Contents

[x86处理器架构 2](#_Toc34572699)

[32位x86处理器架构 2](#_Toc34572700)

[x86-64处理器架构 3](#_Toc34572701)

[汇编语言基础 4](#_Toc34572702)

[常用汇编器 4](#_Toc34572703)

[汇编语言常量 5](#_Toc34572704)

[汇编语言保留字/关键字 5](#_Toc34572705)

[标识符identifier 5](#_Toc34572706)

[伪指令directive 5](#_Toc34572707)

[指令 7](#_Toc34572708)

[列表文件listing file 8](#_Toc34572709)

[数据类型和数据定义 8](#_Toc34572710)

[数据操作相关运算符和指令 10](#_Toc34572711)

[操作数类型 10](#_Toc34572712)

[MOV、MOVZX、MOVSX指令 10](#_Toc34572713)

[LAHF、SAHF指令 11](#_Toc34572714)

[XCHG指令 11](#_Toc34572715)

[加减运算和相关指令 11](#_Toc34572716)

[OFFSET运算符 12](#_Toc34572717)

[ALIGH伪指令 12](#_Toc34572718)

[PTR运算符 12](#_Toc34572719)

[TYPE运算符 12](#_Toc34572720)

[LENGTHOF运算符 13](#_Toc34572721)

[LABEL伪指令 13](#_Toc34572722)

[TYPEDEF运算符 13](#_Toc34572723)

[汇编语言条件判断 13](#_Toc34572724)

[布尔和比较指令 13](#_Toc34572725)

[检查奇偶标志 15](#_Toc34572726)

[置位和清除单个CPU标志 15](#_Toc34572727)

[JMP指令 16](#_Toc34572728)

[条件跳转指令 16](#_Toc34572729)

[LOOP相关指令 17](#_Toc34572730)

[32位条件控制流伪指令 18](#_Toc34572731)

[汇编语言过程 19](#_Toc34572732)

[PUSH和POP及相关指令 19](#_Toc34572733)

[CALL和RET指令 19](#_Toc34572734)

[USES运算符 19](#_Toc34572735)

[汇编语言高级过程 19](#_Toc34572736)

[调用规范 19](#_Toc34572737)

[LEA指令 21](#_Toc34572738)

[ENTER和LEAVE指令 21](#_Toc34572739)

[LOCAL伪指令 21](#_Toc34572740)

[INVOKE伪指令 22](#_Toc34572741)

[ADDR运算符 22](#_Toc34572742)

[PROC和ENDP伪指令 22](#_Toc34572743)

[PROTO伪指令 23](#_Toc34572744)

[EXTERN伪指令 23](#_Toc34572745)

[Java虚拟机JVM工作原理 24](#_Toc34572746)

[高级语言接口 24](#_Toc34572747)

[.MODEL伪指令 24](#_Toc34572748)

[\_\_asm伪指令 25](#_Toc34572749)

[部分汇编器特殊语法积累 25](#_Toc34572750)

[MASM 25](#_Toc34572751)

[Visual Studio常用功能积累 26](#_Toc34572752)

# 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，任何文本都可以岀现在<...>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

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岀端口。

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）在内存中为变量留岀存储空间，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] ;寄存器内容为索引乘数据宽度（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

·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。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。

## 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 指令总是清除溢岀和进位标志位，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,64Clh ;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 B₀ 到 B₃ 那么计算奇偶性的表达式为:B₀ XOR B₁ XOR B₂ XOR B₃。

## 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

下表列岀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，则跳转到由目标给岀的标号。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 1xprl 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。

# 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

·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。

·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。

将所有章节的visual studio搜索内容归此