struct 宏不仅声明了指定的结构，也为其中的每个元素定义了偏移。例如 POINT 结构定义了 POINT.x 和 POINT.y 标号为 POINT 结构中的偏移，sizeof.POINT.x、sizeof.POINT.y 为相应字段的大小。sizeof.POINT 为结构的大小。标号偏移可以用来间接寻址。比如，当 ebx 包含指向 POINT 结构的指针时：

```
mov eax,[ebx+POINT.x]
```

当这样寻址时 FASM 将检查字段的大小。

结构本身也允许内部结构定义，所以结构中也可以包含其他结构字段

```
struct LINE
  start POINT
  end POINT
ends
```

当子结构中没有指定默认值时，如同上面的定义，将使用子结构定义中的默认值。

既然结构声明中每个字段值都是一个单一参数，为了初始化子结构，必须用尖括号将它们括起来：

```
line1 LINE <0,0>,<100,100>
```

上面这条语句初始化 line1.start.x 和 line1.start.y 值为 0，line1.end.x 和 line1.end.y 为 100。

如果结构定义字段大小比定义中的小，那么将用未定义的字节来填充到定义中大小（当超过时，将产生错误）。例如:

```
struct FOO
 data db 256 dup(?)
ends

some FOO <"ABC",0>
```

将给 some.data 的头四个字节定义数据并且保留剩下的。

结构中也可以定义联合（union）和匿名子结构。联合以 union 开头以 ends 结束，例如：

```
struct BAR
  field_1 dd ?
  union
    field_2  dd ?
    field_2b db ?
  ends
ends
```

union 的每一个字段都有相同的偏移并且共享同一块内存。只有 union 的第一个字段将被初始化，其他字段将被忽略（然后如果其中一些域需要比第一个更大的内存时，union 将用未定义的字节来填充到指定大小）。整个 union 用单一参数来初始化。

匿名子结构定义方式和 union 类似，只是以 struct 开头，例如：

```
struct WBB
  word dw ?
  struct
    byte1 db ?
    byte2 db ?
  ends
ends
```

匿名子结构只允许接收一个参数来初始化，这个参数可以为一组参数，比如上面的结构可以这样定义：

```
my WBB 1,<2,3>
```

union 和匿名子结构字段都可以如同父结构的字段一样来访问。例如上面定义中的 my.byte1 和 my.byte2 都是正确的标号。

子结构和 union 可以无限深度的内嵌：

```
struct LINE
    union
        start POINT
        struct
```

```
        x1 dd ?
        y1 dd ?
      ends
    ends
    union
      end POINT
      struct
        x2 dd ?
        y2 dd ?
      ends
    ends
ends
```

结构定义也可以基于一些已经定义的结构，并且继承此结构所有字段，例如：

```
struct CPOINT POINT
  color dd ?
ends
```

上面的定义等价于：

```
struct CPOINT
  x dd ?
  y dd ?
ends
```

所有头文件都定义了 CHAR 数据类型，可以用来定义数据结构中的字符串。

3.1.2 导入表

import 宏指令用来帮助构造 PE 文件的导入数据（通常将导入表放在一个单独的段中）。有两个相关宏指令： 其一为 library，必须放在导入数据的开头，它定义了哪些库将被导入，后面跟着任何长度的参数对，每一对指定了库的标号以及用引号括起来的库的名称，例如：

```
library kernel32,'kernel32.dll',\
      user32,'user32.dll'
```

上面的语句从 kernel32.dll 和 user32.dll 两个库定义导入数据。对于每个库，导入表必须在导入数据中其它地方定义。这通过 import 宏指令来实现，第一个参数定义标号（同前

面 library 宏中定义相同），每一对参数中包含导入指针和引号括起的函数名（同库导出的函数名相同）。例如上面的 library 定义可以用以下 import 定义来完善：

```
import kernel32,\
    ExitProcess,'ExitProcess'
import user32,\
    MessageBeep,'MessageBeep',\
    MessageBox,'MessageBoxA'
```

每一对参数中的第一个参数将传给 import 宏一个 DWORD 指针地址，当载入 PE 后将被填充为导出函数的地址。

如果不用字符串来导入函数，也可以使用序号数字来定义导入函数，例如：

```
import custom,\
    ByName,'FunctionName',\
    ByOrdinal,17
```

导入宏优化了导入数据，使得只有程序中实际使用了的导入函数才会被放到导入表中，而且假如没有使用某个库的任何函数，那么整个库都不会被引用。这样我们可以很方便让每个库包含完整的导入表 - 对于一些常见的库开发包中包含这样的导入表，他们保存在 APIA 和 APIW 子目录下。每一个文件包含一个导入表，小写的文件名作为标号。一个导入 KERNEL32.DLL 和 USER32.DLL 库完整导入表的定义如下（假设已经设置好 INCLUDE 环境变量到开发包到 include 目录）：

```
library kernel32,'KERNEL32.DLL',\
    user32,'USER32.DLL'
include 'apia\kernel32.inc'
include 'apiw\user32.inc'
```

3.1.3 过程

有四个宏指令用于带参数传递的过程调用。stdcall 声明第一个参数指定的函数使用 STDCALL 方式的函数调用，其他参数为过程的参数，且以相反的方式压栈。invoke 宏和 stdcall 类似，只是通过第一个参数的标号间接的调用过程，因此 invoke 可以用来调用导入表中的过程：

```
invoke MessageBox,0,szText,szCaption,MB_OK
```

等价于：

```
stdcall [MessageBox],0,szText,szCaption,MB_OK
```

它们都将生成下面的代码:

```
push MB_OK
push szCaption
push szText
push 0
call [MessageBox]
```

ccall、cinvoke 同 stdcall、invoke 类似，只是它们使用 C 调用方式，必须由调用者来负责清理堆栈。

定义一个使用参数堆栈和局部变量的过程，可以使用 proc 宏指令。最简单的格式就是过程名后面跟着所有的参数名，如同：

```
proc WindowProc,hwnd,wmsg,wparam,lparam
```

过程名和参数之间的逗号是可选的。过程指令必须另起一行，并且以 endp 结束 proc 宏指令。堆栈帧自动在过程入口创建，EBP 作为基址来访问参数，所以应当避免把这个寄存器用作其他用途。参数名用来定义基于 EBP 的标号，可以类似变量那样访问过程参数。例如 mov eax,[hwnd]指令等同于 mov eax,[ebp+8]。这些标号的范围仅限为本过程。

这样每一个压栈的参数都是 DWORD 类型的，你也可以自定义参数大小：在参数后面跟着冒号和 size 操作符。上面的例子可等价的写为：

```
proc WindowProc,hwnd:DWORD,wmsg:DWORD,wparam:DWORD,lparam:DWORD
```

如果给定大小小于 DWORD，则该标号应用于压栈的 DWORD 的一部分。如果给定大小大于 DWORD，例如四字指针，两个 DWORD 参数将用来保存这个值，并且标识为一个标号。

过程可以跟着 stdcall 或者 c 关键字，用来定义函数调用约定，默认为 stdcall。也可以包含 uses 关键字，其后跟着的一串寄存器（空格分隔）将会自动在过程入口保存并且在退出

时恢复。如果这样寄存器列表和第一个参数之间就需要逗号了。一个完整格式的过程语句类似这样：

```
proc WindowProc stdcall uses ebx esi edi,\
  hwnd:DWORD,wmsg:DWORD,wparam:DWORD,lparam:DWORD
```

local 宏指令定义定义局部变量，其后跟着一个或多个逗号分隔的声明，每一个都包含变量名、冒号以及变量类型 - 类型既可以为标准类型（必须为大写）或者结构名。例如：

```
local hDC:DWORD,rc:RECT
```

定义局部数组，变量名称后面跟着用中括号括起的数组大小，例如：

```
local str[256]:BYTE
```

另一定义局部变量的方式是在一个 locals 宏指令开头、endl 结尾的块中声明局部变量，这样就可以像定义普通数据那样定义局部变量了。上面的的例子可以等价为：

```
locals
    hDC dd ?
    rc RECT
endl
```

局部变量可以在过程中任意地方定义，唯一限制是必须在使用它们之前定义。变量标号的定义域限制为这个过程。如果你给局部变量一些初始值，宏指令将生成指令初始化这些变量，而且这些指令的位置将和它们的声明所在位置相同。

ret 可以放置在过程中任意地方，生成完整的必须的代码以正确的退出过程、恢复堆栈以及过程中使用的寄存器。如果你需要生成原始的返回指令，使用 retn 助记符，或者带有数字参数 ret，这样它们将被解析为单一指令。

综述：一个完整的过程定义如同这样：

```
proc WindowProc uses ebx esi edi,hwnd,wmsg,wparam,lparam
  local hDC:DWORD,rc:RECT
  ; the instructions
  ret
endp
```

3.1.4 导出表

export 宏指令用来构造 PE 文件的导出数据（它必须放在一个标识为 export 的段中，或者在一个 data export 块中）。第一个参数为库名称，其余的为任何数目的参数对。每对参数的第一个为源码某地定义的过程名，第二个参数导出函数名。例如：

```
export 'MYLIB.DLL',\
  MyStart,'Start',\
  MyStop,'Stop'
```

上面的语句将用源码中的 MyStart 和 MyStop 导出两个函数 Start 和 Stop。由于 PE 结构的需要，宏指令还会自动排序导出表。

1.

3.1.5 COM（组件）

interface 宏用来声明 COM 对象指针，第一个参数为接口名，然后是一串方法，例如：

```
interface ITaskBarList,\
  QueryInterface,\
  AddRef,\
  Release,\
  HrInit,\
  AddTab,\
  DeleteTab,\
  ActivateTab,\
  SetActiveAlt
```

comcall 宏用来调用给定对象的方法。该宏的第一个参数为对象句柄，第二个参数为该对象实现的 COM 接口名，以及方法名和方法参数。例如：

```
comcall ebx,ITaskBarList,ActivateTab,[hwnd]
```

使用 ebx 寄存器存放 COM 对象，ITaskBarList 为接口，调用 ActivateTab 方法，以[hwnd]为参数。

COM 接口名可以类似结构名那样使用，定义变量用来保存给定类型对象的句柄：

```
ShellTaskBar ITaskBarList
```

上面一行定义了一个 DWORD 变量，其中可以存放 COM 对象句柄。当其中存放了对象句柄后，就可以使用 cominvk 来调用它的方法。cominvk 只需要存放接口的变量名以及方

法名作为头两个参数，后面为方法的参数。一个对象句柄保存到 ShellTaskBar 变量中，调用对象 ActivateTab 方法可以这样调用：

```
cominvk ShellTaskBar,ActivateTab,[hwnd]
```

等同于：

```
comcall [ShellTaskBar],ITaskBarList,ActivateTab,[hwnd]
```

3.1.6 资源

有两种方法定义资源，一种是包含外部资源文件，另一种是手动创建资源段。后一种方法，虽然不需要借助任何外部程序，但比较费时。标准头文件提供了一套基本的宏指令帮助组合资源段。

directory 宏指令必须放在手动构造资源数据的开头，它定义了包含了哪些类型的资源。它后面跟着一对数据，第一个为资源类型标识，第二个为资源类型子目录的标号。例如：

```
directory RT_MENU,menus,\
  RT_ICON,icons,\
  RT_GROUP_ICON,group_icons
```

子目录可以放在资源区域内主目录后面的任何位置，而且他们必须使用 resource 宏指令来定义：第一个参数为子目录标号（对应主目录中的标号）后面跟着三个参数 - 每一项中第一个参数定义了资源标识符（这个值可以随意选择，其后程序使用该标识符来访问该资源），第二个参数指定语言，第三个为资源标号。标准的常数可以用来创建资源标识符。例如菜单子目录可以这样定义：

```
resource menus,\
  1,LANG_ENGLISH+SUBLANG_DEFAULT,main_menu,\
  2,LANG_ENGLISH+SUBLANG_DEFAULT,other_menu
```

如果资源是语言类型无关的，应当使用 LANG_NEUTRAL 标识符。有特殊的宏指令来定义各种不同的资源，应当放在资源区域中。

位图资源使用 RT_BITMAP 类型标识符。定义位图资源可以使用 bitmap 宏指令，后面第一个参数为资源标号（和位图子目录项对应），第二个参数为位图文件路径，如同：

```
bitmap program_logo,'logo.bmp'
```

有两个资源类型和图标有关，RT_GROUP_ICON 为资源类型，链接一个或多个 RT_ICON 类型的资源，每一个都包含一单一图片。它允许在同一资源标识符下定义不同大小和色深

的图片。这个 RT_GROUP_ICON 类型的标识符稍后可以传给 LoadIcon 函数，它可以选择图片的尺度。定义图标可以使用 icon 宏指令，第一个参数为 GT_GROUP_ICON 资源标号，后面为声明图片的参数对。每个参数对的第一个为 RT_ICON 资源标号，第二个为图标文件路径。当定义一个图标只包含一个图片，可以这样：

```
icon main_icon,icon_data,'main.ico'
```

菜单为 RT_MENU 资源类型，并且使用 menu 宏指令来定义。menu 本身只接收一个参数 - 资源标号。menuitem 定义了菜单中的项，它接收五个参数，但只有两个是必须的 - 第一个参数为菜单项名称，第二个为标识符（当用户从菜单上选择一菜单项时将返回此值）。menuseparator 不接收任何参数，用来定义菜单中的分隔符。

可选的第三个参数 menuitem 指定了菜单资源的标志。有两个这样的标志可供选择 - MFR_END 为最后菜单项的标志，MFR_POPUP 表明指定菜单项为子菜单，而且稍后直到 MFR_END 标志为止将组成子菜单。MFR_END 标志也可以作为 memseparator 的参数，而且这个宏指令只接受一个参数。为了完整的定义菜单，每一个子菜单都应当用 MFR_END 标志结束，而且整个菜单也必须这样结束。下面是一个完整的菜单定义：

```
menu main_menu
  menuitem '&File',100,MFR_POPUP
  menuitem '&New',101
  menuseparator
  menuitem 'E&xit',109,MFR_END
  menuitem '&Help',900,MFR_POPUP + MFR_END
  menuitem '&About...',901,MFR_END
```

可选的第四个参数 menuitem 指定了给定菜单项的状态标志，这些标志和 API 函数中使用的相同，比如：MFS_CHECK、MFS_DISABLED。类似，第五个参数为类型标志。例如下面定义了一个选中的单选按钮：

```
menuitem 'Selection',102, ,MFS_CHECKED,MFT_RADIOCHECK
```

对话框为 RT_DIALOG 资源类型，并且使用 dialog 宏指令后面跟着任何数目的 dialogitem 开头 enddialog 结尾的项来定义。

dialog 接收十一个参数，只有前七个是必须的。第一个为资源标号，第二个为对话框标题字符串，后面四个参数为水平和垂直坐标、对话框的宽度和高度。第七个参数为对话框窗口样式，可选的第八个参数为对话框的扩展样式。第九个参数为窗口菜单 - 必须为菜单资源的标识符，和子目录中指定的 RT_MENU 类型相同。最后第十和第十一个参数用来定义对话框的字体 - 其中的第一个为字体名称的字符串，后面的为字体大小。当没有可选的参数时，将使用默认的 MS Sans Serif 的 8 号字体。

下面这个例子为 dialog 宏指令，除了 menu（为空值）外给出了所有参数，可选部分在第二行。

```
dialog about,'About',50,50,200,100,WS_CAPTION+WS_SYSMENU,\
  WS_EX_TOPMOST, ,'Times New Roman',10
```

dialogitem 有 8 个必须参数和一个可选参数。第一个参数为窗口类。第二个参数既可以为对话框项字符串或者资源的标识符（当对话框项必须使用另外的一些资源定义，例如包含 SS_BITMAP 样式的 STATIC 类）。第三个参数为对话框项的标识符，用作 API 函数来标识此项。另外的四个参数指定了水平、垂直坐标，宽度和高度。第八个参数为样式，可选的第九个参数为扩展样式。一个对话框项的定义如下：

```
dialogitem 'BUTTON','OK',IDOK,8,8,45,15,WS_VISIBLE+WS_TABSTOP
```

一个包含位图的 static 项，假设存在一个标识符为 7 的位图：

```
dialogitem 'STATIC',7,0,10,50,50,20,WS_VISIBLE+SS_BITMAP
```

对话框资源的定义可以包含任何数据的对话框项或者没有，最后必须使用 enddialog 宏指令来结束。

RT_ACCELERATOR 资源类型使用 accelerator 宏指令来创建。第一个参数为资源标号，它们必须跟着三个参数组成的参数对 - accelerator 标志，虚拟键或者 ASCII 字符，以及标识符（和菜单项标识符相同）。一个简单的 accelerator 定义可以如下：

```
accelerator main_keys,\
    FVIRTKEY+FNOINVERT,VK_F1,901,\
    FVIRTKEY+FNOINVERT,VK_F10,109
```

版本信息为 RT_VERSION 资源类型，使用 versioninfo 宏指令来创建。在资源标号后面，第二个参数指定了 PE 文件的操作系统（通常值为 VOS_WINDOWS32），第三个参数为文件类型（对于可执行程序为 VFT_APP，对于动态库为 VFT_DLL），第四个为子类型（通常为 VFT2_UNKNOWN），第五个为语言标识符，第六个为代码页，后面为字符串参数，为属性名称和对应值组。最简单的版本定义如下：

```
versioninfo vinfo,VOS__WINDOWS32,VFT_APP,VFT2_UNKNOWN,\
    LANG_ENGLISH+SUBLANG_DEFAULT,0,\
    'FileDescription','Description of program',\
    'LegalCopyright','Copyright et cetera',\
    'FileVersion','1.0',\
    'ProductVersion','1.0'
```

其它类型的资源可以使用 resdata 宏指令来定义，resdata 只接受一个参数 - 资源标号，后面可以用任何指令来定义数据，最后用 endres 宏指令结束，例如：

```
resdata manifest
    file ’manifest.xml’
endres
```

3.1.7 字符编码

resource 宏指令使用 du 指令来定义资源中的 unicode 字符串 - 然后这个伪指令只是简单的零扩展字符为 16 位，对于包含非 ASCII 字符的字符串，du 可能需要重新定义。对于一些编码，在 ENCODING 子目录下，宏指令重新定义了 du 操作符来生成正确的 UNICODE 字符串。比如：如果源码是以 Windows 1250 代码页，下面这一行必须放在文件开头：

```
include ’encoding\win1250.inc’
```

3.2 扩展头文件

扩展头文件 WIN32AX.INC 和 WIN32WX.INC 提供了基本头文件的所有功能，还包括一些复杂的宏指令。而且如果没有声明 PE format 的话，扩展头文件也将自动声明之。WIN32AXP.INC 和 WIN32WXP.INC 是扩展头文件的另一种，他们执行额外的过程调用参数个数检查。

3.2.1 过程参数

在扩展头文件下，调用过程的宏指令允许比基本头文件中头文件中的 DWORD 数据更多参数类型。首先，当一个字符串作为参数时，通常需要定义一个字符串数据，然后传给过程一个字符串指针。

```
invoke MessageBox,HWND_DESKTOP,"Message","Caption",MB_OK
```

如果参数前有 addr，它的意思是这个值为一个 DWORD 地址并且将传给过程，即使不能直接传入 - 比如局部变量是基于 EBP 寻址的，在这种情况下 EDX 将临时用来计算地址，并且传给过程，例如：

```
invoke RegisterClass,addr wc
```

当 wc 为局部变量，地址为 ebp-100h，将生成下面的指令：

```
lea edx,[ebp-100h]
push edx
call [RegisterClass]
```

当给定的地址没有任何关联寄存器时，它将直接保存。

如果参数前面包含 double，它将被当做 64 位值，将被当做两个 32 位值传递。例如：

```
invoke glColor3d,double 1.0,double 0.1,double 0.1
```

上面的语句将传入三个 64 位参数。如果 double 参数为以内存操作数，不允许包含大小运算符，因为 double 已经包含了 size 覆盖。

最后，过程调用可以嵌套，也就是说一个过程可以当作另一个过程的参数。这种情况下，EAX 中的返回值将传递给嵌套的那个函数，例如：

```
invoke MessageBox,,\
"Message","Caption",MB_OK
```

函数嵌套没有层数限制。

3.2.2 结构化源码

扩展头文件中包含一些宏指令用来帮助简化源码结构。.data 和.code 为定义数据段和代码段的快捷方式。.end 宏指令应当放在程序末尾，带有一个指定程序入口的参数，而且它将自动使用标准的导入表创建导入段数据。

.if 宏指令生成一些在执行期间检查的条件语句，根据执行结果决定继续执行下面的块还是跳过。块必须以.endif 结尾。

条件可以使用比较运算符 - =, <, >, <=, >=, 和 <>。第一个必须为寄存器或者内存操作数。比较将执行无符号比较。如果你只提供了一个值作为条件，那么它将和零做比较，例如：

```
.if eax
ret
.endif
```

上面的语句当 EAX 为零时跳过执行 ret。

还有一些特殊的符号用来作为条件：ZERO？当 ZF 为 1 时为 true，类似的还有 CARRY?、SIGN?、OVERFLOW?和 PARITY?，分别对应 CF、SF、OF 和 PF 标志。

上面简单的条件可以组合成复杂的条件表达式：&表示条件与、|条件或、~用来取反，以及括号。例如：

```
.if eax<=100 & ( ecx | edx )
inc ebx
.endif
```

上面的语句将生成比较和跳转指令，当 EAX 小于或等于 100 并且至少 ECX 或者 EDX 不为零时执行给定的块。

.while 宏指令生成的指令只要条件为 true 将重复执行给定块（.while 宏指令必须以.endw 结尾）。.repeat 和.until 宏指令执行给定块直到.until 后面的条件满足。例如：

```
.repeat
add ecx,2
.until ecx>100
```