



第8章 子程序设计

- 8.1 子程序的概念
- 8.2 子程序的基本用法
 - 子程序的定义、调用和返回
 - 参数传递、现场保护
- 8.3 子程序应用示例
- 8.4 C语言程序中函数运行机理
- 8.5 汇编程序中子程序的高级用法
- 8.6 递归子程序的设计





第8章 子程序设计

学习重点

子程序的定义、调用、返回；
主程序与子程序的参数传递

学习难点

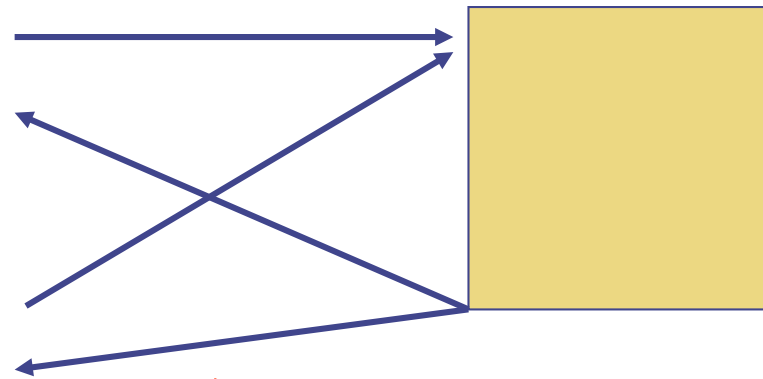
CALL与RET指令的执行过程
参数的传递、子程序对参数的访问
编写和调试子程序



8.1 子程序的概念

K : 调用子程序 A
DK:
.....
J : 调用子程序 A
DJ:
.....

子程序A



思考:

转移的本质是什么?

改变CS:EIP

断点: 转子指令的直接后继指令的地址。

子程序执行完毕, 返回主程序的断点处继续执行

如何改变CS:EIP, 使其按照预定的路线图前进?



8.1 子程序的概念

独立程序段

用控制指令调用它

在执行完后再返回调用它的程序中继续执行。

这样的独立程序段称为子程序。

调用子程序的程序称为主程序（或称调用程序）





8.2 子程序的基本用法

子程序的定义

子程序名 PROC NEAR 或者 FAR

.....

.....

子程序名 ENDP





8.2 子程序的基本用法

子程序的调用与返回

调用指令

直接 **CALL** 过程名

间接 **CALL** DWORD PTR OPD

CPU 要做的工作:

(1) 保存断点

(EIP) \rightarrow \downarrow (ESP)





8.2 子程序的基本用法

子程序的调用与返回

(1) 保存断点

ESP →

断点的EA

(2) 将子程序的地址送 EIP

注意：间接调用是如何取地址的。





8.2 子程序的基本用法

子程序的调用与返回

返回指令： RET

返回指令的执行过程：

返回： $\uparrow (\text{ESP}) \rightarrow \text{EIP}$

特别注意： 栈顶必须是主程序的断点地址，
否则，运行会出现问题





8.2 子程序的基本用法

子程序的调用与返回

返回指令: `RET n`

消除不再使用的入口参数对堆栈空间的占用

STEP 1:

$\uparrow (\text{ESP}) \rightarrow \text{EIP}$

`.model flat, stdcall`

`stdcall` 采用在子程序中恢复堆栈的方法

STEP 2:

$(\text{ESP}) + n \rightarrow \text{ESP}$





8.2 子程序的基本用法

子程序调用现场的保护方法

现场：执行到某一条指令时，各寄存器的值。
存储单元中的内容等等。

现场是否需要保护？为什么？

如何保护和恢复现场？

何时保护和恢复现场？

保护和恢复现场可以在主程序中完成，也可在子程序中完成。一般在子程序中完成。





8.2 子程序的基本用法

主程序与子程序间的参数传递

例：求一串数据的和。

涉及到三个参数：

数据的起始地址、数据的个数，存放结果的地址

- 使用寄存器传递参数
- 约定单元传递参数
- 使用堆栈传递参数





8.2 子程序的基本用法

子程序设计应注意的问题

➤ 子程序摆放的位置

一般在代码段开头或者代码段结束前

```
p1:    call  func1
func1  proc
f1:    .....
        ret
func1  endp
p2:    .....
```

Q: 程序段的执行
流程是什么?





8.2 子程序的基本用法

子程序设计应注意的问题

➤ 子程序中堆栈的使用问题

ret: 从当前的栈顶弹出一个双字送给EIP。

若执行ret前栈顶元素不是调用子程序时保存的断点地址，就不能回到原断点处继续执行。

若在刚进入子程序时，有：

```
push    ebp
```

```
mov     ebp, esp
```

在子程序中保持 (ebp)不变，在ret之前，可以：

```
mov     esp, ebp
```

```
pop     ebp
```





8.3 子程序应用示例

➤ 字符串比较，实现类似于C语言的strcmp函数。

```
.data
lpFmt db '%s > %s ? %d (0, =; -1,<;1,>)',0dh,0ah,0
string1 db 'hello',0
string2 db 'very good',0
string3 db 'hello',0
.....
main proc c
push offset string2
push offset string1
call strcmp
Add esp, 8
invoke printf, offset lpFmt, offset string1, offset string2, eax
push offset string3
Push offset string1
call strcmp
add esp, 8
invoke printf, offset lpFmt, offset string1, offset string3, eax
```

```
strcmp proc
push ebp
mov ebp, esp
push esi
push edi
push edx
mov edi, [ebp+8]
mov esi, [ebp+12]
strcmp_start: mov dl, [edi]
cmp dl, [esi]
ja strcmp_large
jb strcmp_small
cmp dl, 0
je strcmp_eq
inc esi
inc edi
jmp strcmp_start
strcmp_large: mov eax, 1
jmp strcmp_exit
strcmp_small: mov eax, -1
jmp strcmp_exit
strcmp_eq: mov eax, 0
strcmp_exit:
pop edx
pop edi
pop esi
pop ebp
ret
strcmp endp
```





8.3 子程序应用示例

➤ 字符串比较，实现类似于C语言的strcmp函数。

➤ invoke printf的调用反汇编，CALL指令存放的是什么？

```
invoke printf, offset lpFmt, offset string1, offset string2, eax
00508432 50                                push     eax
00508433 68 26 B0 57 00                  push     offset string2 (057B026h)
00508438 68 20 B0 57 00                  push     offset string1 (057B020h)
0050843D 68 00 B0 57 00                  push     offset lpFmt (057B000h)
00508442 E8 9E 97 FF FF                  call     _printf (0501BE5h)
00508447 83 C4 10                        add      esp, 10h
```

同样地call strcmp的调用

```
call strcmp
00508454 E8 22 00 00 00                  call     strcmp (050847Bh)
add esp, 8
00508459 83 C4 08                        add      esp, 8
invoke printf, offset lpFmt, offset string1, offset string3, eax
0050845C 50                                push     eax
```



8.3 子程序应用示例

```
main proc c
    CALL DISPLAY
    msg1 DB 'Very Good', 0DH, 0AH, 0
    CALL DISPLAY
    msg2 db '12345', 0DH, 0AH, 0
    invoke ExitProcess, 0
main endp

DISPLAY PROC
    .....
DISPLAY ENDP

END
```

编写子程序，
使其能够显示**CALL**指令
下面的串





8.3 子程序应用示例

```
.686P
.model flat, stdcall
ExitProcess proto :dword
includelib kernel32.lib
putchar      proto c :byte ;显示给定 ASCII 对应的字符
includelib libcmt.lib
includelib legacy_stdio_definitions.lib
.stack 200
.code
main proc c
    call display
    msg1 db 'Very Good', 0DH, 0AH, 0
    call display
    msg2 db '12345', 0DH, 0AH, 0
    invoke ExitProcess, 0
main endp
```





8.3 子程序应用示例

```
display proc
    pop    ebx
p1:
    cmp    byte ptr [ebx], 0
    je     exit
    invoke putchar, byte ptr [ebx]
    inc    ebx
    jmp    p1
exit:
    inc    ebx
    push   ebx
    ret
display endp
end
```

Q1: 若将 ExitProcess
移到子程序之后，结果
如何？

Q2: 若少写RET之前的
INC EBX，结果如何？





8.3 子程序应用示例

```
008A2040 E8 20 00 00 00 call    display (08A2065h)
008A2045 56          push    esi
008A2046 65 72 79      jb     _display@0+5Dh (08A20C2h)
008A2049 20 47 6F      and     byte ptr [edi+6Fh],al
008A204C 6F          outs    dx,dword ptr [esi]
008A204D 64 0D 0A 00 E8 0F or      eax,0FE8000Ah
```

.....

--- display_string.asm -----

```
008A2065 5B          pop     ebx
p1: cmp byte ptr [ebx],0
008A2066 80 3B 00      cmp     byte ptr [ebx],0
      je  exit
```

.....

第一个CALL指令的机器码中，存储的：00 00 00 20，是什么含义？

008A2065 - 008A2045 = 00 00 00 20

Q: 56 65 72 79 20 47 6F 6F 64 0D 0A 00 ，是什么含义？





华中科技大学

8.4 C语言程序中函数的运行机理





8.4 C语言程序中函数的运行机理

- 如何传递参数？
- 传递什么？
 - 按值传递、按地址传递、按引用传递
 - 不同类型的形参/实参， 传递的内容有何差别？
- 传到什么地方去了？
- 如何进入函数？
- 如何从函数返回？
- 如何传递函数返回值？
- 函数中变量空间如何分配？
- 如何理解递归函数调用？





8.4 C语言程序中函数的运行机理

```
#include <stdio.h>
int fadd(int x, int y)
{
    int u, v, w;
    u=x+10;
    v=y+25;
    w=u+v;
    return w;
}
```

```
int main( )
{
    int a=100;    // 0x 64
    int b=200;    // 0x C8
    int sum=0;
    sum=fadd(a, b);
    printf("%d\n", sum);
    return 0;
}
```





8.4 C语言程序中函数的运行机理

变量空间分配

```
13:      int    a=100;      // 0x 64
00401088      mov     dword ptr [ebp-4], 64h
14:      int    b=200;      // 0x C8
0040108F      mov     dword ptr [ebp-8], 0C8h
15:      int    sum=0;
00401096      mov     dword ptr [ebp-0Ch], 0
16:      sum=fadd(a, b);
```

in(int, char **)		Name	Value
	Value	&a, x	0x0012ff7c
	100	&b, x	0x0012ff78
	200	&sum, x	0x0012ff74
	0	ebp, x	0x0012ff80

0012FF74 00 00 00 00 C8 00 00 00 64 00 00 00



8.4 C语言程序中函数的运行机理

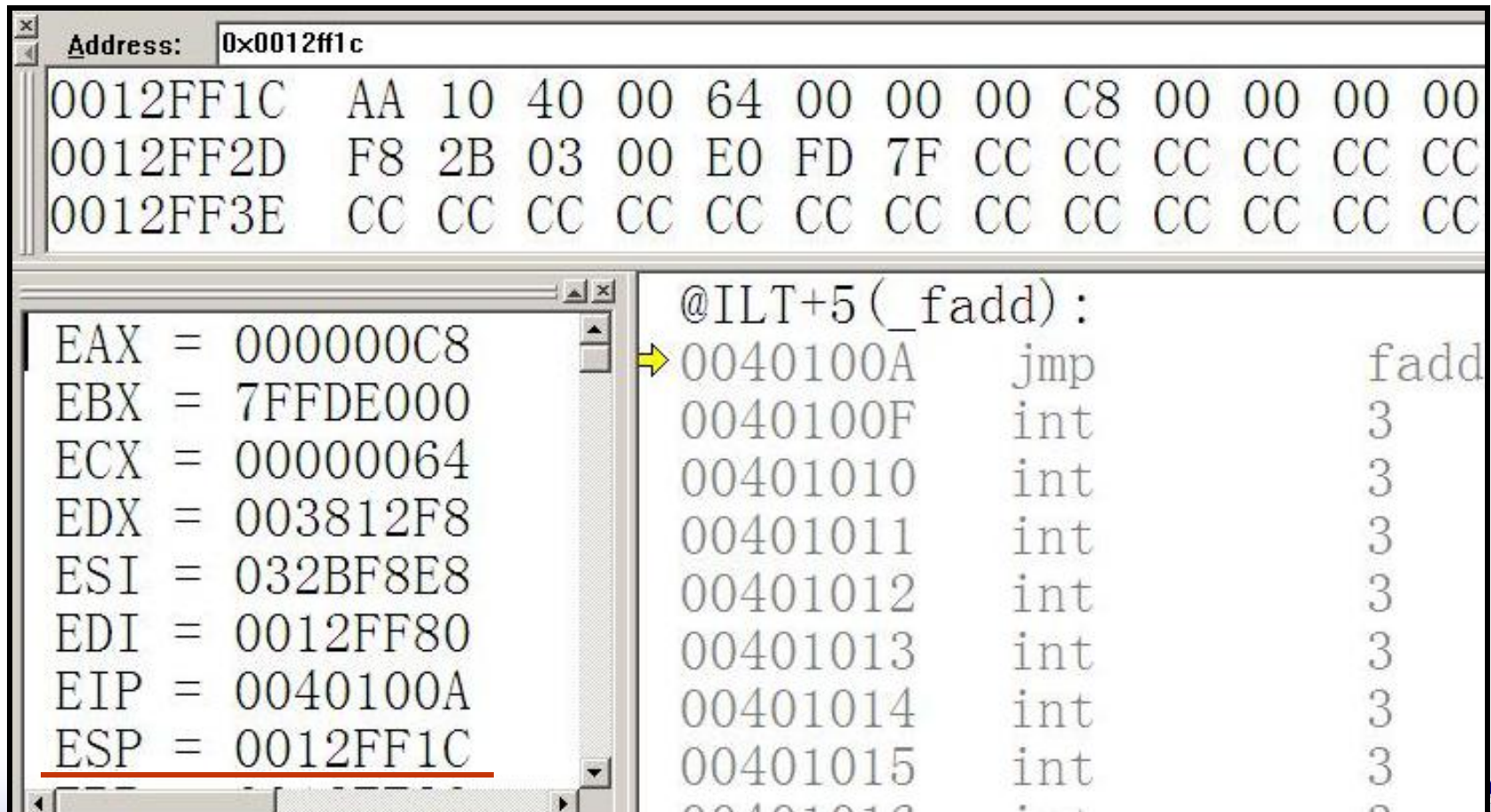
函数调用

```
13:      int  a=100;      // 0x 64
00401088      mov          dword ptr [ebp-4], 64h
14:      int  b=200;      // 0x C8
0040108F      mov          dword ptr [ebp-8], 0C8h
15:      int  sum=0;
00401096      mov          dword ptr [ebp-0Ch], 0
16:      sum=fadd(a, b);
● 0040109D      mov          eax, dword ptr [ebp-8]
➔ 004010A0      push         eax
004010A1      mov          ecx, dword ptr [ebp-4]
004010A4      push         ecx
004010A5      call          @ILT+5(_fadd) (0040100a)
004010AA      add          esp, 8
004010AD      mov          dword ptr [ebp-0Ch], eax
```


8.4 C语言程序中函数的运行机理

Sum = fadd(a, b)

执行CALL指令后的状态



The screenshot shows a debugger window with the following content:

Address: 0x0012ff1c

Address	Hex	Hex	Hex	Hex	Hex	Hex	Hex	Hex	Hex	Hex	Hex	Hex	Hex	Hex
0012FF1C	AA	10	40	00	64	00	00	00	C8	00	00	00	00	00
0012FF2D	F8	2B	03	00	E0	FD	7F	CC	CC	CC	CC	CC	CC	CC
0012FF3E	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC

Registers:

- EAX = 000000C8
- EBX = 7FFDE000
- ECX = 00000064
- EDX = 003812F8
- ESI = 032BF8E8
- EDI = 0012FF80
- EIP = 0040100A
- ESP = 0012FF1C

Disassembly:

@ILT+5(_fadd):

- 0040100A jmp fadd
- 0040100F int 3
- 00401010 int 3
- 00401011 int 3
- 00401012 int 3
- 00401013 int 3
- 00401014 int 3
- 00401015 int 3

8.4 C语言程序中函数的运行机理

Sum=fadd(a,b)

执行CALL指令后的状态

```
004010A5  call    @ILT+5(_fadd) (0040100a)
004010AA  add     esp,8
```

0012FF1C

ESP →

断点地址

EIP 为函数的入口地址

(EIP) = 0040100A

a的值(即100)压栈

b的值(即200)压栈

0012FF1C AA 10 40 00 64 00 00 00 C8 00 00 00



8.4 C语言程序中函数的运行机理

3: *int fadd(int x, int y)*

4: {

00401020 *push ebp*

00401021 *mov ebp,esp*

00401023 *sub esp,4Ch*

00401026 *push ebx*

00401027 *push esi*

00401028 *push edi*

00401029 *lea edi,[ebp-4Ch]*

0040102C *mov ecx,13h*

00401031 *mov eax,0CCCCCCCCh*

00401036 *rep stos dword ptr [edi]*

5: *int u,v,w;*

6: *u=x+10;*

00401038 *mov eax,dword ptr [ebp+8]*

0040103B *add eax,0Ah*

0040103E *mov dword ptr [ebp-4],eax*



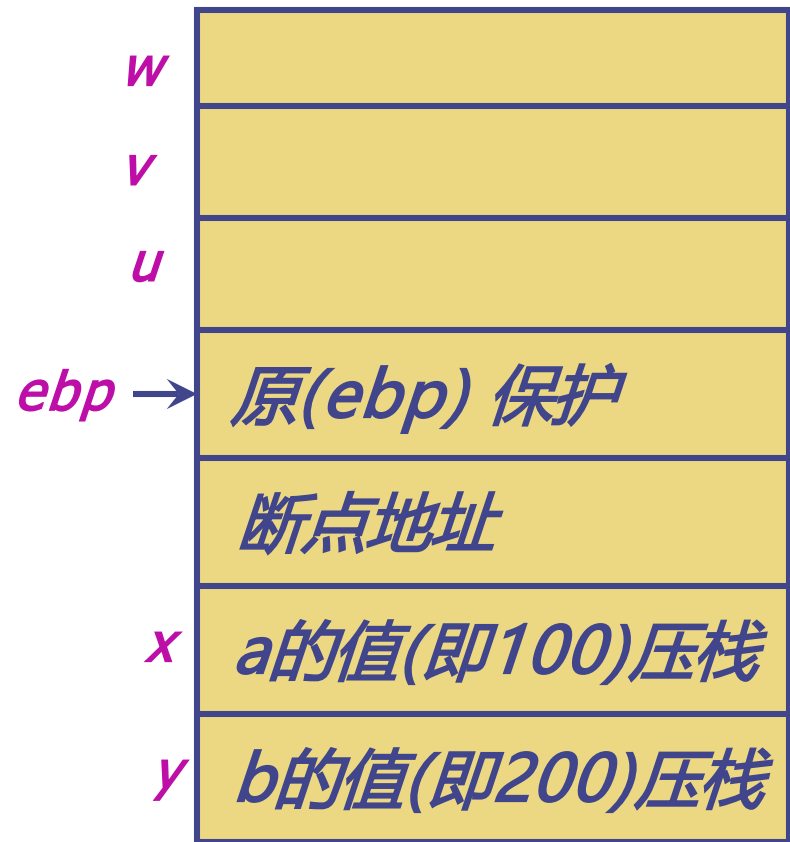
观察形参*x*的位置



8.4 C语言程序中函数的运行机理

```

5:      int u,v,w;
6:      u=x+10;
mov     eax,dword ptr [ebp+8]
add     eax,0Ah
mov     dword ptr [ebp-4],eax
7:      v=y+25;
mov     ecx,dword ptr [ebp+0Ch]
add     ecx,19h
mov     dword ptr [ebp-8],ecx
8:      w=u+v;
mov     edx,dword ptr [ebp-4]
add     edx,dword ptr [ebp-8]
mov     dword ptr [ebp-0Ch],edx
9:      return w;
mov     eax,dword ptr [ebp-0Ch]
    
```



观察局部变量的位置

8.4 C语言程序中函数的运行机理

函数调用——返回

```
9:      return w;  
mov     eax,dword ptr [ebp-0Ch]  
10: }  
pop     edi  
pop     esi  
pop     ebx  
mov     esp, ebp  
pop     ebp  
ret
```



思考：局部变量的作用域？
局部空间的释放？
函数如何的返回？
改变形参的值，对实参有影响吗？

8.4 C语言程序中函数的运行机理

```
15:    int sum=0;
      mov     dword ptr [ebp-0Ch],0
16:    sum=fadd(a,b);
      mov     eax,dword ptr [ebp-8]
      push    eax
      mov     ecx,dword ptr [ebp-4]
      push    ecx
      call    @ILT+5(_fadd)
      add     esp,8
      mov     dword ptr [ebp-0Ch],eax
```





8.4 C语言程序中函数的运行机理

```
17:      printf("%d\n", sum);
004010B0      mov          edx, dword ptr [ebp-0Ch]
004010B3      push         edx
004010B4      push         offset string "%d\n" (0042201c)
004010B9      call          printf (004010f0)
004010BE      add          esp, 8
18:      return 0;
004010C1      xor          eax, eax
19:      }
004010C3      pop          edi
004010C4      pop          esi
004010C5      pop          ebx
004010C6      add          esp, 4Ch
004010C9      cmp          ebp, esp
004010CB      call          __chkesp (00401170)
004010D0      mov          esp, ebp
004010D2      pop          ebp
004010D3      ret
```





8.4 C语言程序中函数的运行机理

函数编译——代码优化

Debug版本调试中:

在局部变量之上, 留了 40H个字节的空間?

局部变量的初始化值是多少?

保护了未用的一些的寄存器?

Release 版本





8.4 C语言程序中函数的运行机理

Release 版本

函数编译——
代码优化

```
:00401010  push 000000C8  
:00401015  push 00000064  
:00401017  call 00401000  
:0040101C  push eax
```

; Possible StringData Ref from Data Obj -> "%d"

```
:0040101D  push 00407030  
:00401022  call 00401030  
:00401027  add esp, 00000010  
:0040102A  xor eax, eax  
:0040102C  ret
```

```
:00401000  mov eax, dword ptr [esp+08]  
:00401004  mov ecx, dword ptr [esp+04]  
:00401008  lea eax, dword ptr [ecx+eax+23]  
:0040100C  ret
```





8.4 C语言程序中函数的运行机理

1. 函数的参数和函数的局部变量一样，它们的空间分配都是在堆栈中
2. 以刚进入函数时的堆栈为参考，函数参数所在单元的地址是高于堆栈栈顶地址（栈顶是执行call指令时压入的断点地址），函数的局部变量的空间地址是低于当前堆栈栈顶；
3. 函数的参数和变量的地址都是 $[EBP+N]$ 的形式，是变址寻址，参数： N 为正数，局部变量： N 为负数
4. EBP 在函数中保持不变，进入函数时要注意保存 EBP ，然后将 $ESP \rightarrow EBP$
5. 调用函数时参数压栈，参数地址与调用函数时的实参地址无关；
6. 返回值在 eax 中，不要返回局部变量和参数的地址
7. 破坏堆栈中存放的断点地址，会导致程序出现异常





8.4 C语言程序中函数的运行机理

精雕细琢——程序优化

strcpy的函数实现，看汇编代码

- 一次传送一个字节吗？
- 物理上，实现一个双字（位于不同位置）的传送的速度相同吗？

例如，从（1000H），（1001H）分别取出一个双字送EAX。

- 如何快速判断一个双字中某个字节为 0 ？
- 用C语言，写strcpy的实现函数，可以采用哪些技巧？





8.5 汇编语言中子程序的高级用法

8.5.1 局部变量的定义和使用

在proc语句之后，用local伪指令说明仅在本函数内使用的局部变量。

格式：local 变量名[[数量]][[:类型]]
{ , 变量名[[数量]][[:类型]] }

local u:dword, v:dword, w:dword





8.5 汇编语言中子程序的高级用法

8.5.1 局部变量的定义和使用

```
func proc
    local flag:dword
    push    ebx
    mov     ebx, [ebp+8]
    mov     flag, ebx
    pop     ebx
    ret
func endp
```

```
PUSH EBP
MOV  EBP, ESP
ADD  ESP, 0FFFFFFFCH
PUSH EBX
MOV  EBX, [EBP+8]
MOV  [EBP-4], EBX
POP  EBX
LEAVE
RET
```

leave等效于
mov esp, ebp
pop ebp





8.5 汇编语言中子程序的高级用法

局部变量 VS 全局变量

- 只能用 `lea` 指令来获取局部变量的地址；
可以用 `offset` 获取全局变量的地址；
- 单个局部变量对应变址寻址方式，
 $x \rightarrow [ebp - n]$ ；其中 n 为正数；
单个全局变量对应的寻址是直接寻址；
- 对于局部变量 x , $x[IR * F]$ 对应机器指令表现是：
 $[IR * F + ebp + n]$
不能使用 $x[BR + IR * F]$ 的形式访问；





8.5 汇编语言中子程序的高级用法

8.5.2 子程序的原型说明

函数名 **proto** [函数类型] [语言类型]

[[参数名]:参数类型], [[参数名]:参数类型]...

功能：本模块要调用的过程或函数

➤ 函数类型：默认值是 NEAR。

在32位段扁平内存管理模式下，存储模型说明为
“.model flat”，应该选择NEAR。

➤ 语言类型：C ， stdcall

默认值与 .model flat之后的语言类型一致。





8.5 汇编语言中子程序的高级用法

8.5.2 子程序的完整定义

函数名 **proc** [函数类型][语言类型][uses 寄存器表]
[, 参数名[:参数类型]]...

- 函数类型：默认值是 NEAR。
- 语言类型：C ， stdcall
- 参数类型：dword （32位段）。

函数类型、语言类型、参数类型 与函数原型说明应一致。





8.5 汇编语言中子程序的高级用法

8.5.2 子程序的调用

invoke 函数名 [, 参数1] [, 参数2]...

invoke 是伪指令，编译生成的机器指令序列

.....

PUSH 参数2

PUSH 参数1

CALL 函数名

ADD ESP, 参数占的字节数 (C语言类型)

注：在参数为一个局部变量的地址时，可以用 **ADDR** 获得变量的地址，**全局变量前用 OFFSET**





8.5 汇编语言中子程序的高级用法

8.5.3 子程序的高级用法举例

```
myfadd  proc    x:dword, y:dword
        local  u:dword, v:dword, w:dword

        mov    eax, x
        add    eax, 10
        mov    u,    eax    ; u=x+10;
        mov    eax, y
        add    eax, 25
        mov    v,    eax    ; v=y+25;
        add    eax, u
        mov    w,    eax    ; w=u+v;
        ret
myfadd  endp
```

注意:

当子程序有局部变量或参数时，编译生成的代码，在函数开头会自动加上：

PUSH EBP

MOV EBP, ESP

ret指令自动生成的代码：

leave

ret

leave等效于

mov esp, ebp

pop ebp





8.5 汇编语言中子程序的高级用法

8.5.3 子程序的高级用法举例

```
main proc
    local  a:dword, b:dword
    local  sum:dword
    mov    a, 100
    mov    b, 200
    invoke myfadd, a, b
    mov    sum, eax
    invoke printf, offset lpFmt, sum
    invoke ExitProcess, 0
    ret
main endp
```





子程序填空

F2T10:

将AX中的16位
无符号二进制数
转换为十进制的
数字串，并将其
ASCII码送入
(ESI) 所指定的
缓冲区中。

例：7B -> 123

-> 31H 32H 33H

算法？

```
F2T10      PROC
            MOV     CX, 0
            MOV     BX, 10
DIV_LOOP:  MOV     _____, 0
            DIV     BX
            PUSH    DX

            _____
            CMP     AX, 0

            _____
CHG_LOOP:  POP     AX
            ADD     AL, 30H
            MOV     [ESI], AL

            _____
            DEC     CX
            JNZ     CHG_LOOP
            RET
```





第8章 子程序设计

子程序设计

子程序的定义、调用、返回

参数传递、现场保护

局部变量的定义与访问

作业：8.8

