### 零标志（ZF, Zero Flag）：如果运算结果为零，则该标志位被设置为1。这通常用于判断条件语句，如比较操作后是否得到了零值

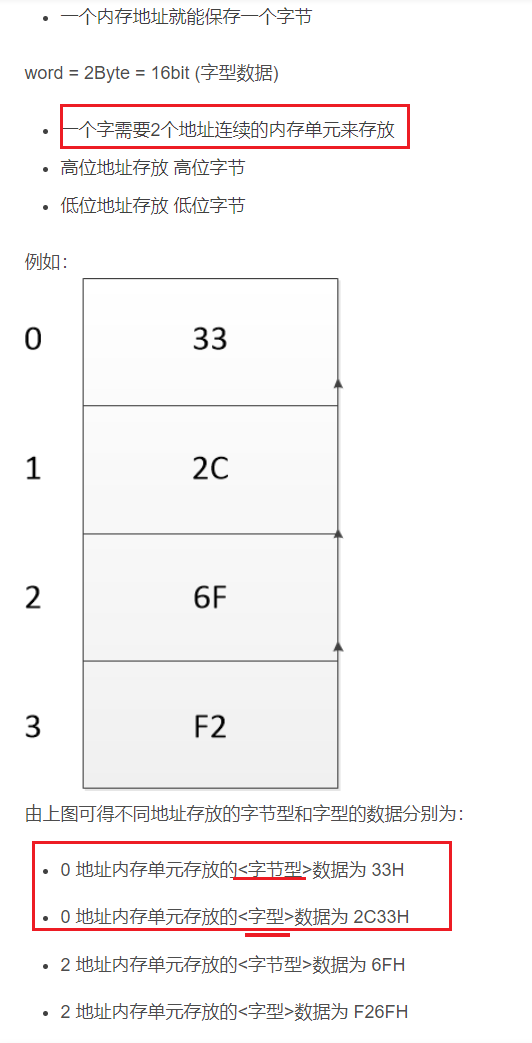
### 溢出标志（OF, Overflow Flag）：在带符号数运算中，如果结果超出了该数据类型所能表示的范围，则OF被设置为1。对于无符号数运算，OF没有意义。例如，两个正数相加得到一个负数，或两个负数相加得到一个正数，都会导致OF被设置

### 符号标志（SF, Sign Flag）：如果运算结果为负，则SF被设置为1。这通常用于判断运算结果的符号位，即最高位（符号位）是否为1

### 进位/借位标志（CF, Carry Flag）：在无符号数运算中，如果最高位产生了进位或借位，则CF被设置为1。在加法运算中，如果最高位有进位输出，则CF为1；在减法运算中，如果有借位发生，则CF也为1。CF对于有符号数运算没有意义

**符号和零标志位不可能同时被设置，一般CF和ZF搭配，SF和OF搭配**

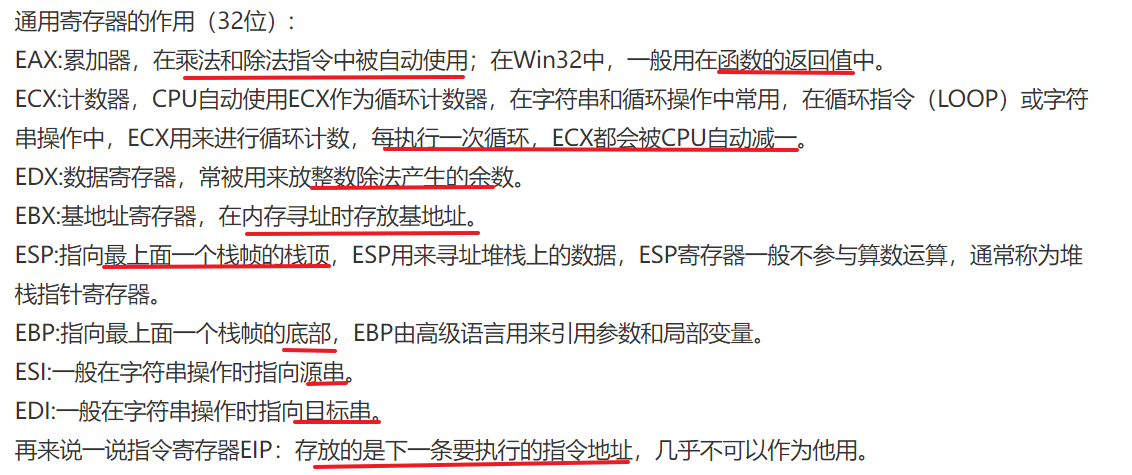
AX = AH(High) + AL(Low) = AX可以存放一个字型数据 = 存放2个字节型数据

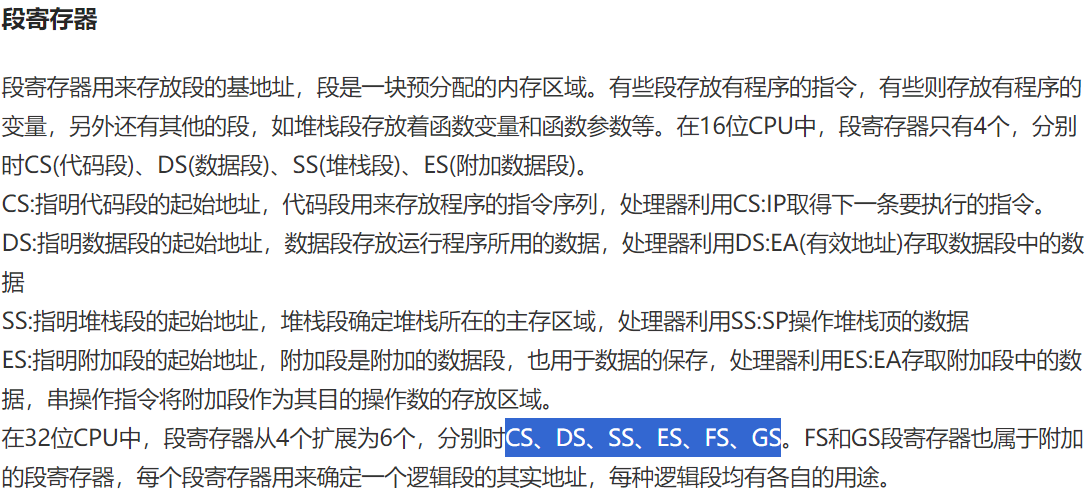






ESP为堆栈指针寄存器，指示**栈顶的偏移地址**





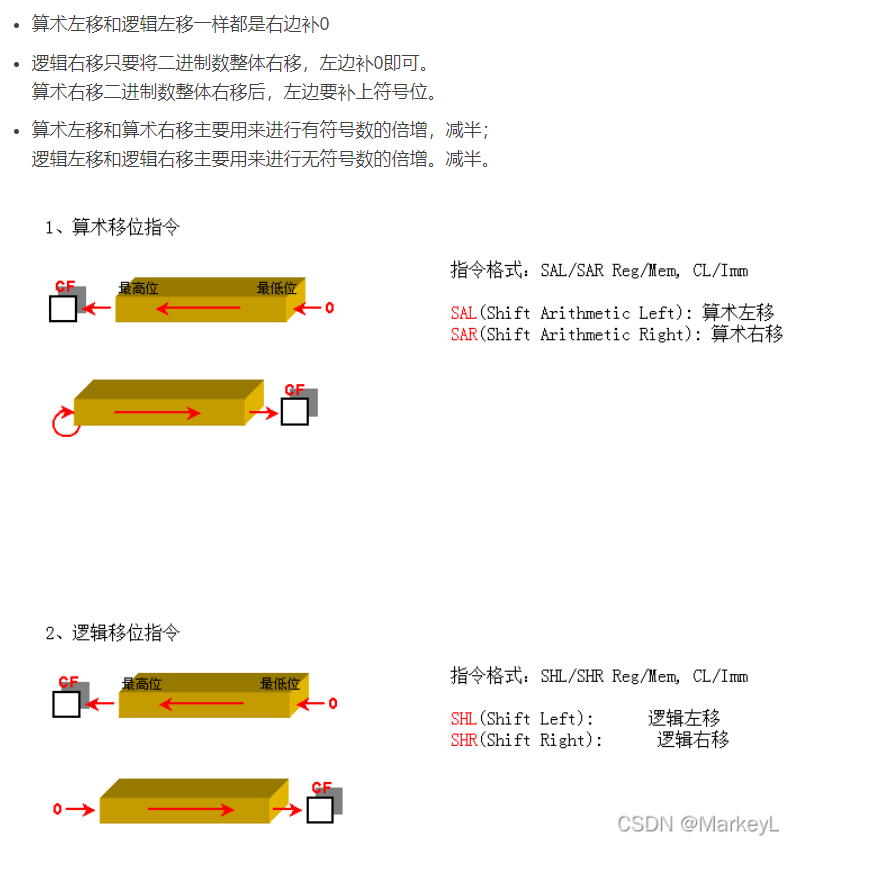


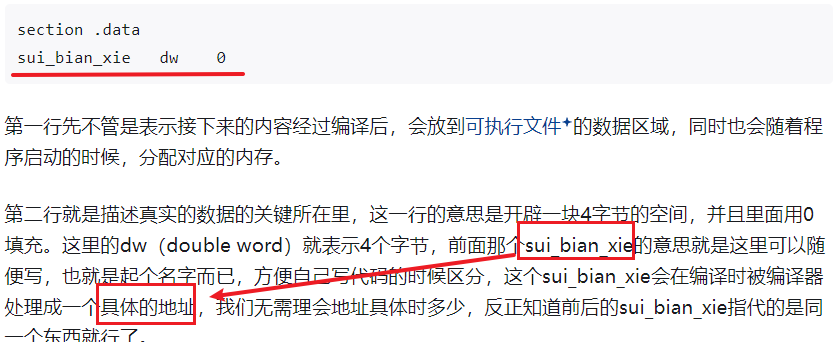
ROL（循坏左移）、ROR（循坏右移）移出的位复制到CF、

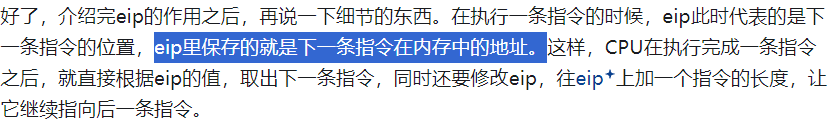
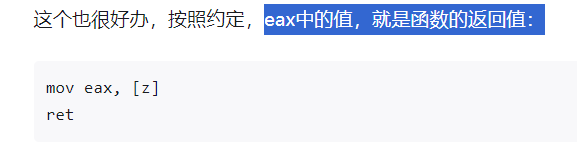
RCL、RCR、首尾间加个CF

SAL（算数左移）、SAR（算数右移）、左移补0，右移补符号

SHL（逻辑左移）、SHR（逻辑右移），空位补0



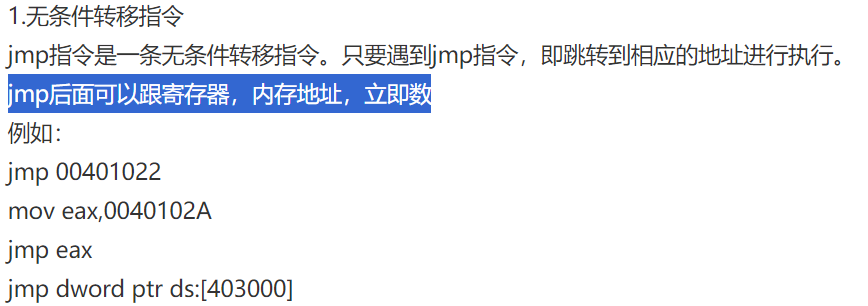


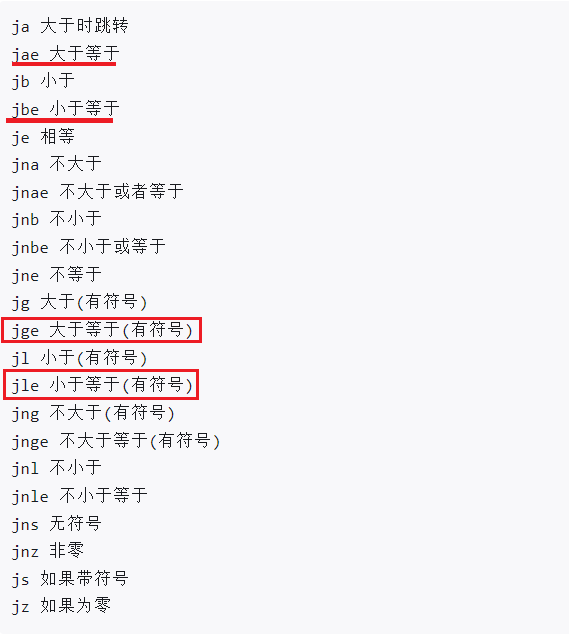


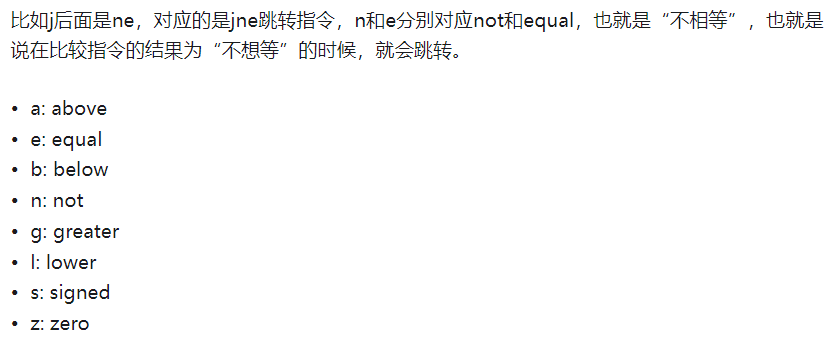




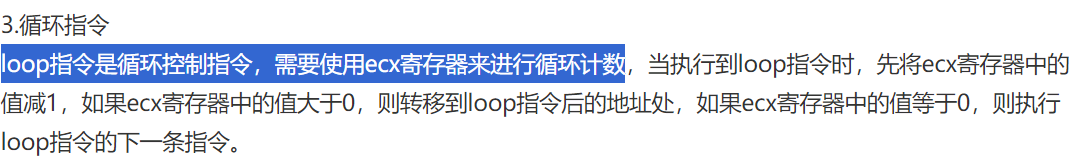
**跳转指令：**





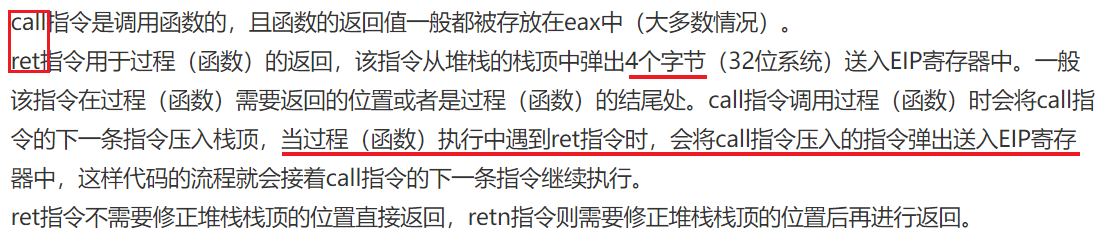


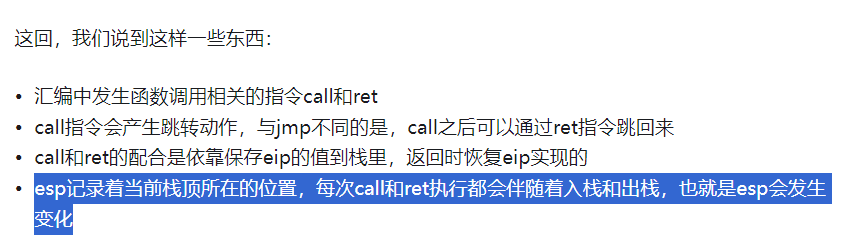
**循环指令（Loop）：**

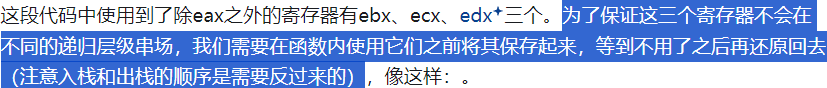


**调用过程（函数）指令和返回指令：（call和ret）**

**与jmp指令不同的地方是，call指令在修改EIP寄存器的值之前，会将call指令的下一条指令的地址保存至堆栈，以便在调用过程（函数）执行完之后再返回到原来call指令处执行下一条指令。**



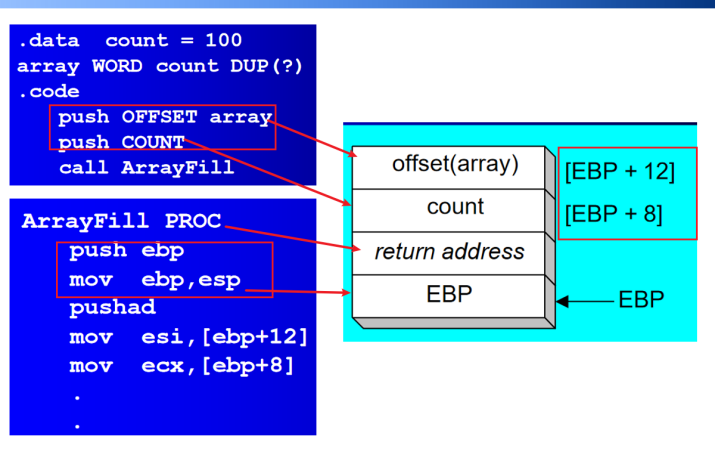
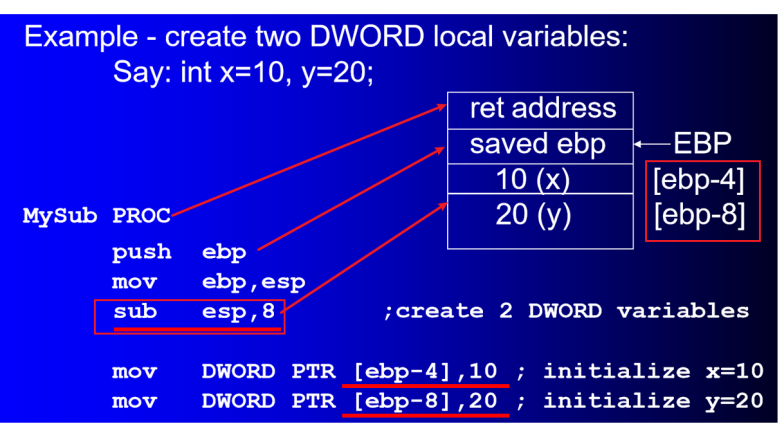




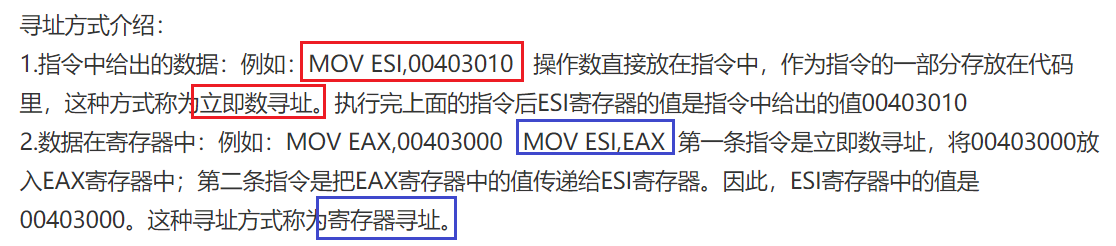
注意：pop后是还原到push进栈之前的值

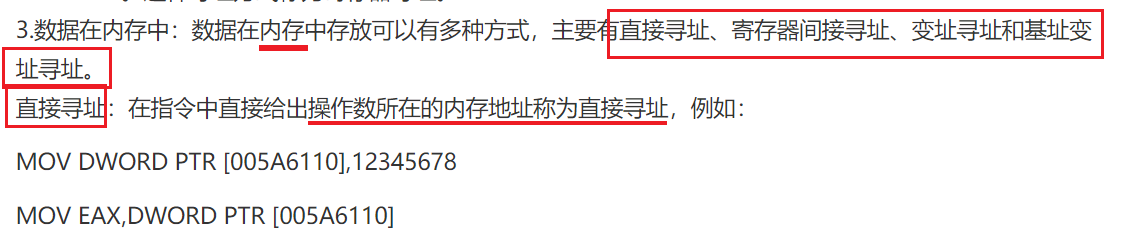


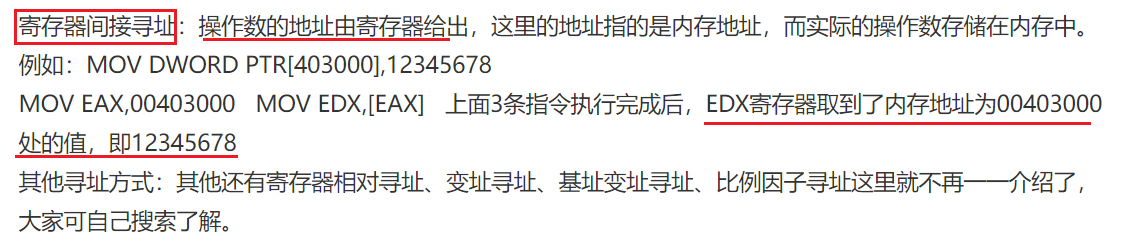
寄存器子过程中，压，别的实参在main中压

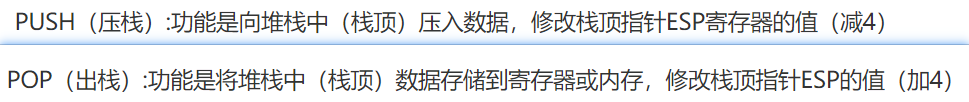


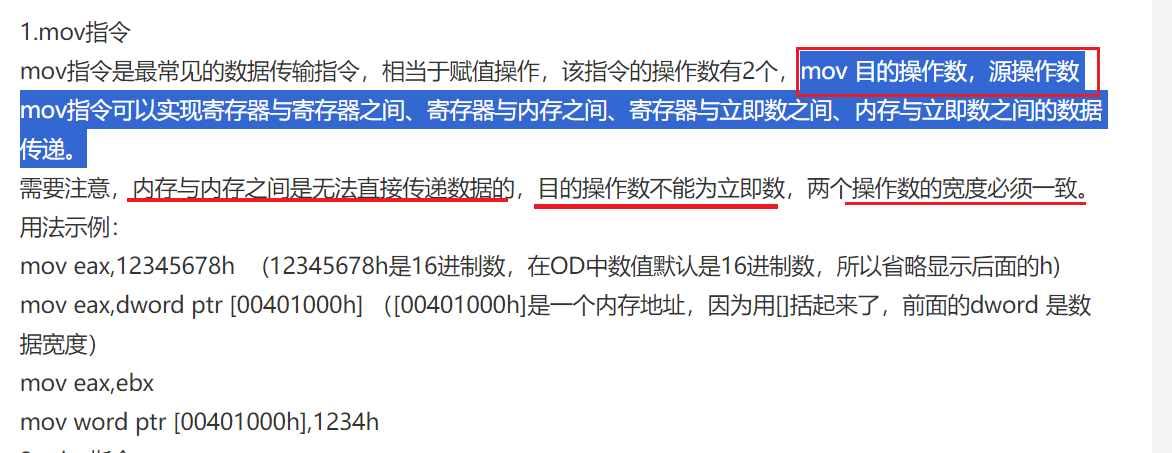
**寻址方式：**



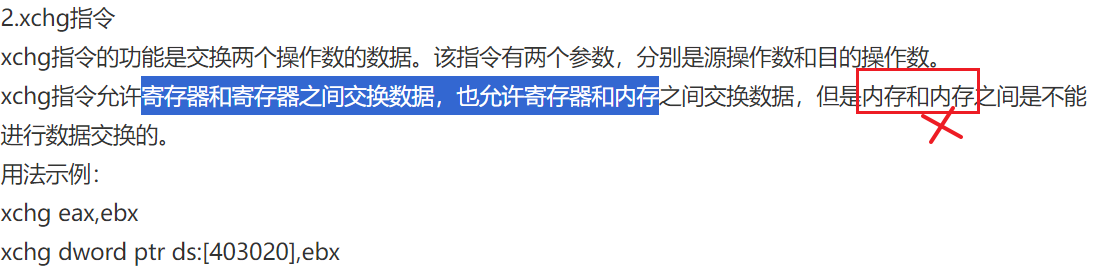




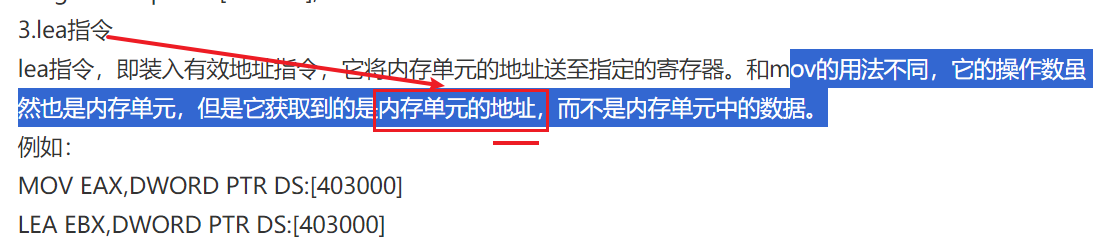


**Mov指令：**  


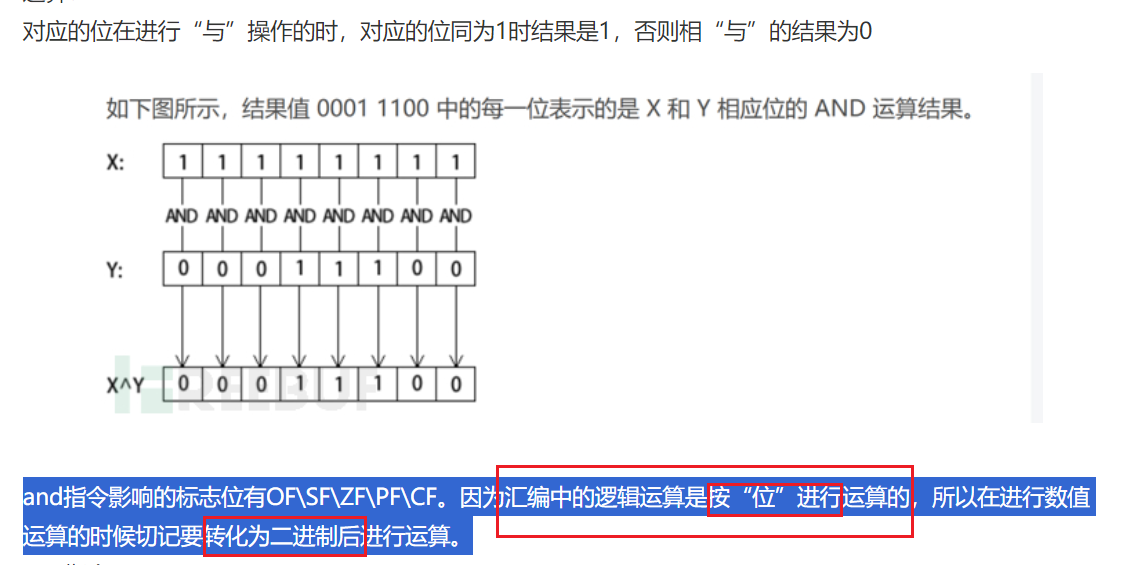
**Xch指令：**



**Lea指令：**

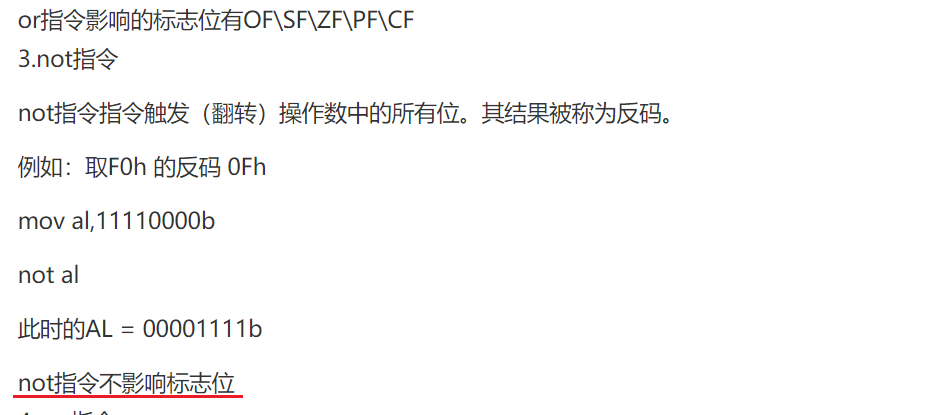


**And指令;**

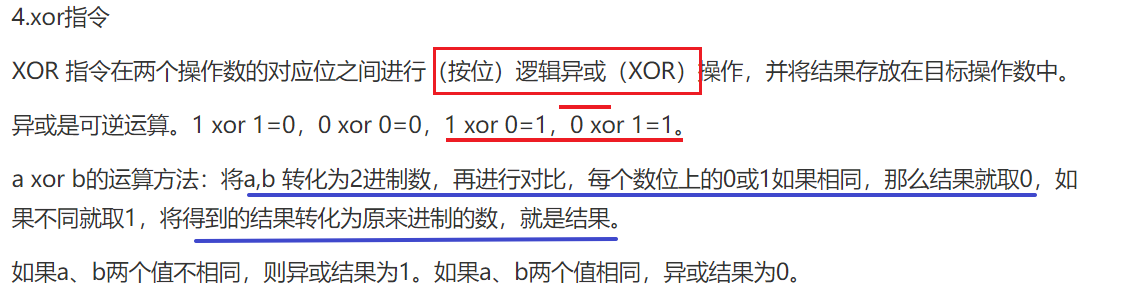


**Or指令：（或运算）按位进行**

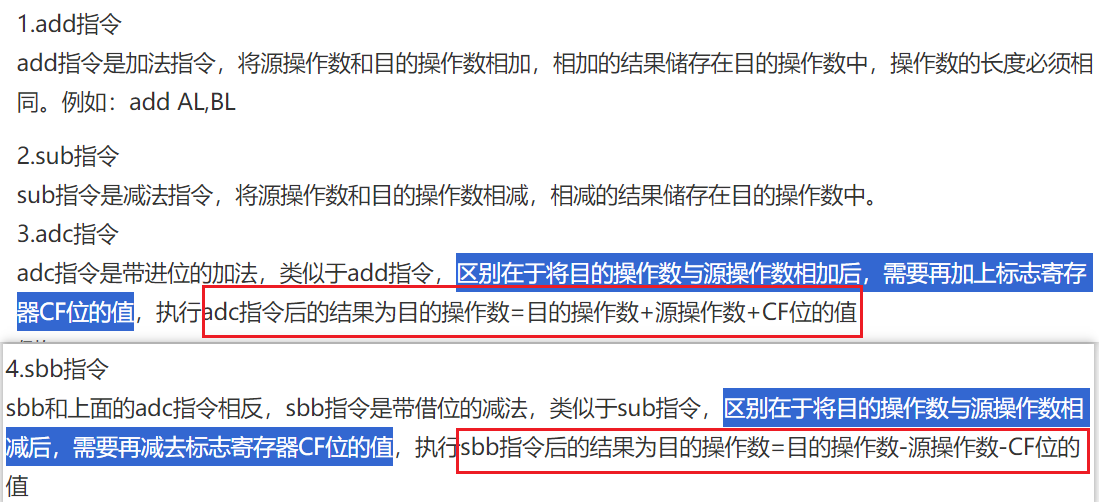
**Not指令:**（非）



**Xor指令：（异或）**



**Add、adc、sub、sbb指令;**



**Inc、dec指令：**

inc指令是加一指令，用于对目的操作数进行加一操作。

Dec指令是减一指令，用于对目的操作数进行减一操作。

### 1. ****寄存器传参****

**定义**：参数通过寄存器传递，而不是通过内存或堆栈。这种方式在函数调用时效率最高。

* 快速，因为寄存器访问速度比内存快。
* 减少内存操作

### 2. ****变量传参数****

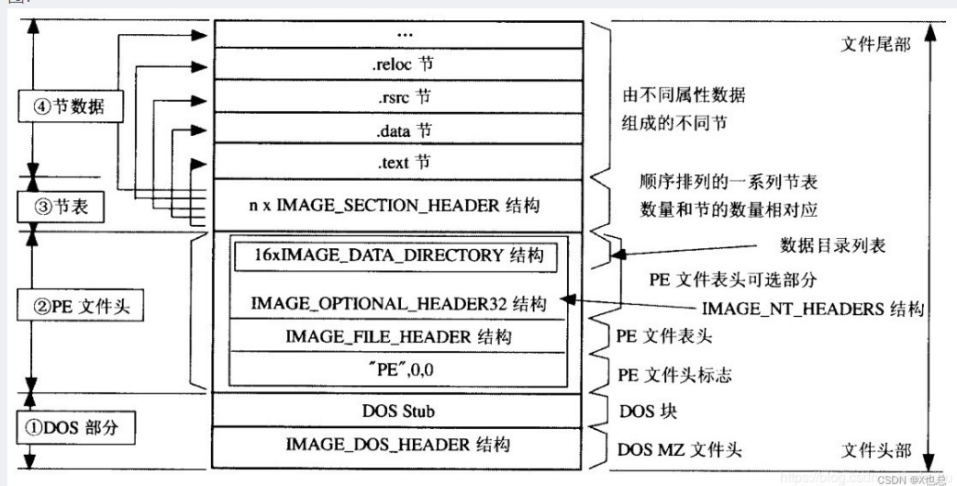
**定义**：参数直接通过内存中的变量传递，而不是寄存器或地址。这种方式常见于简单函数调用，变量通常定义在 .DATA 段。

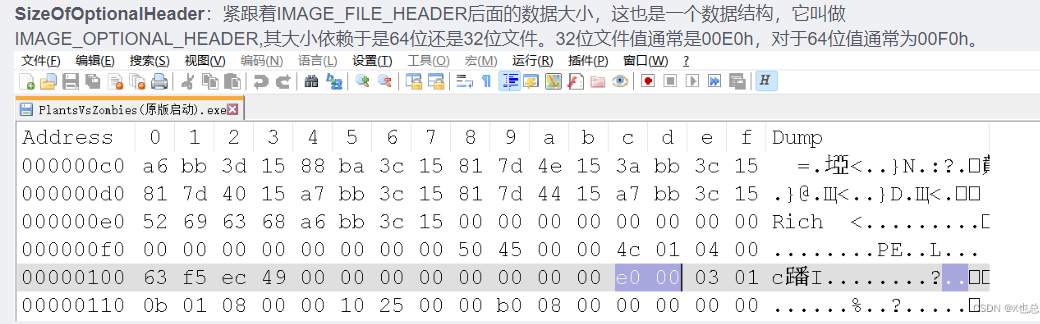
* 易于理解和使用。
* 不依赖寄存器数量。

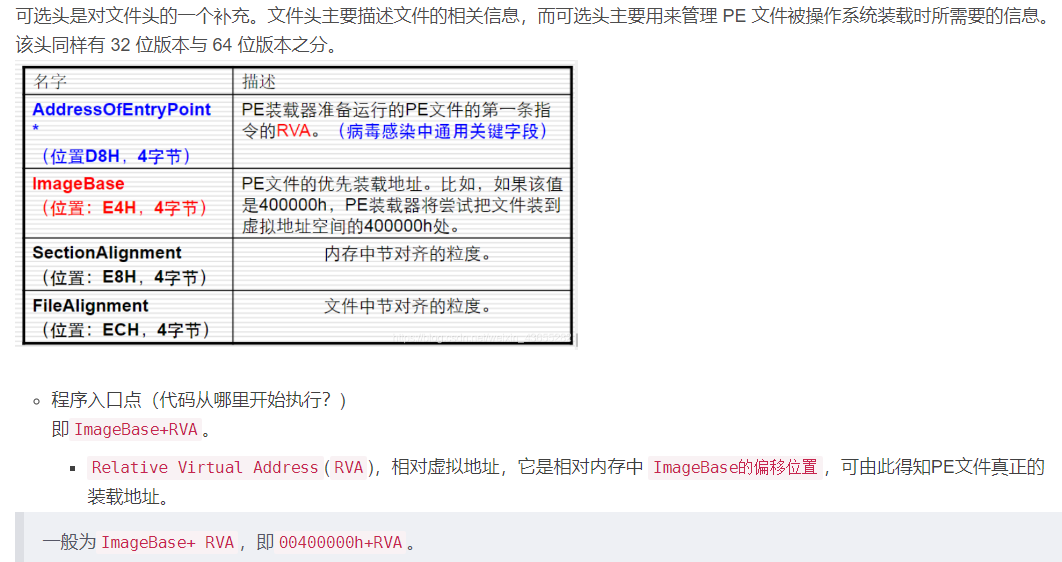
### 3. ****变量地址传参数****

**定义**：参数通过内存地址传递到函数中，函数操作该地址指向的值。这种方式通常用于需要函数修改调用者变量值的场景。

* 可以在函数中修改调用者的变量值。
* 适用于大型数据（如数组、结构体）的传递，避免值传递的开销。

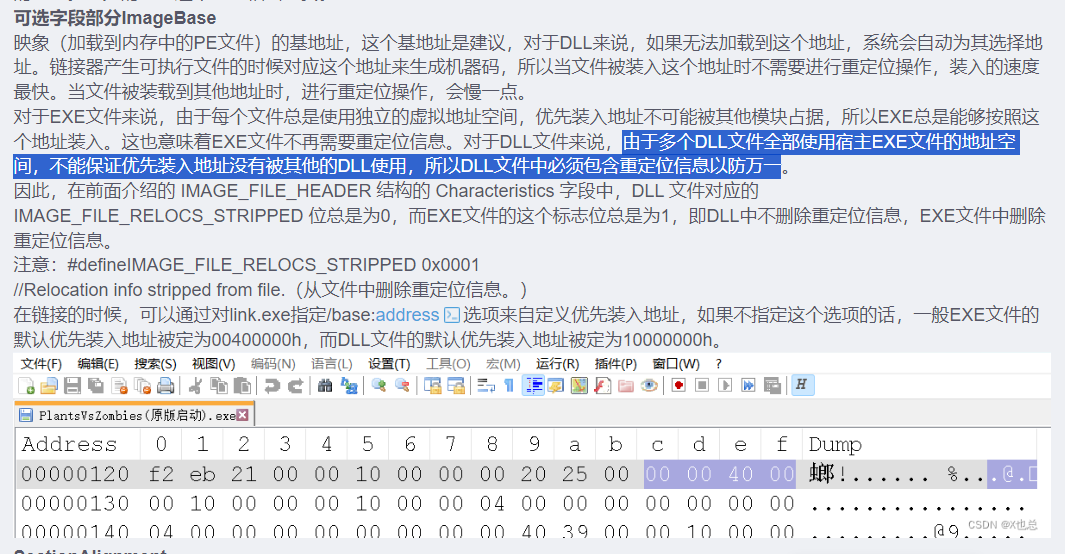


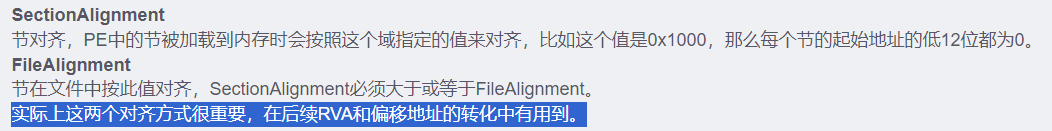


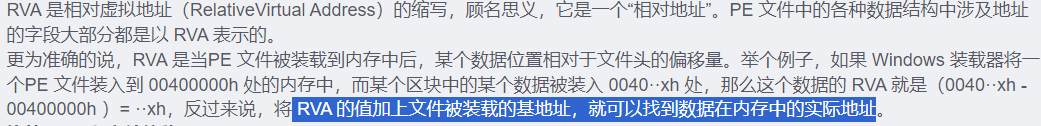


AddressOfEntryPoint

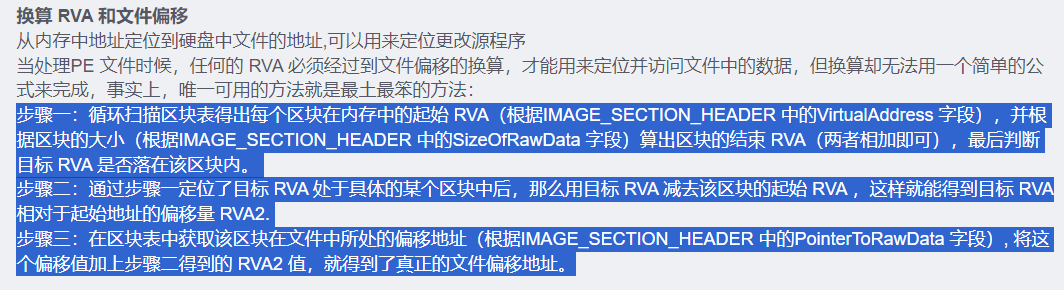
程序入口的RVA，对于exe可以理解为WinMain的RVA。对于DLL可以理解为DllMain的RVA，对于驱动程序，可以理解为DriverEntry的RVA。写壳的时候这个地址相当重要。

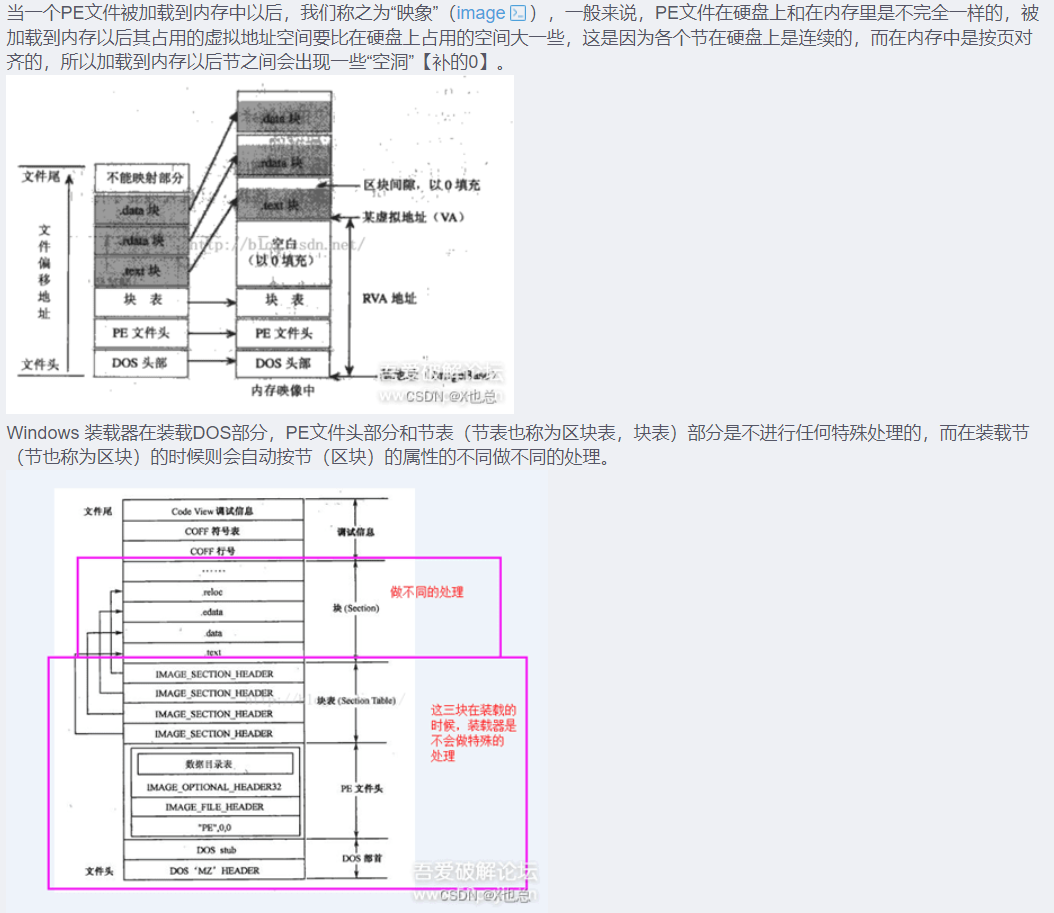


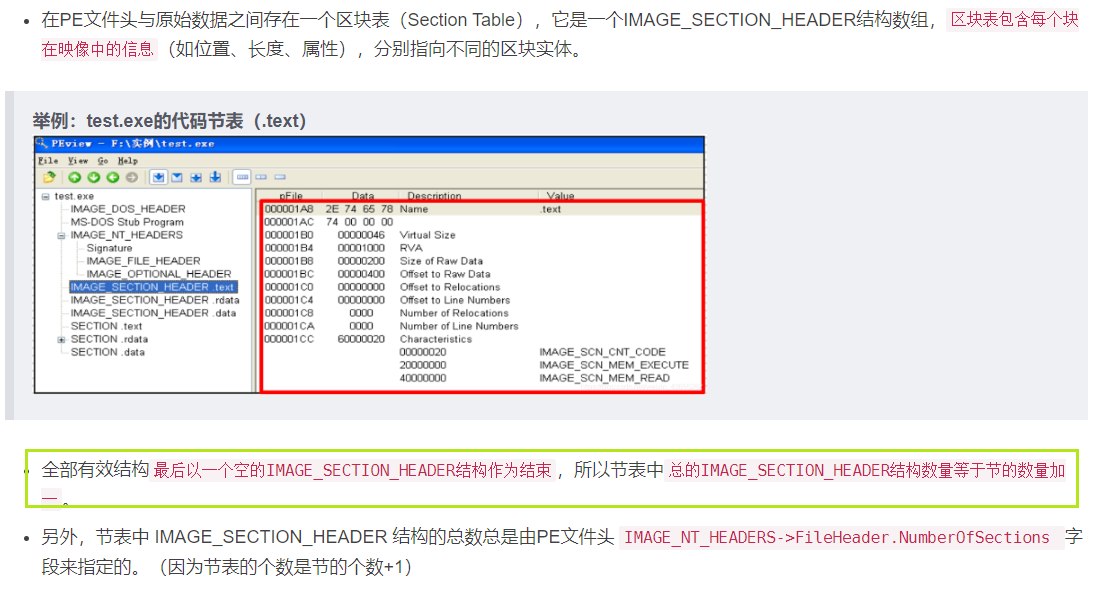




**\*\*\*\***







OriginalFirstThunk和FirstThunk ：

在文件中：它们所指向IMAGE\_THUNK\_DATA结构的数组所有值都是相同的；

在内存中：FirstThunk所指向IMAGE\_THUNK\_DATA结构的数组的值会改变。



为什么需要重定位？

PE文件中部分数据是以VA地址存储的，当PE文件无法加载到预期ImageBase时，这些地址就需要修正。

**Lab1**

**Test03:**

.386

.model flat,stdcall

option casemap:none

.stack 4096//给栈分配4096字节空间

//引入提供 Windows API 调用和通用汇编例程所需的库

includelib user32.lib

includelib kernel32.lib

includelib masm32.lib

//声明 Windows API 函数 ExitProcess 的原型，用于终止当前进程

ExitProcess PROTO,dwExitCode:DWORD

.data

//用六个16进制的dword值初始化数组a

A dd 567Fh, 78FFh, 4C6Ah, 90A4h, 6ABCh, 0F56Ah

//包括6个dword条目,所有值均为0

B dd 6 DUP(0)

.code

main proc

mov ecx, LENGTHOF A//a的长度

lea esi, A//数组 A 的地址加载到 esi（源索引）

lea edi, B//数组 B 的地址加载到 edi（源索引）

；复制（L1）：把数组A中的数据复制到数组B中

L1:

mov ax, [esi]

mov [edi], ax

add esi, TYPE A//将源指针esi移动到下一个元素(增加A的大小,4字节)

add edi, TYPE B//目标指针同理

loop L1//减少ecx的值并检查是否为0.不为0跳回标签L1继续循环

//准备进行算数操作和异或操作(L2)

mov ecx, LENGTHOF A

lea esi, A//地址

lea edi, B

；L2：A算数左移2位，B循环右移2位，A、B按位异或并将异或结果存入A数组

L2:

mov eax, [esi]

sal eax, 2//左移2位(乘以4)

mov ebx, [edi]

ror ebx, 2//右循环移位2位

xor eax, ebx//按位异或

mov [esi], eax//将异或结果存储到A当前位置

add esi, TYPE A//a的下一个元素(4字节)

add edi, TYPE B

loop L2//减少ecx并检查0,不为0跳回l2继续循环

invoke ExitProcess,0;//调用 Windows API 函数 ExitProcess，正常退出程序，返回值为 0。

main endp

End main

**Test04**

.386

.model flat,stdcall//定义程序的内存模型为平面模型（flat model），调用约定为stdcall，这意味着参数从右到左传递，调用方负责清理堆栈。

option casemap:none//禁用大小写映射，使得标识符对大小写敏感

.stack 4096//4096字节的堆栈

includelib user32.lib

includelib kernel32.lib

includelib masm32.lib

ExitProcess PROTO,dwExitCode:DWORD

.data

strA byte "asdterDXRT345\*!@as4",0//定义一个以null结尾的字节数组strA,其内容为字符串"asdterDXRT345\*!@as4"

arrayB dword 4 DUP(0)//dword数组,包括4个初始值为0的元素,用来存储处理后的数字

.code

main proc

lea esi, strA//a地址放esi

lea edi, arrayB//加载b地址到寄存器edi,用于指向存储数字的位置

L1:

mov al,[esi]//将当前esi指向的字符加载到寄存器al

cmp al,'0'//al中的值与0比较

jb next2//小于’0’,跳next2

cmp al,':'//

jb next1//小于’:’,跳next1

cmp al,'A'

jb next2//’A’,跳next2

cmp al,'['

jb ChangeToLower//小于’[‘,跳

cmp al,'{'

jb ChangeToCapital//小于’{‘,跳

next1:

;next1是用来将数字存入新的数组

sub al,'0'//将al中字符0-9转为对应整数0-9

mov [edi],al//将转换后的数字存在edi指向的数组arrayB当前位置

add edi,TYPE arrayB//移动edi到下一个dword位置,准备存储下一个数字

jmp next2//跳next2,处理下一个字符

next2:

;next2是用来处理非数字非字母字符，直接保留在原字符串中

inc esi//指向字符串中下一个字符

jmp L1//跳回L1

ChangeToLower:

;ChangeToLower是用来处理字母大写向小写反转的

or al,00100000b//使用按位 或运算将al中字符转为小写(通过设置第五位,asii码中大写与小写之间差32,大写+32=小写)

mov [esi],al//转换后的字符存回原字符串

jmp next2//跳next2,处理下一个字符

ChangeToCapital:

;ChangeToCapital是用来处理字母小写向大写反转的

and al,11011111b//按位 与运算将al字符转为大写(通过清除第五位)

mov [esi],al//转换后的字符存回原字符串

jmp next2//处理下一个字符

main endp

END main

**Lab2**

1. **加法子程序实现**

**（1）寄存器传参**

.386

.model flat,stdcall

//指定不对标识符进行大小写敏感的映射

option casemap:none

.stack 4096

includelib user32.lib

includelib kernel32.lib

includelib masm32.lib

//定义一个外部过程原型 ExitProcess，接受一个 DWORD 类型的退出代码参数

ExitProcess PROTO,dwExitCode:DWORD

;寄存器传参

.data

//sum用于存储两个数相加的结果，初始值未定 (用 ? 表示未初始化)

sum dd ?

//dd:是一个伪指令，用于分配一个双字（double word）大小的内存空间

a dd 1

b dd 2

.code

AddTwo proc

add eax,ebx

//用于返回调用该过程的位置

ret

AddTwo endp

main proc

mov eax,a

mov ebx,b

call AddTwo

mov sum,eax

//调用 Windows API ExitProcess 以退出程序，返回码为 0，表示正常结束

invoke ExitProcess,0

ret

main endp

end main

1. **变量传参**

**//**使用 ax 和 bx 来直接处理传递的值。直接推送 a 和 b 的值到栈中，AddTwo 过程直接使用这些值进行运算。

.386

.model flat,stdcall

option casemap:none

.stack 4096

includelib user32.lib

includelib kernel32.lib

includelib masm32.lib

ExitProcess PROTO,dwExitCode:DWORD

;变量传参

.data

//dw双字变量

sum dw ?

a dw 1

b dw 2

.code

AddTwo proc

//保存现有的基指针 (base pointer) 并设置新的基指针

push ebp

mov ebp,esp

//从栈中获取第一个参数

mov ax,[ebp+12]应该是12吧

//从栈中获取第二个参数

mov bx,[ebp+8]

//两个参数相加，结果存储在 ax 中

add ax,bx

pop ebp

ret

AddTwo endp

main proc

//将 a 和 b 的值推入栈中

push a

push b

//调用 AddTwo 过程

call AddTwo

//将结果 (从 ax 中) 存储到 sum 中

mov sum,ax

// 恢复 a 和 b 的值

pop a

pop b

ret

main endp

end main

1. **变量地址传参**

**//**使用 eax 和 ebx 来首先获取参数的地址，再通过地址去获取实际的值。推送 a 和 b 的地址到栈中，AddTwo 通过这些地址去获取实际的值进行运算。

.386

.model flat,stdcall

option casemap:none

.stack 4096

includelib user32.lib

includelib kernel32.lib

includelib masm32.lib

ExitProcess PROTO,dwExitCode:DWORD

;变量地址传参

.data

sum dw ?

a dw 1

b dw 2

.code

AddTwo proc

push ebp

mov ebp,esp

//从栈中获取第一个参数的地址

//需要 +12 是因为在 b 的位置之后还有 ret\_addr=+8 (四字节变量)，且后面的 +4 是接下来的 a(再通过地址来获取实际值)

mov eax,[ebp+12]

//将参数 a 的值加载到 ax 中

mov ax,[eax]

//从栈中获取第二个参数的地址（b

mov ebx,[ebp+8]

//将参数 b 的值加载到 bx 中

mov bx,[ebx]

add ax,bx

pop ebp

ret

AddTwo endp

main proc

// 将 a 和 b 的地址推入栈中

push offset a

push offset b

call AddTwo

//将结果 (从 ax 中) 存储到 sum 中

mov sum,ax

ret

main endp

end main

1. **编写子程序实现对数组求和、平均值，在主程序中传递数组地址和数组个数调用子程序，记录堆栈变化情况**

**.386**

**.model flat,stdcall**

**option casemap:none**

**.stack 4096**

**includelib user32.lib**

**includelib kernel32.lib**

**includelib masm32.lib**

**ExitProcess PROTO, dwExitCode:DWORD**

**.data**

**arrayA dd 1h, 2h, 3h, 4h, 5h ; 定义整数数组**

**sum dd ? ; 存储数组的总和**

**ave dd ? ; 存储数组的平均值**

**.code**

**addArray proc**

**push ebp ; 保存基指针**

**mov ebp, esp ; 建立新的基指针**

**mov esi, [ebp + 12] ; 获取数组地址**

**mov ecx, [ebp + 8] ; 获取数组长度**

**xor eax, eax ; 将 eax 清零，用于存储总和**

**L1:**

**add eax, [esi] ; 将当前数组元素加到总和中**

**add esi, 4 ; 指向下一个数组元素**

**loop L1 ; 循环，直到 ecx 为 0**

**mov sum, eax ; 将总和存储到 sum 变量中**

**pop ebp ; 恢复基指针**

**ret ; 返回**

**addArray endp**

**averageArray proc**

**push ebp ; 保存基指针**

**mov ebp, esp ; 建立新的基指针**

**mov eax, [ebp + 12] ; 获取总和**

**mov ebx, [ebp + 8] ; 获取数组长度**

**xor edx, edx ; 清除 edx，准备进行除法**

**div ebx ; 执行除法，结果存入 eax，余数存入 edx**

**mov ave, eax ; 将平均值存储到 ave 变量中**

**pop ebp ; 恢复基指针**

**ret ; 返回**

**averageArray endp**

**main proc**

**push offset arrayA**

**push lengthof arrayA ; 然后推送数组的长度**

**call addArray ; 调用数组求和子程序**

**push sum ; 先推送求和结果的地址**

**push lengthof arrayA ; 再推送数组的长度**

**call averageArray ; 调用平均值计算子程序**

**invoke ExitProcess, 0 ; 结束程序**

**ret**

**main endp**

**end main**

1. **编写子程序将其中字母大小写反转、将其中的数字转换为二进制存储到数组中。主程序定义字符串为“asdterDXRT345\*!@as4”，以及存储数组，通过传递字符串地址和数组地址调用子程序，记录堆栈变化情况**

.386

.model flat,stdcall

option casemap:none

.stack 4096

includelib user32.lib

includelib kernel32.lib

includelib masm32.lib

ExitProcess PROTO,dwExitCode:DWORD

.data

strA byte "asdterDXRT345\*!@as4",0定义一个以 null 结尾的字符串 strA

arrayB dword 4 DUP(0)定义一个 dword 数组 arrayB，初始化为 0，长度为 4

.code

Switch proc

push esi保存 esi 寄存器的值（通常用于指向字符串）

L1:

mov al,[esi]; 读取字符串中的当前字符到 al 中

cmp al,'0'; 比较 al 与字符 '0'

jb next2; 如果小于 '0'，跳转到 next2

cmp al,':'; 比较 al 与字符 ':'

jb next1; 如果小于 ':'，跳转到 next1

cmp al,'A'; 比较 al 与字符 'A'

jb next2; 如果小于 'A'，跳转到 next2

cmp al,'['; 比较 al 与字符 '['

jb ChangeToLower; 如果小于 '['，跳转到 ChangeToLower

next1:

;next1是用来将数字0-9存入新的数组

sub al,'0'; 将字符数字转换为实际数字（'0' -> 0，'1' -> 1等）

mov [edi],al; 将处理后的数字存入 arrayB 的当前位置

add edi,TYPE arrayB; 移动 edi 指针到 arrayB 下一个储存位置

jmp next2; 跳转到 next2

next2:

;next2是用来处理非数字非字母字符，直接保留在原字符串中

inc esi; 移动到下一个字符

jmp L1; 返回到字符处理循环

ChangeToLower:

;ChangeToLower是用来处理字母大写向小写反转的

or al,00100000b; 将大写字母转换为小写（设置第 6 位）

mov [esi],al; 将转换后的字符存回原字符串

jmp next2; 跳转到 next2

ChangeToCapital:

;ChangeToCapital是用来处理字母小写向大写反转的

and al,11011111b; 将小写字母转换为大写（清除第 6 位）

mov [esi],al; 将转换后的字符存回原字符串

jmp next2; 跳转到 next2

pop esi; 恢复 esi 寄存器的值

ret; 返回到调用者

Switch endp

main proc

mov esi, offset strA; 将 strA 的地址加载到 esi

mov edi, offset arrayB; 将 arrayB 的地址加载到 edi

call Switch; 调用 Switch 过程来处理字符串和数组

main endp

end main

**Lab3**

.386

.model flat,stdcall

option casemap:none

.stack 4096include windows.inc

include user32.inc

includelib user32.lib

includelib kernel32.lib

includelib masm32.lib

ExitProcess PROTO, dwExitCode:DWORD

.data

* //定义一个名为 arr 的数组，其类型为 word（2 字节）并包含十个十六进制数值。从 1h 到 0Ah（即 1 到 10）。

arr word 1h, 2h, 3h, 4h, 5h, 6h, 7h, 8h, 9h, 0Ah

* //定义一个字符串 msg，以 null 字符（0）作为结束符，内容为 "Hello! I am Liu Jingting."。

msg db 'Hello! I am Liu Jingting.', 0

.code

COPY proc ;名为 COPY 的过程。用于复制数组内容的功能

push esi ;将数组arr的首地址压入栈中 esi 的当前值压入栈中，用于保存后续可能使用的值

push ecx ecx 的当前值压入栈中，以保存这个寄存器的值

push ebp ebp 的当前值压入栈中，以便后续恢复使用

mov ebp,esp esp 的值（即栈指针）赋给 ebp，建立栈帧，方便后续引用栈中的变量

sub edi,20 ;??????

mov edi,ebp ebp 赋值给 edi。现在 edi 将用于存储复制的数据

L1:

mov ax, [esi] 从 esi 指向的内存地址读取一个 word（2 字节）到寄存器 ax

mov WORD PTR [edi], ax 将 ax 中的值存入 edi 指向的内存位置

add esi, TYPE WORD  esi 自增一个 word 的大小（2 字节），指向下一个数组元素

add edi, TYPE WORD 同理

loop L1 loop 指令会减少 ecx 的值，并且如果 ecx 不为零，控制流会跳回标签 L1，继续循环

pop ebp

pop ecx

pop esi

ret 返回到调用的程序

COPY endp

//调用系统的 MessageBox 函数，显示一个消息框，内容为 msg 定义的字符串。NULL 表示不指定父窗口，MB\_OK 表示消息框中只显示一个 "OK" 按钮

main procinvoke MessageBox,NULL,offset msg, NULL, MB\_OK

mov esi, offset arr 数组 arr 的地址放入寄存器 esi

mov ecx, lengthof arr 数组 arr 的长度（元素个数）加载到寄存器 ecx 中

call COPY 调用之前定义的 COPY 过程，执行数组复制操作

invoke ExitProcess,NULL 调用 ExitProcess 函数终止程序，并返回系统

ret main 过程返回

main endp

end main定义 main 过程的结束，标记程序的结束