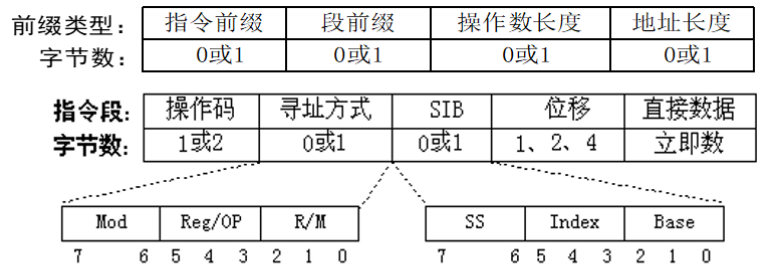
Chapter3 程序的转换及机器级表示

第一讲：程序转换概述

机器指令与汇编指令



操作码：opcode; w：与机器模式（16 / 32位）一起确定寄存器位数（AL / AX / EAX）；D：操作方向

Instruction Prefix：0/1个字节（可选）
Address-Size Prefix： 切换默认地址长度为32位或16位，0/1个字节（可选）0x67
Operand-Size Prefix： 切换默认操作数长度为32位或16位，0/1个字节（可选） 0x66
Segment Override：若需要，用指定的段寄存器取代缺省段寄存器，0/1个字节（可选）

第二讲：IA-32 指令系统

支持的数据类型及格式

C 语言声明	Intel 操作数类型	汇编指令长度后缀	存储长度（位）
(unsigned) char	整数 / 字节	b	8
(unsigned) short	整数 / 字	w	16
(unsigned) int	整数 / 双字	l	32
(unsigned) long int	整数 / 双字	l	32
(unsigned) long long int	-	-	2×32
char *	整数 / 双字	l	32
float	单精度浮点数	s	32
double	双精度浮点数	l	64
long double	扩展精度浮点数	t	80 / 96

寄存器组织

IA-32的寄存器组织

IA-32的寄存器组织

编号	8 位寄存器	16 位寄存器	32 位寄存器	64 位寄存器	128 位寄存器
000	AL	AX	EAX	MM0 / ST(0)	XMM0
001	CL	CX	ECX	MM1 / ST(1)	XMM1
010	DL	DX	EDX	MM2 / ST(2)	XMM2
011	BL	BX	EBX	MM3 / ST(3)	XMM3
100	AH	SP	ESP	MM4 / ST(4)	XMM4
101	CH	BP	EBP	MM5 / ST(5)	XMM5
110	DH	SI	ESI	MM6 / ST(6)	XMM6
111	BH	DI	EDI	MM7 / ST(7)	XMM7

寻址方式

寻址方式	说明
立即寻址	指令直接给出操作数
寄存器寻址	指定的寄存器R的内容为操作数
位移	LA=(SR)+A
基址寻址	LA=(SR)+(B)
基址加位移	LA=(SR)+(B)+A
比例变址加位移	LA=(SR)+(I)×S + A
基址加变址加位移	LA=(SR)+(B)+(I)+A
基址加比例变址加位移	LA=(SR)+(B)+(I)×S+A
相对寻址	LA=(PC)+A 跳转目标指令地址

存储器操作数

注：LA:线性地址 (X):X的内容 SR:段寄存器 PC:程序计数器 R:寄存器
A:指令中给定地址段的位移量 B:基址寄存器 I:变址寄存器 S:比例系数

- SR段寄存器（间接）确定操作数所在段的段基址
- 有效地址给出操作数在所在段的偏移地址
- 寻址过程涉及到“分段虚拟管理方式”，将在第6章讨论

指令的两个操作数不能都是内存单元（必须以寄存器为媒介）

第三讲：IA-32常用指令类型及其操作

传送指令

（1）通用数据传送指令

MOV：一般传送，包括movb、movw和movl等

MOVS：符号扩展传送，如movsbw、movswl等

MOVZ：零扩展传送，如movzwl、movzbl等

XCHG：数据交换

PUSH/POP：入栈/出栈，如pushl,pushw,popl,popw等

（2）地址传送指令

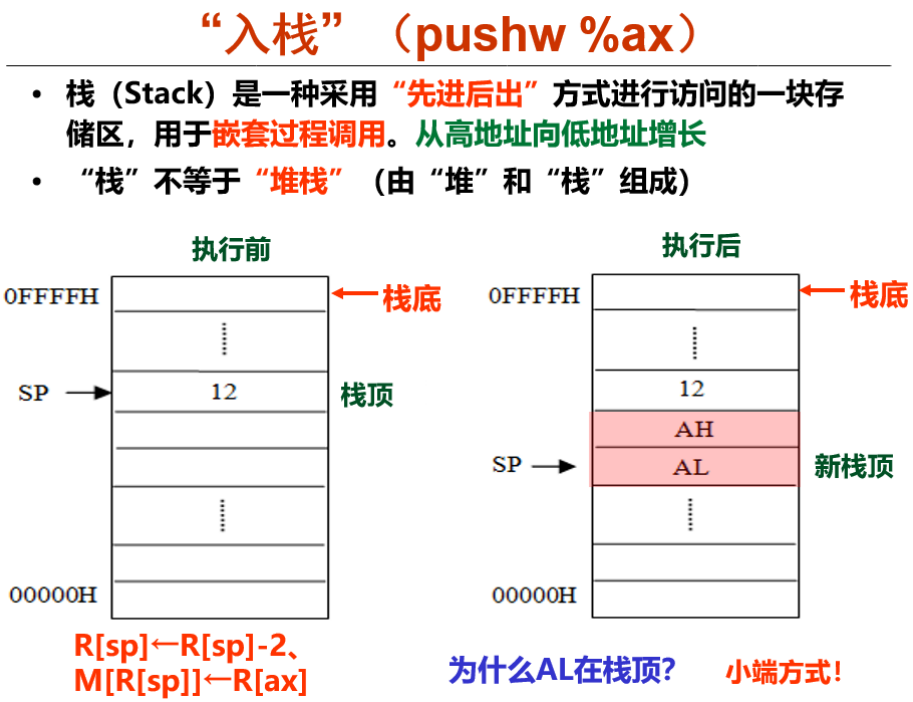
LEA：加载有效地址，如leal (%edx,%eax), %eax”的功能为R[eax]←R[edx] + R[eax]，执行前，若R[edx] = i，R[eax] = j，则指令执行后，R[eax] = i + j（可以用作实现简单算数）

（3）输入输出指令

IN和OUT：I/O端口与寄存器之间的交换

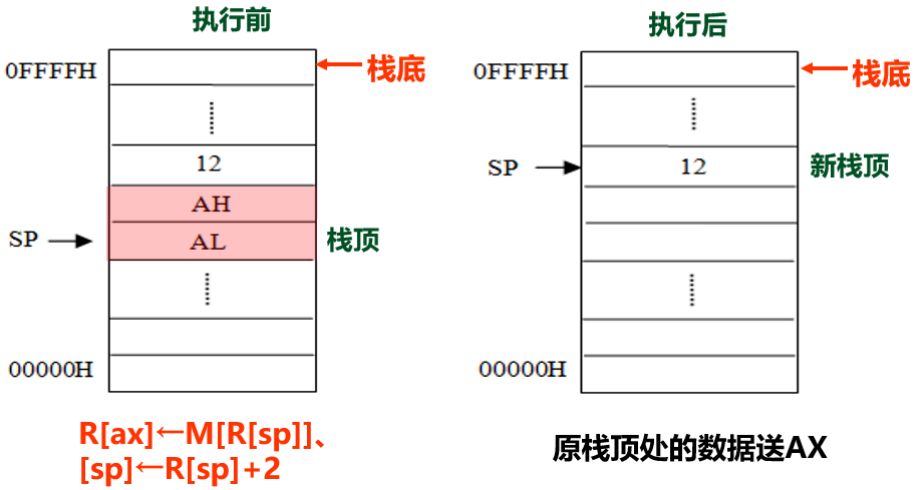
（4）标志传送指令

PUSHF、POPF：将EFLAG压栈，或将栈顶内容送EFLAG



“出栈” （popw %ax）

- 栈（Stack）是一种采用“先进后出”方式进行访问的一块存储区，用于嵌套过程调用。从高地址向低地址增长



R[ax]←M[R[sp]]、
[sp]←R[sp]+2

原栈顶处的数据送AX

定点算术指令

（1）加 / 减运算（影响标志、不区分无/带符号）

ADD：加，包括addb、addw、addl等

SUB：减，包括subb、subw、subl等

（2）增1 / 减1运算（影响除CF以外的标志、不区分无/带符号）

INC：加，包括incb、incw、incl等

DEC：减，包括decb、decw、decl等

（3）取负运算（影响标志、若对0取负，则结果为0/CF=0,否则CF=1）

NEG：取负，包括negb、negw、negl等

（4）比较运算（做减法得到标志、不赋值、不区分无/带符号）

CMP：比较，包括cmpb、cmpw、cmpl等

（5）乘 / 除运算（区分无/带符号）

MUL / IMUL：无符号乘 / 带符号乘（影响标志OF和CF）

DIV/ IDIV：无符号除 / 带符号除

- **布斯乘法：** “imulb %bl”

R[ax]=B4H × 11H

```
          1 0 1 1  0 1 0 0
        x 0 0 1-1 0 0 1-1    0001 0001 0
-------------------------------------------
  0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 0
  1 1 1 1 1 1 1 1 1 0 1 1 0 1 0 0
  0 0 0 0 0 1 0 0 1 1 0 0
  1 1 1 1 1 0 1 1 0 1 0 0
-------------------------------------------
  1 1 1 1 1 0 1 0 1 1 1 1 0 1 0 0
  AH=?          AL=?
```

R[ax]=FAF4H, 真值为-1292=-76 × 17

位操作指令

（1）逻辑运算（仅NOT不影响标志，其他指令OF=CF=0，而ZF和SF根据结果设置：若全0，则ZF=1；若最高位为1，则SF=1 ）

NOT：非，包括 notb、notw、notl等

AND：与，包括 andb、andw、andl等

OR：或，包括 orb、orw、orl等

XOR：异或，包括 xorb、xorw、xorl等

（2）TEST：做"与"操作测试，仅影响标志（一般用于判断是否为0）

（3）移位运算（左/右移时，最高/最低位送CF）

SHL/SHR：逻辑左/右移，包括 shlb、shrw、shrl等

SAL/SAR：算术左/右移，左移判溢出，右移高位补符

（移位前、后符号位发生变化，则OF=1 ）

ROL/ROR: 循环左/右移，包括 rolb、rorw、roll等

RCL/RCR: 带循环左/右移，将CF作为操作数一部分循环移位

控制转移指令

（1）无条件转移指令

JMP DST：无条件转移到目标指令DST处执行

（2）条件转移

Jcc DST：cc为条件码，根据标志（条件码）判断是否满足条件，若满足，则转移到目标指令DST处执行，否则按顺序执行

分三类：

(1)根据单个标志的值转移　(2)按无符号整数比较转移　(3)按带符号整数比较转移

序号	指令	转移条件	说明
1	jc label	CF=1	有进位/借位
2	jnc label	CF=0	无进位/借位
3	je/jz label	ZF=1	相等/等于零
4	jne/jnz label	ZF=0	不相等/不等于零
5	js label	SF=1	是负数
6	jns label	SF=0	是非负数
7	jo label	OF=1	有溢出
8	jno label	OF=0	无溢出
9	ja/jnbe label	CF=0 AND ZF=0	无符号整数 A＞B
10	jae/jnb label	CF=0 OR ZF=1	无符号整数 A≥B
11	jb/jnae label	CF=1 AND ZF=0	无符号整数 A＜B
12	jbe/jna label	CF=1 OR ZF=1	无符号整数 A≤B
13	jg/jnle label	SF=OF AND ZF=0	带符号整数 A＞B
14	jge/jnl label	SF=OF OR ZF=1	带符号整数 A≥B
15	jl/jnge label	SF≠OF AND ZF=	带符号整数 A＜B
16	jle/jng label	SF≠OF OR ZF=1	带符号整数 A≤B

（3）条件设置

SETcc DST：将条件码cc保存到DST（通常是一个8位寄存器 ）

（4）调用和返回指令 （用于过程调用）

CALL DST：返回地址RA入栈，转DST处执行

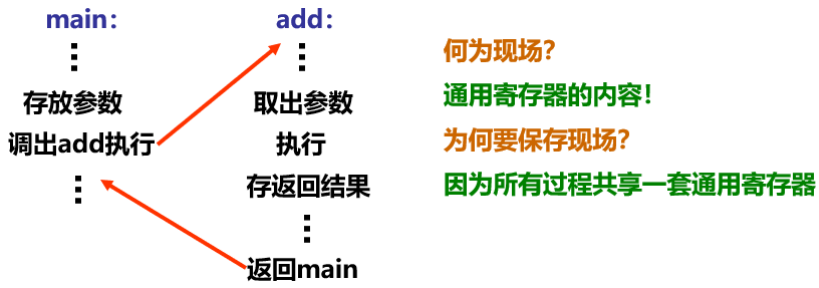
RET：从栈中取出返回地址RA，转到RA处执行

（5）中断指令 （详见第7、8章）

浮点运算指令

了解即可

第四讲：过程的表示



过程调用的执行步骤(P为调用者，Q为被调用者)

（1）P将入口参数（实参）放到Q能访问到的地方；

（2）P保存返回地址，然后将控制转移到Q；

（3）Q保存P的现场，并为自己的非静态局部变量分配空间；

（4）执行Q的过程体（函数体）；

（5）Q恢复P的现场，释放局部变量空间；

（6）Q取出返回地址，将控制转移到P。

IA-32的寄存器使用约定

- 调用者保存寄存器：EAX、EDX、ECX

  当过程P调用过程Q时，Q可以直接使用这三个寄存器，不用将它们的值保存到栈中。如果P在从Q返回后还要用这三个寄存器的话，P应在转到Q之前先保存，并在从Q返回后先恢复它们的值再使用。

- 被调用者保存寄存器：EBX、ESI、EDI

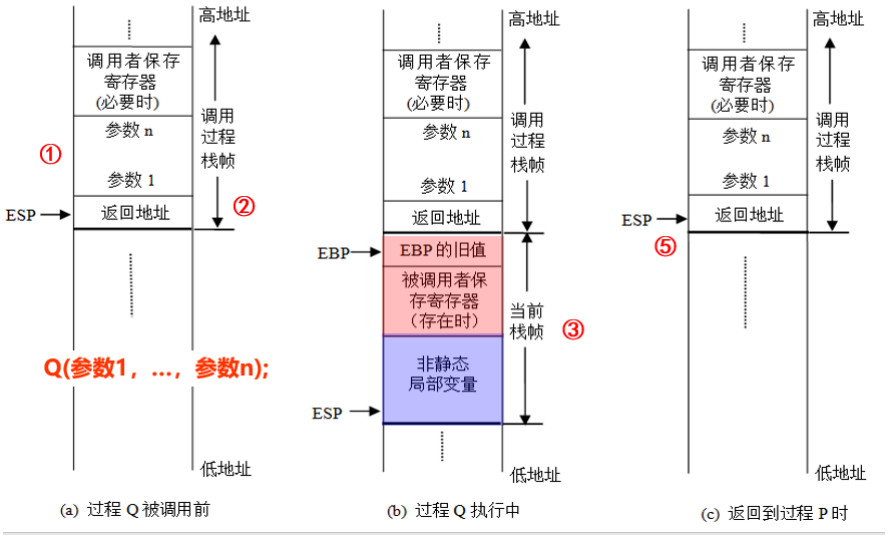
  Q必须先将它们的值保存到栈中再使用它们，并在返回P之前恢复它们的值。

- EBP和ESP分别是帧指针寄存器和栈指针寄存器，分别用来指向当前栈帧的底部和顶。

栈帧的变化和执行过程

过程调用的机器级表示

- **过程调用过程中栈和栈帧的变化 (Q为被调用过程)**



- **一个C过程的大致结构如下：**
  - **准备阶段**
    - **形成帧底：push指令 和 mov指令**
    - **生成栈帧（如果需要的话）：sub指令 或 and指令**
    - **保存现场（如果有被调用者保存寄存器）：mov指令**
  - **过程（函数）体**
    - **分配局部变量空间，并赋值**
    - **具体处理逻辑，如果遇到函数调用时**
      - **准备参数：将实参送栈帧入口参数处**
      - **CALL指令：保存返回地址并转被调用函数**
    - **在EAX中准备返回参数**
  - **结束阶段**
    - **退栈：leave指令 或 pop指令**
    - **取返回地址返回：ret指令**

参数访问和重要指令

IA-32中，若参数类型是unsigned char、char或unsigned short、short，也都分配4个字节

故在被调用函数中，使用R[ebp]+8、R[ebp]+12、R[ebp]+16作为有效地址来访问函数的入口参数

```
movl  参数3，8(%esp)
……
movl  参数1, (%esp)
call   add
```

准备入口参数

每个过程开始两条指令：

pushl %ebp

movl %esp, %ebp

call等价于：

R[esp]←R[esp] - 4

M[R[esp]]←返回地址

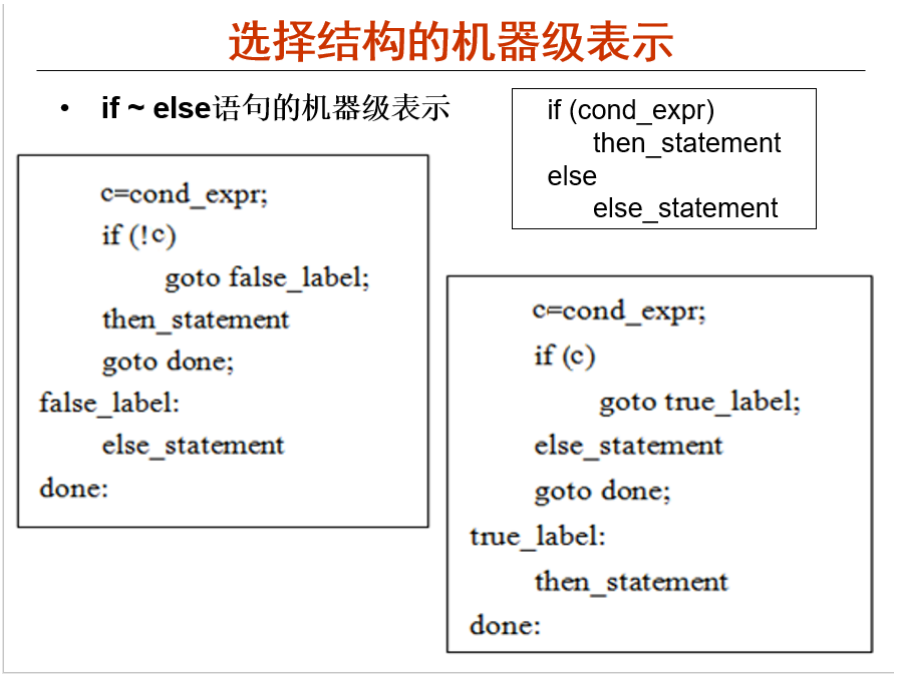
R[eip]←add函数首地址

leave等价于：

movl %ebp, %esp

popl %ebp

选择语句的机器级表示



具体例子看课件"Ch3-程序的表示-3"

第五讲：复杂数据类型的分配和访问

数组的分配和访问

基址 + 比例变址

假定数组A的首地址存放在EDX中，i 存放在ECX中，现要将A[i]取到AX中，则所用的汇编指令是什么?

movw (%edx, %ecx, 2), %ax

数组定义	数组名	数组元素类型	数组元素大小（B）	数组大小（B）	起始地址	元素 i 的地址
char S[10]	S	char	1	10	&S[0]	&S[0]+i
char * SA[10]	SA	char *	4	40	&SA[0]	&SA[0]+4*i
double D[10]	D	double	8	80	&D[0]	&D[0]+8*i
double * DA[10]	DA	double *	4	40	&DA[0]	&DA[0]+4*i

结构体的分配与访问

分配在栈中的auto结构型变量的首地址由EBP或ESP来定位

分配在静态区的结构型变量首地址是一个确定的静态区地址

结构型变量 x 各成员首址可用"基址加偏移量"的寻址方式

结构体数据作为入口参数

当结构体变量需要作为一个函数的形参时，形参和调用函数中的实参应具有相同结构

有按值传递和按地址传递两种方式

- 若采用按值传递，则结构成员都要复制到栈中参数区，这既增加时间开销又增加空间开销
- 通常应按地址传递，即：在执行CALL指令前，仅需传递指向结构体的指针而不需复制每个成员到栈中

联合体的分配与访问

联合体各成员共享存储空间，按最大长度成员所需空间大小为目标

数据的对齐

I386 System V ABI策略：short型为2字节边界对齐，其他的如int、double、long double和指针等类型都是4字节边界对齐（即为4的倍数）

i386 System V ABI对struct结构体数据的对齐方式有如下几条规则：

① 整个结构体变量的对齐方式与其中对齐方式最严格的成员相同；

② 每个成员在满足其对齐方式的前提下，取最小的可用位置作为成员在结构体中的偏移量，这可能导致内部插空；

③ 结构体大小应为对齐边界长度的整数倍，这可能会导致尾部插空。

前两条规则是为了保证结构体中的任意成员都能以对齐的方式访问。

第③条规则是为了保证使结构体数组中的每个元素都能满足对齐要求

#pragma pack(n)

- **为编译器指定结构体或类内部的成员变量的对齐方式。**
- **当自然边界（如int型按4字节、short型按2字节、float按4字节）比n大时，按n字节对齐。**
- **缺省或#pragma pack()，按自然边界对齐。**

__attribute__((aligned(m)))

- **为编译器指定一个结构体或类或联合体或一个单独的变量(对象)的对齐方式。**
- **按m字节对齐(m必须是2的幂次方)，且其占用空间大小也是m的整数倍，以保证在申请连续存储空间时各元素也按m字节对齐。**

__attribute__((packed))

- **不按边界对齐，称为紧凑方式。**

packed(n)与自然边界，取偏小的那个

aligned(n)表示**成员变量的地址**不能跨n的整数倍，比如n=8表示某个变量不能垮8，16，24来分配

第六讲 越界访问和缓冲区溢出

产生溢出的原因

- C语言中的数组元素可使用指针来访问，因而对数组的引用没有边界约束，也即程序中对数组的访问可能会有意或无意地超越数组存储区范围而无法发现。
- 数组存储区可看成是一个缓冲区，超越数组存储区范围的写入操作称为缓冲区溢出。
- 例如，对于一个有10个元素的char型数组，其定义的缓冲区有10个字节。若写一个字符串到这个缓冲区，那么只要写入的字符串多于9个字符（结束符'\0'占一个字节），就会发生"写溢出"。
- 缓冲区溢出是一种非常普遍、非常危险的漏洞，在各种操作系统、应用软件中广泛存在。
- 缓冲区溢出攻击是利用缓冲区溢出漏洞所进行的攻击行动。利用缓冲区溢出攻击，可导致程序运行失败、系统关机、重新启动等后果。

两个方面的防范

从程序员角度去防范

- 用辅助工具帮助程序员查漏，例如，用grep来搜索源代码中容易产生漏洞的库函数（如strcpy和sprintf等）的调用；用fault injection查错

从编译器和操作系统方面去防范

- 地址空间随机化ASLR

  是一种比较有效的防御缓冲区溢出攻击的技术

  目前在Linux、FreeBSD和Windows Vista等OS使用
- 栈破坏检测
- 可执行代码区域限制