第二章 基础知识

1 汇编——寻址方式

两种寻址方式：顺序寻址方式，跳跃寻址方式

1.1 操作数寻址

```
MOV 目的操作数，源操作数
```

将源地址传到目标地址

操作数寻址分类：

1.**立即寻址**：指令的地址字段给出的不是操作数的地址，而是操作数本身，这种寻址方式称为立即寻址

```
MOV CL，05H
```

表示将05H传到CL寄存器中

2.**直接寻址**：在指令中直接给出操作数的有效地址

```
MOV AL,[3100H]
```

表示将地址为[3100H]中的数据存储到AL中

默认的存储在数据段寄存器DS中，如果在前面标明了寄存器，那么就存到对应的寄存器中去

```
MOV AL，ES:[3100H]
```

这个代码的意思就是将ES寄存器中地址为[3100H]的数据存储到AL寄存器中

3.**间接寻址**：指令地址字段中的形式地址不是操作数的真正地址，而是操作数地址的指示器，或者说此形式地址单元的内容才是操作数的有效地址

```
MOV [BX]，12H
```

这个代码表示，将12H这个数存储到DS:BX寄存器中

4.**相对寻址**：操作数的有效地址是一个基址寄存器（BX, BP）或变址寄存器（SI, DI）的值加上指令中给定的偏移量之和

```
MOV AX，[DI + 1234H]
```

相对寻址就是在间接寻址的基础上增加了偏移量

5.**基址变址寻址**：将基址寄存器的内容，加上变址寄存器的内容而形成操作数的有效地址

```
MOV EAX，[EBX+ESI]
```

或者也可以写成`MOV EAX，[BX][SI]`或`MOV EAX，[SI][BX]`

6.**相对基址变址寻址**：在基址变址寻址上加上偏移量即可

```
MOV EAX，[EBX+ESI+1000H]
```

也可以写成`MOV EAX，1000H [BX][SI]`

2　汇编——主要指令

指令一般有**两个操作符、一个操作符、三个操作符**

数据传送指令集：

- MOV：把源操作数送给目的操作数，其语法为: MOV 目的操作数,源操作数
- XCHG: 交换两个操作数的数据
- PUSH,POP: 把操作数压入或取出堆栈
- PUSHF,POPF,PUSHA,POPA: 堆栈指令群
- LEA,LDS,LES: 取地址至寄存器
  - LEA：将有效地址传送到指定的寄存器
  - `lea eax, dword ptr [4*ecx+ebx]`，源数是地址`[4*ecx+ebx]`里的数值，`dword ptr`是说，地址中的数值是一个dword型的数据

位运算指令集：

- AND,OR,XOR,NOT,TEST: 执行BIT与BIT之间的逻辑运算
- SHR,SHL,SAR,SAL: 移位指令
- ROR,ROL,RCR,RCL: 循环移位指令

算数运算指令：

- ADD,ADC：加法指令
- SUB,SBB：减法指令
- INC,DEC：把OP的值加一或者减一
- NEG：将OP的符号反相，取二进制补码
- MUL,IMUL：乘法指令
- DIV,IDIV：除法指令

程序流程指令集：

- CMP:比较两个操作数的大小关系
- JMP:跳转到指定的地址
- LOOP:循环指令
- CALL,RET:子程序调用，返回指令（RET指令的功能是从一个代码区域中退出到调用CALL的指令处）
- INT,IRET:中断调用，返回指令
- REP,REPE,REPNE:重复前缀指令集

条件转移命令：

JXX：当特定条件成立，就跳转到指定地址执行

- Z:为0则转移
- G:大于则转移
- L:小于则转移
- E:等于则转移
- N:取相反条件

字符串操作指令集：

- 字符串传送指令：MOVSB,MOVSW,MOVSD
- 字符串比较指令：CMPSB,CMPSW,CMPSD
- 字符串搜索指令：SCASB,SCASW
- 字符串载入或存贮指令：LODSB,LODSW,STOSB,STOSW

3 汇编——函数调用示例

一个简单的函数

```
#include <iostream>
int add(int x,int y)
{
    int z=0;
    z=x+y;
    return z;
}
void main()
{
    int n=0;
    n=add(1,3);
    printf("%d\n",n);
}
```

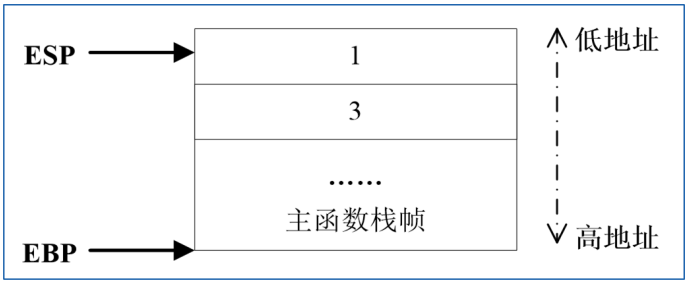
查看汇编代码

```
Void main()
{ ******************
            int n=0;
004114OE   mov         dword ptr [n],0
            n=add(1,3);
00411415   push         3
00411417   push         1
00411419   call            add(411096h)
0041141E   add           esp,8
00411421   mov          dwod ptr [n],eax
      printf(“%d\n,n”);
******************
}
```

我们对其进行分析

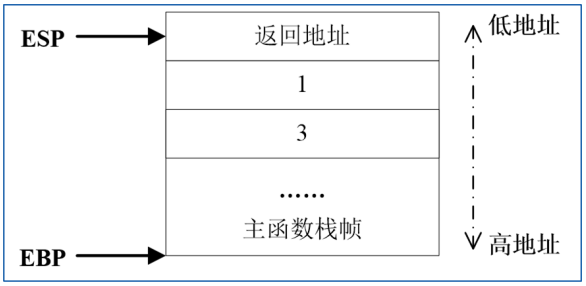
int n=0这一句没啥好说的，就是赋值，给n赋值0

下面执行函数块add，我们在执行call之前，还需要进行参数的入栈，将3和1都push到栈里面，我们得到栈区状态为：



我们在调用函数时需要使用`00411419 call add(411096h)`

主要功能是，向栈中压入当前指令在内存中的位置，保存返回地址；跳转到调用函数的入口地址，就是函数的入口处，此时栈区的状态：



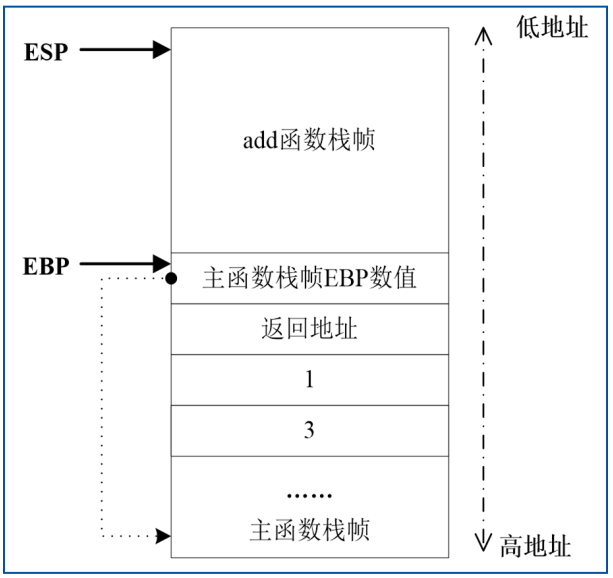
下面分析add函数的汇编代码

```
Int  add (int  x,int  y)
{
004113A0    push          ebp
004113A1    mov           ebp,esp
004113A3    sub            esp, 0CCh
004113A9    push          ebx
004113AA    push          esi
004113AB    push          edi
004113AC    lea             edi,[ebp-0CCh]
004113B2    mov           ecx,33h
004113B7    mov           eax,0CCCCCCCCh
004113BC    rep stos    dword ptr  es:[edi]
         int z=0;
004113BE    mov           dword ptr [z],0
         z=x+y;
004113C5    mov           eax,dword ptr [x]
004113C8    add            eax,dword ptr [y]
004113CB    mov          dword ptr [z],eax
        return z;
004113CE    mov          eax,dword ptr[z]
}
```

首先是**栈帧的切换**

```
004113A0    push          ebp
004113A1    mov           ebp,esp
004113A3    sub           esp, 0CCh
```

这三句，首先将EBP的值入栈，将ESP赋值给EBP，然后再将ESP的值抬高0CCh，完成了栈帧的切换，保存了主函数栈帧的EBP的值，也通过改变两个寄存器的位置为add函数分配了栈帧空间



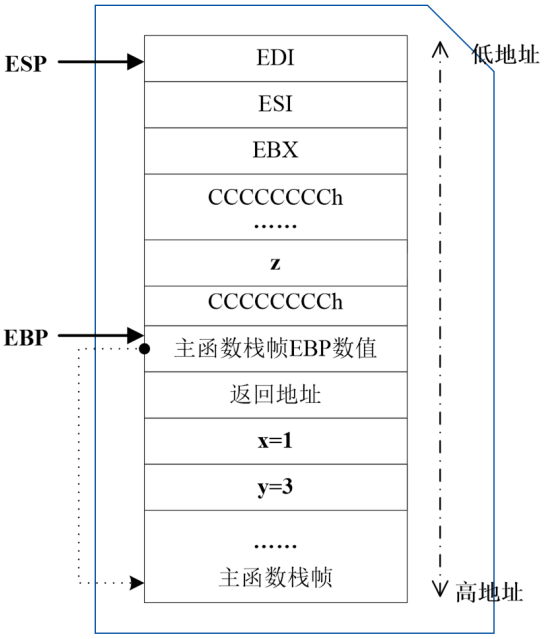
然后是**函数状态的保存**

```
004113A9    push        ebx
004113AA    push        esi
004113AB    push        edi
004113AC    lea         edi,[ebp-0CCh]
```

这几句代码，用于保存`ebx, esi, edi`寄存器的位置，将ebp寄存器抬升0CCh来装入EDI

然后是栈帧的切换，后面的就是将33个4字节的位置都初始化为`0CCCCCCCCh`

然后我们就可以执行函数体了，完成1+3的加法，并将结果存储在eax寄存器中



执行完加法后，我们需要恢复栈帧的状态到main函数，后面几句实现了恢复的过程：首先恢复`edi, esi, ebx`寄存器的值，然后`mov esp,ebp`这一句是恢复了esp寄存器的值，后面一句恢复了ebp寄存器的值，最后ret表示根据返回地址来恢复EIP寄存器的值，相当于`pop EIP`