# 指令系统

里昂学长

考纲

(一) 指令系统的基本概念
(二) 指令格式
(三) 寻址方式
(四) 数据的对齐和大/小端存放方式
(五) CISC和 RISC的基本概念
(六) 高级语言程序与机器代码之间的对应
1.编译器、汇编器和链接器的基本概念
2.选择结构语句的机器级表示
3.循环结构语句的机器级表示
4.**过程（函数）调用**对应的机器级表示

过去考过8道，考察频率还是挺高的，23刚考过

## 指令的基本格式

| 操作码字段 | 地址码字段 |
|---|---|

操作码指出指令中该指令应该执行什么性质的操作以及具有何种功能

地址码给出被操作的信息（指令或数据）的地址

**零地址指令**

| OP |
|---|

1）不需要操作数的指令，如空操作指令、停机指令、关中断指令等。
2）零地址的运算类指令仅用在堆栈计算机中。**通常参与运算的两个操作数隐含地从栈顶和次栈顶弹出，送到运算器进行运算，运算结果再隐含地压入堆栈。**

王道书本P147，有兴趣可以看看，一般具体哪些指令是一地址，哪些是二地址，这种考察的概率不大

**一地址指令**

| OP | A1 |
|---|---|

**二地址指令**

| OP | A1 | A2 |
|---|---|---|

**三地址指令**

| OP | A1 | A2 | A3(结果) |
|---|---|---|---|

**四地址指令**

| OP | A1 | A2 | A3(结果) | A3(下址) |
|---|---|---|---|---|

## 定长操作码

操作码长度固定，n位操作码的指令系统最大能表示2^n次方条指令。
**有利于简化硬件涉及，提高指令译码和识别速度**，当计算机字长为32位甚至更长时，这是常规用法

## 扩展操作码

为了在指令字长有限的前提下仍保持比较丰富的指令种类，可采取可变长度操作码，
**即全部指令的操作码字段的位数不固定**，且分散地放在指令字的不同位置上。
显然，**这将增加指令译码和分析的难度，使控制器的设计复杂化。**

| OP | A1 | A2 | A3 |
|---|---|---|---|

本质就是前缀编码

4位操作码

| 0000 | A1 | A2 | A3 |
|---|---|---|---|
| 0001 | A1 | A2 | A3 |
| ... | ... | ... | ... |
| 1110 | A1 | A2 | A3 |

15条三地址指令

8位操作码

| 1111 | 0000 | A2 | A3 |
|---|---|---|---|
| 1111 | 0001 | A2 | A3 |
| ... | ... | ... | ... |
| 1111 | 1110 | A2 | A3 |

15条二地址指令

12位操作码

| 1111 | 1111 | 0000 | A3 |
|---|---|---|---|
| 1111 | 1111 | 0001 | A3 |
| ... | ... | ... | ... |
| 1111 | 1111 | 1110 | A3 |

15条一地址指令

16位操作码

| 1111 | 1111 | 1111 | 0000 |
|---|---|---|---|
| 1111 | 1111 | 1111 | 0001 |
| ... | ... | ... | ... |
| 1111 | 1111 | 1111 | 1111 |

16条零地址指令

## 常见的运算数指令

| 功能 | 英文 | 汇编指令 | 注释 |
| --- | --- | --- | --- |
| 加 | add | add d,s | #计算d+s，结果存入d |
| 减 | subtract | sub d,s | #计算d-s，结果存入d |
| 乘 | multiply | mul d,s<br>imul d,s | #无符号数d*s，乘积存入d<br>#有符号数d*s，乘积存入d |
| 除 | divide | div s<br>idiv s | #无符号数除法edx:eax/s，商存入eax，余数存入edx<br>#有符号数除法edx:eax/s，商存入eax，余数存入edx |
| 取负数 | negative | neg d | #将d取负数，结果存入d |
| 自增++ | increase | inc d | #将d++，结果存入d |
| 自减-- | decrease | dec d | #将d--，结果存入d |

## 常见的逻辑运算指令

| 功能 | 英文 | 汇编指令 | 注释 |
| --- | --- | --- | --- |
| 加 | and | and d,s | #将d、s 逐位相与,结果放回d |
| 或 | or | or d,s | #将d、s逐位相或，结果放回d |
| 非 | not | not d | #将d逐位取反，结果放回d |
| 异或 | exclusive | xor d,s | #将d、s逐位异或，结果放回d |
| 左移 | shift left | shl d,s | #将d逻辑左移s位，结果放回d(通常s是常量) |
| 右移 | shift right | shr d,s | #将d逻辑右移s位，结果放回d(通常s是常量) |

## 其他指令

用于实现分支结构、循环结构的指令: cmp、 test、 jmp、 jxxx
用于实现函数调用的指令: push、pop、 call、ret
用于实现数据转移的指令:mov
陷入指令，分为无条件陷入和有条件陷入
例如x86下

Test命令将两个操作数进行逻辑与运算，
并根据运算结果设置相关的标志位。
但是，Test命令的两个操作数不会被改变。
TEST AL，80H；测试AL中最高位
JNZ NEXT；如果最高位为1，转到标志NEXT处。

```
int $0x80 #引发一次系统调用  0x80 = 128（d）
```

即为Linux中断向量表中专门用于陷入指令的中断号，调用异常机制

## 寻址方式

| 寻址方式 | 有效地址 | 访存次数 |
|---|---|---|
| 隐含寻址 | 程序指定 | 0 |
| 立即寻址 | A即是操作数 | 0 |
| 直接寻址 | EA=A | 1 |
| 一次间接寻址 | EA=(A) | 2 |
| 寄存器寻址 | EA=Ri | 0 |
| 寄存器间接一次寻址 | EA=(Ri) | 1 |
| 偏移寻址 转移指令 相对寻址 | EA=(PC)+A | 1 |
| 多道程序 基址寻址 | EA=(BR)+A | 1 |
| 循环程序 变址寻址 数组问题 | EA=(IX)+A | 1 |
| 堆栈寻址 | 入栈/出栈时EA的确定方式不同 | 硬堆栈不访存，软堆栈访存1次 |

## 选择结构语句的机器级表示

| 条件转移指令——jxxx | |
|---|---|
| cmp a,b | #比较a和b两个数 |
| je <地址> | #jump when equal, 若a==b则跳转 |
| jne <地址> | #jump when not equal, 若a ! =b则跳转 |
| jg <地址> | #jump when graeter than, 若a>b则跳转 |
| jge <地址> | #jump when greater than or equal to, 若a>=b 则跳转 |
| jl <地址> | #jump when less than, 若a<b 则跳转 |
| jle <地址> | #jump when less than or equal to, 若a<=b则跳转 |

**示例：选择语句的机器级表示**

```
if (a>b){
  c=a;
} else {
  c=b;
  }
```

```
mov eax, 7          #假设变量a=7，存入eax
mov ebx, 6          #假设变量b=6，存入ebx
cmp eax, ebx        #比较变量a和b
jg NEXT             #若a>b，转移到NEXT :
mov ecx, ebx        #假设用ecx存储变量c，令c=b
jmp END             #无条件转移到END :
NEXT :
mov ecx, eax        #假设用ecx存储变量c，令c=a
END :
```

| 示例：选择语句的机器级表示 | |
|---|---|
| if (a≤b){<br>c=a;<br>} else {<br>c=b;<br>} | mov eax, 7 #假设变量a=7，存入eax<br>mov ebx, 6 #假设变量b=6，存入ebx<br>cmp eax, ebx #比较变量a和b<br>jle NEXT #若a<=b，转移到NEXT :<br>mov ecx, eax #假设用ecx存储变量c，令c=a<br>jmp END #无条件转移到END :<br>NEXT :<br>mov ecx, ebx #假设用ecx存储变量c，令c=b<br>END : |

## 循环结构语句的机器级表示

| 示例：用条件转移指令实现循环 | | |
|---|---|---|
| int result =0;<br>for(int i=1;i<=100)<br>{<br>result+=i;<br>} | mov eax,0<br>mov edx,1 | #用eax保存result，初值为0<br>#用edx保存i，初始值为1 ① |
| | cmp edx,100<br>jg L2 | #比较i和100<br>#若i>100，转跳到L2执行 ② |
| | L1: | #循环主体 |
| | add eax,edx<br>inc edx | #实现result +=i<br>#inc自增指令，实现i++ ③ |
| | cmp edx,100<br>jle L1 | #i和100<br>#若i<=100，转跳到L1执行 ④ |
| | L2: | #跳出循环主体 |

用条件转移指令实现循环,需要4个部分构成
①循环前的初始化
②是否直接跳过循环
③循环主体
④是否继续循环?

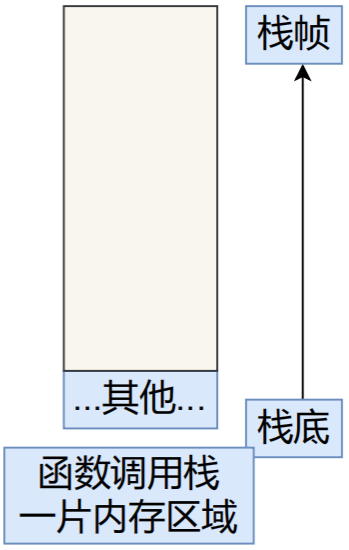
| 示例：用loop指令实现循环 | |
|---|---|
| for(int i=500;i>0;i--){<br>做某些处理;<br>}//循环500轮 | mov ecx ,500 #用ecx作为循环计数器<br>Looptop: #循环的开始<br>...<br>做某些处理<br>...<br>loop Looptop #ecx--，若ecx!=0，跳转到Looptop |

等价于:
dec ecx
cmp ecx,0
jne Looptop

理论上，能用loop指令实现的功能一定能用条件转移指令实现使用loop指令可能会使代码更清晰简洁

补充: loopx指令——如 loopnz, loopz
loopnz—当ecx!=0 &&ZF==0时，继续循环
loopz—当ecx!=0 &&ZF==1时，继续循环

# 高级语言的函数调用




```
void main()
{
...
P();
...
return;
}
```

```
void P( ){
...
Q( );
 ...
caller ( ) ;'
..
return;
}
```

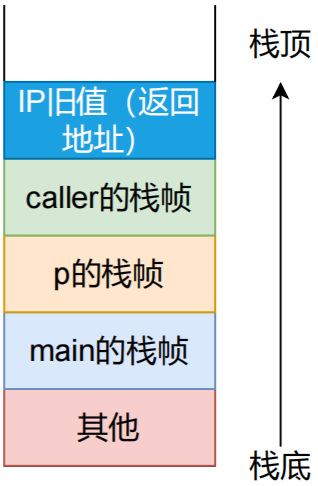
```
void Q( ) {
...
return;
}
```

```
int caller ( ){
int templ=125;
int temp2=80;
int sum=add ( templ,temp2);
return sum;
}
```

```
int add ( int x,
int y){
return x+y;
}
```

函数的栈帧(Stack Frame)
保存函数大括号内定义的局部变量、保存函数调用相关的信息

## call、ret 指令




```
caller :
push ebp
mov ebp, esp
sub esp, 24
mov [ ebp - 12 ], 125
mov [ ebp - 8 ],80
mov eax, [ebp - 8 ]
mov [esp + 4] ,eax
mov eax, [ebp - 12 ]
mov esp , eax
call add
mov [ ebp - 4], eax
mov eax, [ebp - 4 ]
leave
ret
```

IP旧值

```
add :
push ebp
mov ebp，esp
mov eax，[ ebp + 12 ]
mov edx，[ ebp + 8 ]
add eax，edx
leave
ret
```

函数调用指令：call<函数名>
函数返回指令：ret

call 指令的作用：
① 将IP旧值压栈保存（保存在函数的栈帧顶部）
②设置IP新值，无条件转移至被调用函数的第一条指令

ret 指令的作用：
从函数的栈帧顶部找到IP旧值，将其出栈并恢复IP寄存器

指令系统

里昂学长

# 函数调用栈时，如何切换栈帧

```
caller:
push ebp
mov ebp,esp
sub esp ,24
mov [ ebp-12 ],125
mov [ ebp-8 ] ,80
mov eax , [ ebp-8 ]
mov [ esp+4 ] ,eax
mov eax , [ ebp-12]
mov esp,eax
call add
mov [ ebp-4 ] ,eax
mov eax , [ ebp-4 ]
leave
ret
```

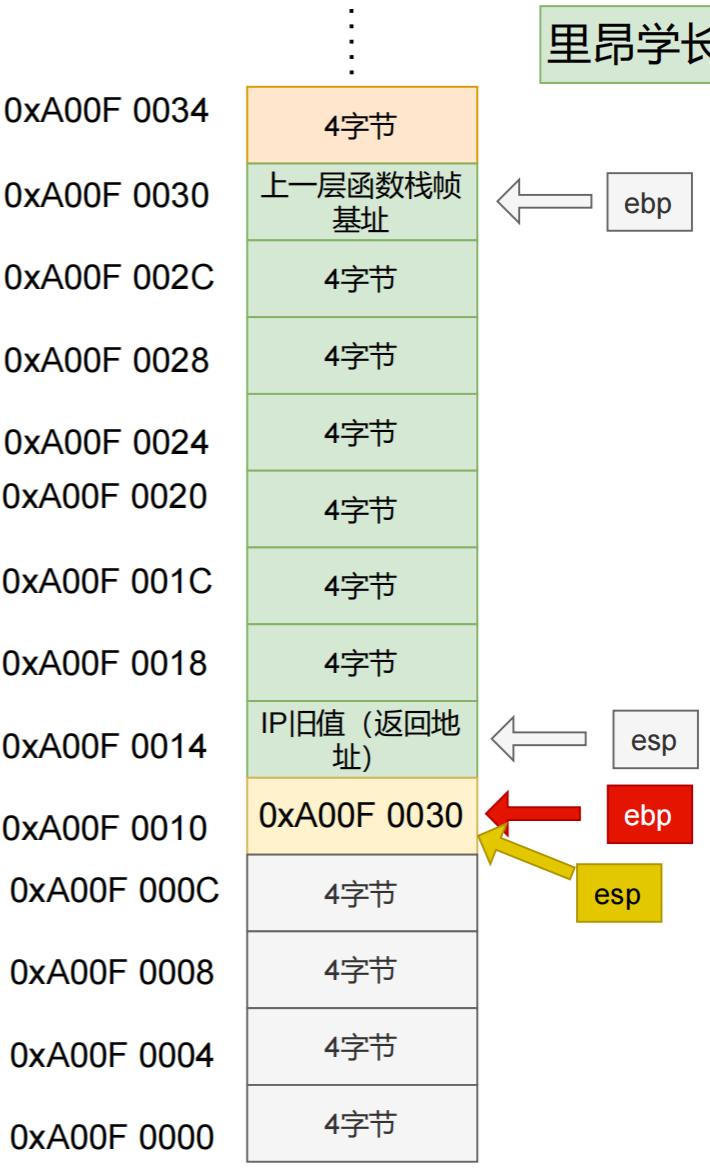
IP旧值

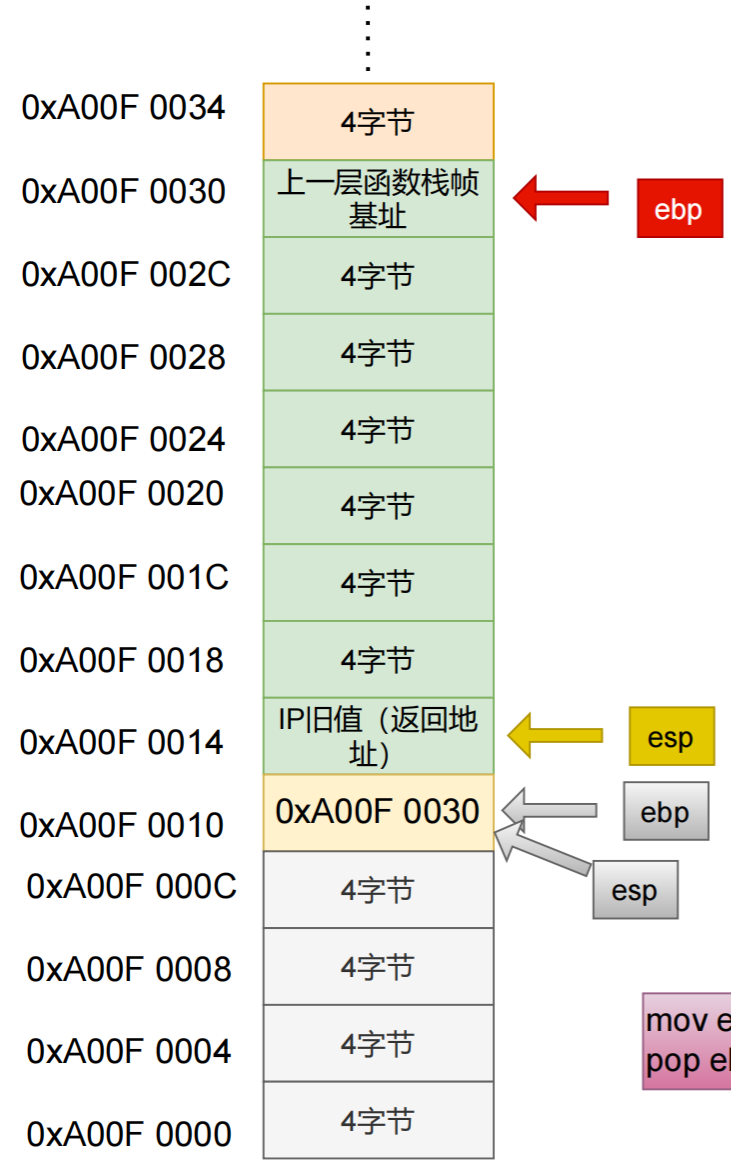
```
add:
push ebp
mov ebp ,esp
mov eax,[ebp+12]
mov edx,[ebp+8]
add eax ,edx
leave
ret
```

指令:enter

等价

```
push ebp  #保存上一层函数的栈帧基址(ebp旧值)
mov ebp,esp  #设置当前函数的栈帧基址（(ebp新值)
```




# 函数返回栈时，如何切换栈帧

```
caller:
push ebp
mov ebp,esp
sub esp ,24
mov [ ebp-12 ],125
mov [ ebp-8 ] ,80
mov eax , [ ebp-8 ]
mov [ esp+4 ] ,eax
mov eax , [ ebp-12]
mov esp,eax
call add
mov [ ebp-4 ] ,eax
mov eax , [ ebp-4 ]
leave
ret
```

IP旧值

```
add:
push ebp
mov ebp ,esp
mov eax,[ebp+12]
mov edx,[ebp+8]
add eax ,edx
leave
ret
```

指令:leave

等价

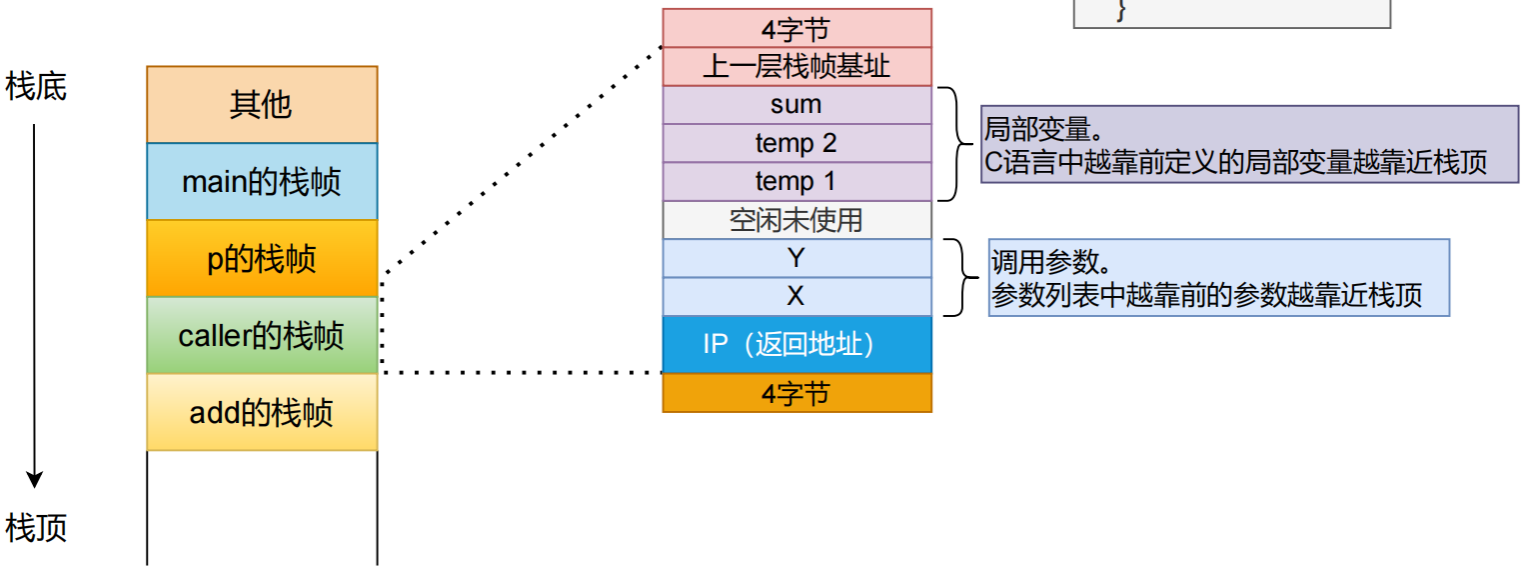
```
mov esp, ebp#让esp指向当前栈帧的底部
pop ebp#将esp所指元素出栈，写入寄存器ebp
```

## 一个栈帧内可能包含哪些内容?

- gcc编译器将每个栈帧大小设置为16B的整数倍（当前函数的栈帧除外），
- 因此栈帧内可能出现空闲未使用的区域。
- 通常将局部变量集中存储在栈帧底部区域
- 通常将调用参数集中存储在栈帧顶部区域
- 栈帧最底部一定是上一层栈帧基址 (ebp旧值)
- 栈帧最顶部一定是返回地址 (当前函数的栈帧除外)

```
int caller（）{
int temp1=125;
int temp2=80;
int sum=add(temp1,
temp2);
return sum;
}
```

```
int add ( int x, int y ){
  return x+7 ;
}
```




## 如何区分x86和MIPS

①观察是否有注释——通常来说，真题中x86汇编语言不会给太多注释（考研中默认大家懂x86）

②观察指令长度是否固定
x86属于CISC，指令长度不固定
MIPS属于RISC，指令长度固定

③观察寄存器名
x86的寄存器名为eax、 ebx、ecx、 edx...
MIPS的寄存器名为R[0]、R[1]、R[2].....

## 如何区分ATT格式和Intel格式

| | ATT格式 | Intel格式 |
|---|---|---|
| 目的操作数d、源操作数s | op s, d<br>注:源操作数在左，目的操作数在右 | op d, s<br>注:源操作数在右，目的操作数在左 |
| 寄存器的表示 | mov %ebx,%eax<br>注:寄存器名之前必须加"%" | mov eax, ebx<br>注:直接写寄存器名即可 |
| 立即数的表示 | mov $985,%eax<br>注:立即数之前必须加“$” | mov eax, 985<br>注:用“中括号” |
| 主存地址的表示 | mov %eax , (af996h)<br>注:用“小括号” | mov [af996h], eax<br>注:用“中括号” |
| 读写长度的表示 | movb $5, (af996h)<br>movw $5, (af996h)<br>movl $5, (af996h)<br>addb $4,(af996h)<br>注:指令后加b、w、l分别表示读写长度为byte、word、dword | mov byte ptr [af996h],5<br>mov word ptr [af996h],5<br>mov dword ptr [af996h],5<br>add byte ptr [af996h],4<br>注:在主存地址前说明读写长度byte、word、dword |
| 主存地址偏移量的表示 | movl -8(%ebx), %eax<br>注:偏移量基址)<br>movl 4(%ebx, %ecx,32),%eax<br>注:偏移量(基址,变址,比例因子) | mov eax, [ebx- 8]<br>注;[基址+偏移量]<br>mov eax, [ebx + ecx*32+4]<br>注:[基址+变址*比例因子+偏移量] |