第三章 程序的转换与机器级表示

程序转换概述

IA-32 /x86-64指令系统

C语言程序的机器级表示

复杂数据类型的分配和访问

越界访问和缓冲区溢出、x86-64架构

程序的转换与机器级表示

- 主要教学目标
  - 了解高级语言与汇编语言、汇编语言与机器语言之间的关系
  - 掌握有关指令格式、操作数类型、寻址方式、操作类型等内容
  - 了解高级语言源程序中的语句与机器级代码之间的对应关系
  - 了解复杂数据类型（数组、结构等）的机器级实现
- 主要教学内容
  - 介绍C语言程序与IA-32机器级指令之间的对应关系。
  - 主要包括：程序转换概述、IA-32指令系统、C语言中控制语句和过程调用等机器级实现、复杂数据类型（数组、结构等）的机器级实现等。
  - 本章所用的机器级表示主要以汇编语言形式表示为主。

采用逆向工程方法！

程序的机器级表示

- 分以下五个部分介绍
  - 第一讲：程序转换概述
    - 机器指令和汇编指令
    - 机器级程序员感觉到的属性和功能特性
    - 高级语言程序转换为机器代码的过程
  - 第二讲：IA-32 /x86-64指令系统
  - 第三讲： C语言程序的机器级表示
    - 过程调用的机器级表示
    - 选择语句的机器级表示
    - 循环结构的机器级表示
  - 第四讲：复杂数据类型的分配和访问
    - 数组的分配和访问
    - 结构体数据的分配和访问
    - 联合体数据的分配和访问
    - 数据的对齐
  - 第五讲：越界访问和缓冲区溢出

从高级语言程序出发，用其对应的机器级代码以及内存（栈）中信息的变化来说明底层实现

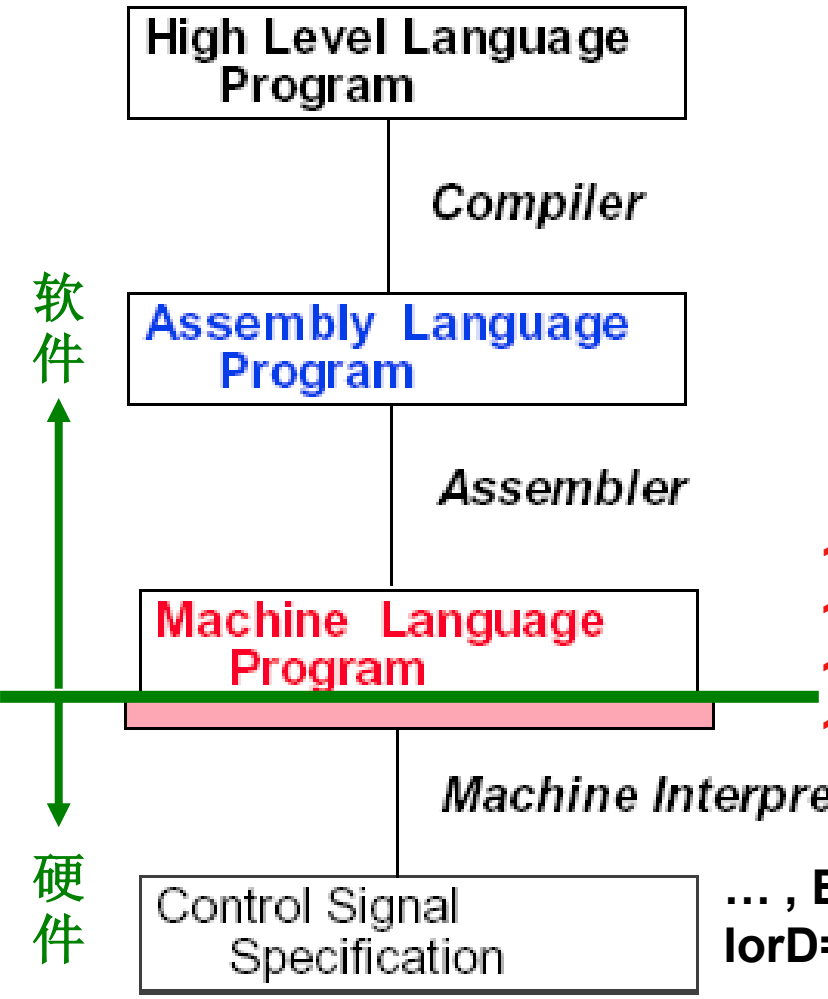
围绕C语言中的语句和复杂数据类型，解释其在底层机器级的实现方法

“指令”的概念

- 有微指令、机器指令、汇编指令、伪（宏）指令等指令相关概念
- 微指令是微程序级命令，属于硬件范畴
- 机器指令介于二者之间，处于硬件和软件的交界面
  - 本章中提及的指令都指机器指令
- 汇编指令是机器指令的汇编表示形式，即符号表示
- 机器指令和汇编指令一一对应，它们都与具体机器结构有关，都属于机器级指令
- 伪指令是由若干汇编指令组成的序列，属于软件范畴

SKIP

回顾：Hardware/Software Interface



BACK

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

汇编指令

```
lw $15, 0($2)
lw $16, 4($2)
sw $16, 0($2)
sw $15, 4($2)
```

伪指令 swap 0($2),4($2)

机器指令

```
1000 1100 0100 1111 0000 0000 0000 0000
1000 1100 0101 0000 0000 0000 0000 0100
1010 1100 0101 0000 0000 0000 0000 0000
1010 1100 0100 1111 0000 0000 0000 0100
```

… , EXTop=1,ALUSelA=1,ALUSelB=11,ALUop=add,
IorD=1,Read,MemtoReg=1,RegWr=1,.....

微指令 → … 1 1 11 100 1 0 1 1 …

机器级指令

- 机器指令和汇编指令一一对应，都是机器级指令
- 机器指令是一个0/1序列，由若干字段组成
- 如8086/8088指令：

补码11111010的真值为多少?

100010 DW	mod	reg	r/m	disp8
100010 0 0	01	001	001	11111010

操作码　寻址方式　寄存器编号　立即数(位移量)

- 汇编指令是机器指令的符号表示（可能有不同的格式）

mov [bx+di-6], cl　或　movb %cl, -6(%bx,%di)

Intel格式　AT&T 格式

本课程采用AT&T格式

mov、movb、bx、%bx等都是助记符

指令的功能为：M[R[bx]+R[di]-6]←R[cl]

寄存器传送语言 RTL（Register Transfer Language）

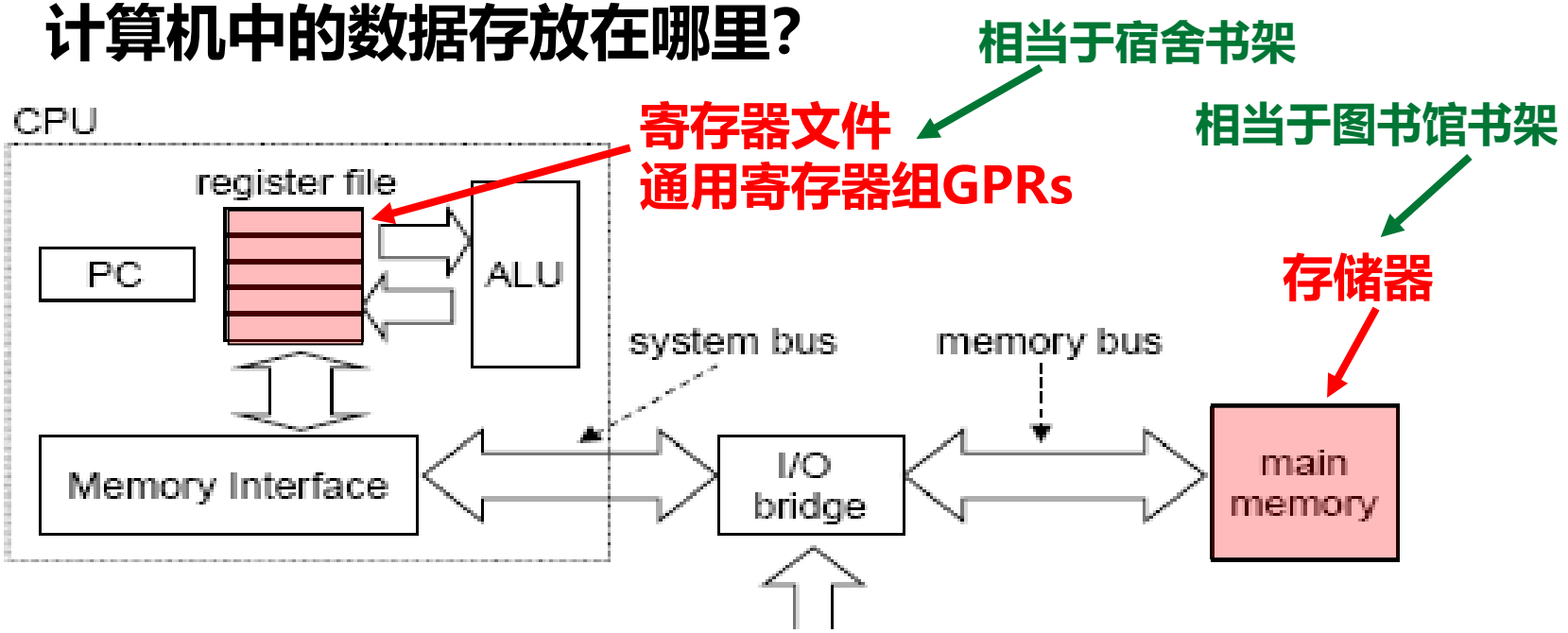
R：寄存器内容
M：存储单元内容

回顾：指令集体系结构ISA

- ISA（Instruction Set Architecture）位于软件和硬件之间
- 硬件的功能通过ISA提供出来
- 软件通过ISA规定的”指令”使用硬件
- ISA规定了：
  - 可执行的指令的集合，包括指令格式、操作种类以及每种操作对应的操作数的相应规定；
  - 指令可以接受的操作数的类型；
  - 操作数所能存放的寄存器组的结构，包括每个寄存器的名称、编号、长度和用途；
  - 操作数所能存放的存储空间的大小和编址方式；
  - 操作数在存储空间存放时按照大端还是小端方式存放；
  - 指令获取操作数的方式，即寻址方式；
  - 指令执行过程的控制方式，包括程序计数器、条件码定义等。

回顾：计算机中数据的存储

- 计算机中的数据存放在哪里？



指令中需给出的信息：

操作性质（操作码）

源操作数1 或/和 源操作数2　（立即数、寄存器编号、存储地址）

目的操作数地址　（寄存器编号、存储地址）

存储地址的描述与操作数的数据结构有关！

IA-32机器指令格式

前缀类型：	指令前缀	段前缀	操作数长度	地址长度
字节数：	0或1	0或1	0或1	0或1

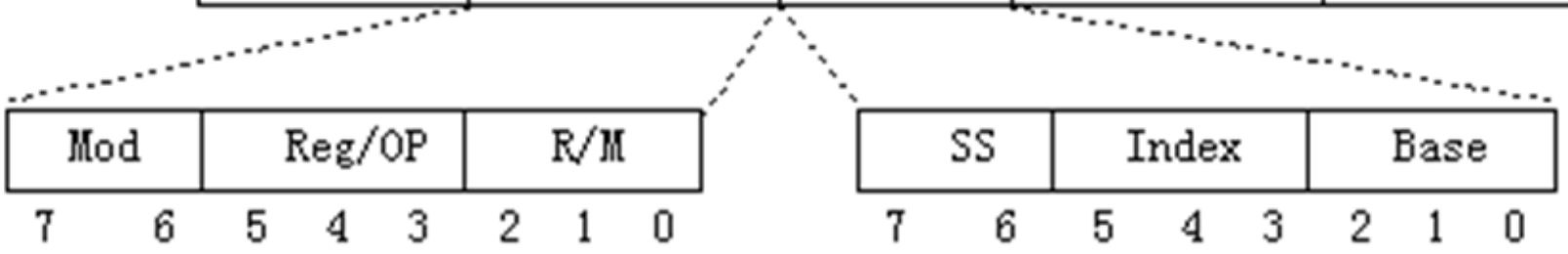
可以同时出现，无先后顺序关系。

指令前缀：加锁(LOCK)和重复执行(REP/REPE/REPZ/REPNE/REPNZ)两种，LOCK编码为F0H，REPNE、REP编码分别为F2H和F3H

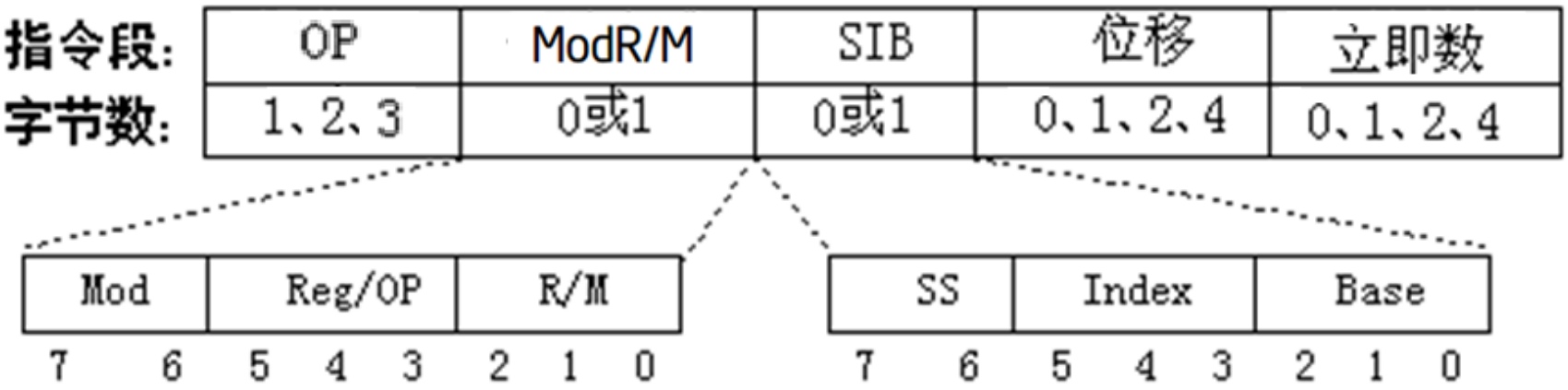
段前缀：指定指令所使用的非默认段寄存器，包括 2EH(CS)、36H(SS)、3EH(DS)、26H(ES)、64H(FS)、65H(GS)

操作数长度和地址长度前缀：分别为66H和67H

指令段：	OP	ModR/M	SIB	位移	立即数
字节数：	1、2、3	0或1	0或1	0、1、2、4	0、1、2、4



IA-32机器指令格式



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，确定Reg是源操作数还是目...

寻址方式： mod、r/m、 reg/op三个字段与w字段和机器模式一起确定操作数所在的寄存器编号或**有效地址计算方式**:

Mod和R/M共5位，表示另一个操作数的寻址方式，可组合成32种情况，当Mod=11时，为寄存器寻址方式，3位R/M表示寄存器编号，其他24种情况都是存储器寻址方式。是否在ModR/M字节后跟一个SIB字节，由Mod和R/M组合确定，例如，当Mod=00且R/M=100时，一定跟SIB字节，寻址方式由SIB确定。

SIB中基址B和变址I存放在寄存器中，寄存器编号占3位；SS给出比例因子

位移量和立即数的长度可以是：1B（8位）、2B（16位）、4B（32位）

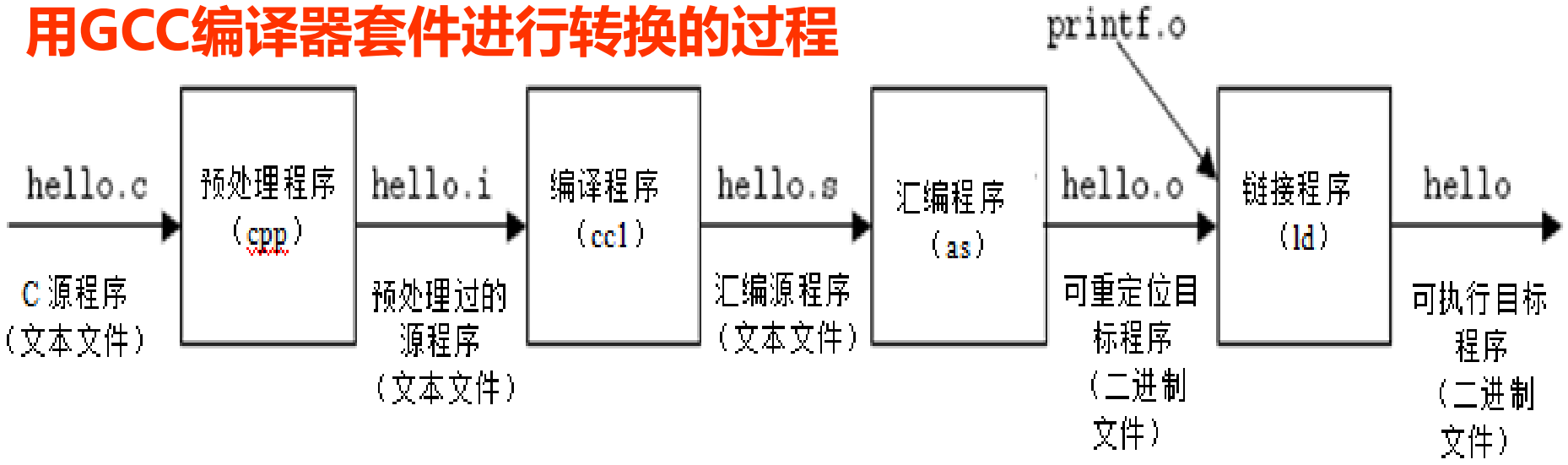
例子“C7 44 24 04 01 00 00 00”见教材3.2.3节

r8(/r)	AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)	AX	CX	DX	BX	SP	BP	SI	DI
r32(/r)	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
/digit (Opcode)	0	1	2	3	4	5	6	7
REG =	000	001	010	011	100	101	110	111

Address	Mod	R/M	ModR/M Values in Hexadecimal							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--][--][1]		100	04	0C	14	1C	24	2C	34	3C
disp32		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
disp8[EAX]	01	000	40	48	50	58	60	68	70	78
disp8[ECX]		001	41	49	51	59	61	69	71	79
disp8[EDX]		010	42	4A	52	5A	62	6A	72	7A
disp8[EBX];		011	43	4B	53	5B	63	6B	73	7B
disp8[--][--]		100	44	4C	54	5C	64	6C	74	7C
disp8[EBP]		101	45	4D	55	5D	65	6D	75	7D
disp8[ESI]		110	46	4E	56	5E	66	6E	76	7E
disp8[EDI]		111	47	4F	57	5F	67	6F	77	7F
disp32[EAX]	10	000	80	88	90	98	A0	A8	B0	B8
disp32[ECX]		001	81	89	91	99	A1	A9	B1	B9
disp32[EDX]		010	82	8A	92	9A	A2	AA	B2	BA
disp32[EBX]		011	83	8B	93	9B	A3	AB	B3	BB
disp32[--][--]		100	84	8C	94	9C	A4	AC	B4	BC
disp32[EBP]		101	85	8D	95	9D	A5	AD	B5	BD
disp32[ESI]		110	86	8E	96	9E	A6	AE	B6	BE
disp32[EDI]		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH		111	C7	CF	D7	DF	E7	EF	F7	FF

高级语言程序转换为机器代码的过程

用GCC编译器套件进行转换的过程



预处理：在高级语言源程序中插入所有用#include命令指定的文件和用#define声明指定的宏。

编译：将预处理后的源程序文件编译生成相应的汇编语言程序。

汇编：由汇编程序将汇编语言源程序文件转换为可重定位的机器语言目标代码文件。

链接：由链接器将多个可重定位的机器语言目标文件以及库例程（如printf()库函数）链接起来，生成最终的可执行目标文件。

GCC使用举例

```
1    // test.c
2    #include <stdio.h>
3    int add(int i, int j )
4    {
5        int x = i +j;
6        return x;
7    }
```

test1.c和test2.c，最终生成可执行文件为test

.c test2.c -o test

-级优化，-O2为二级优化，选项-o指出输出文件名

“gcc –c test.s –o test.o”将test.s汇编为test.o

“objdump -d test.o”将test.o 反汇编为

gcc -E test.c -o test.i

gcc -S test.i -o test.s

gcc –S test.c –o test.s

test.s

```
add:
pushl%ebp
movl  %esp, %ebp
subl   $16, %esp
movl  12(%ebp), %eax
movl  8(%ebp), %edx
leal    (%edx, %eax), %eax
movl  %eax, -4(%ebp)
movl  -4(%ebp), %eax
leave
ret
```

00000000 <add>:

0:	55	push %ebp
1:	89 e5	mov %esp, %ebp
3:	83 ec 10	sub $0x10, %esp
6:	8b 45 0c	mov 0xc(%ebp), %eax
9:	8b 55 08	mov 0x8(%ebp), %edx
c:	8d 04 02	lea (%edx,%eax,1), %eax
f:	89 45 fc	mov %eax, -0x4(%ebp)
12:	8b 45 fc	mov -0x4(%ebp), %eax
15:	c9	leave
16:	c3	ret

位移量　机器指令　汇编指令

编译得到的与反汇编得到的汇编指令形式稍有差异

两种目标文件

```
1    // test.c
2    #include <stdio.h>
3    int add(int i, int j )
4    {
5        int x = i +j;
6        return x;
7    }
```

test.o：可重定位目标文件

test：可执行目标文件

“objdump -d test.o”结果

```
00000000 <add>:
 0:  55          push   %ebp
 1:  89 e5       mov    %esp, %ebp
 3:  83 ec 10    sub    $0x10, %esp
 6:  8b 45 0c    mov    0xc(%ebp), %eax
 9:  8b 55 08    mov    0x8(%ebp), %edx
 c:  8d 04 02    lea    (%edx,%eax,1), %eax
 f:  89 45 fc    mov    %eax, -0x4(%ebp)
12:  8b 45 fc    mov    -0x4(%ebp), %eax
15:  c9          leave
16:  c3          ret
```

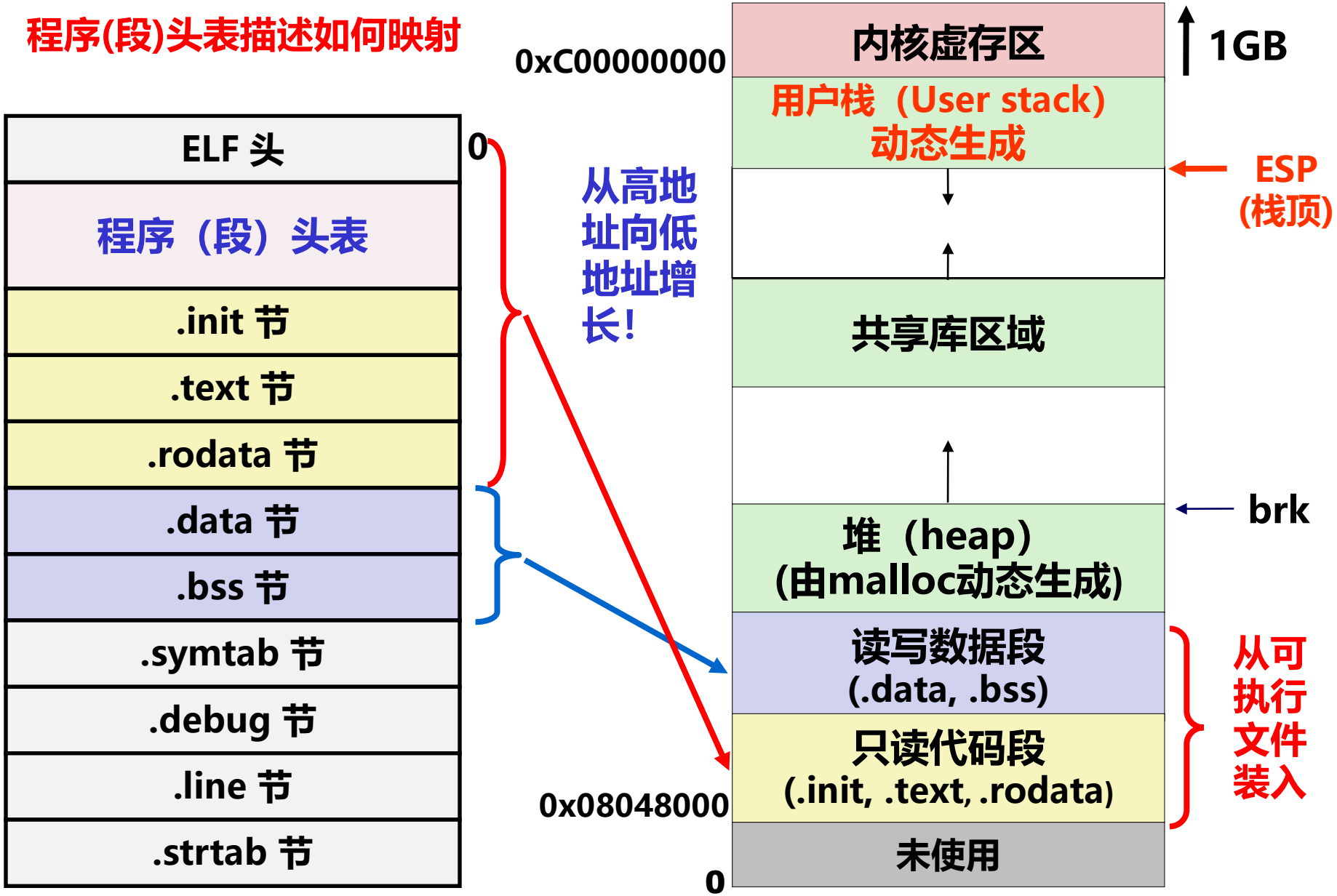
“objdump -d test”结果

```
080483d4 <add>:
80483d4:  55          push ...
80483d5:  89 e5       ...
80483d7:  83 ec 10    ...
80483da:  8b 45 0c    ...
80483dd:  8b 55 08    ...
80483e0:  8d 04 02    ...
80483e3:  89 45 fc    ...
80483e6:  8b 45 fc    ...
80483e9:  c9          ...
80483ea:  c3          ret
```

test.o中的代码从地址0开始，test中的代码从80483d4开始！

可执行文件的存储器映像

程序(段)头表描述如何映射



程序的机器级表示

- 分以下五个部分介绍
  - 第一讲：程序转换概述
    - 机器指令和汇编指令
    - 机器级程序员感觉到的属性和功能特性
    - 高级语言程序转换为机器代码的过程
  - 第二讲：IA-32 /x86-64指令系统
  - 第三讲： C语言程序的机器级表示
    - 过程调用的机器级表示
    - 选择语句的机器级表示
    - 循环结构的机器级表示
  - 第四讲：复杂数据类型的分配和访问
    - 数组的分配和访问
    - 结构体数据的分配和访问
    - 联合体数据的分配和访问
    - 数据的对齐
  - 第五讲：越界访问和缓冲区溢出

从高级语言程序出发，用其对应的机器级代码以及内存（栈）中信息的变化来说明底层实现

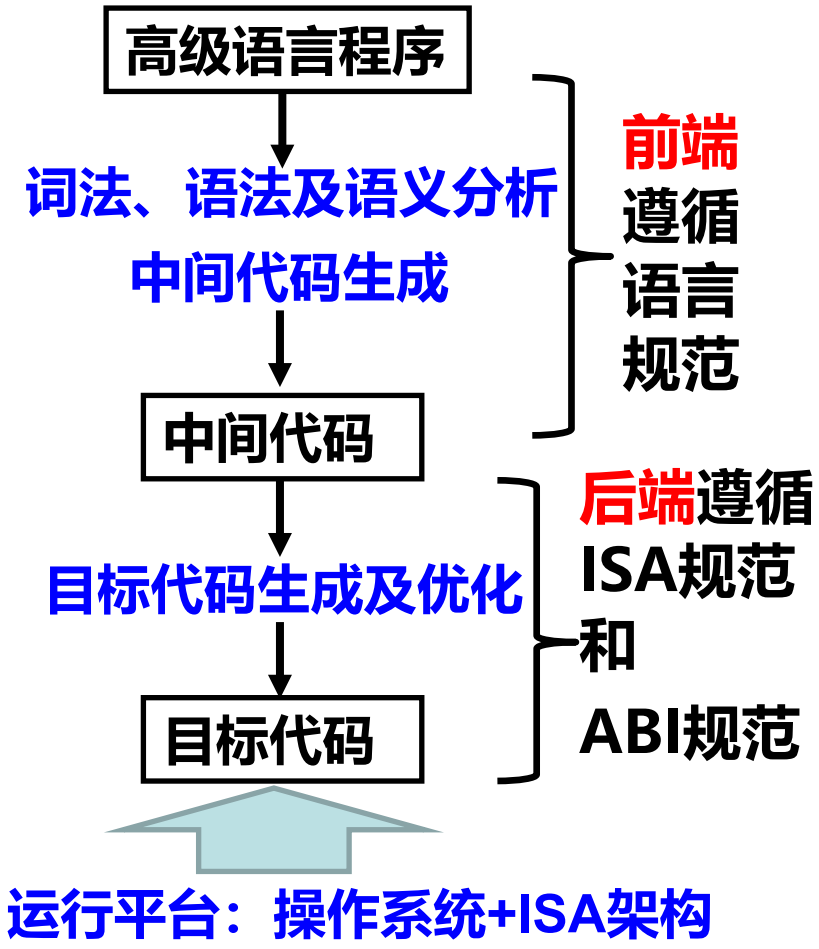
围绕C语言中的语句和复杂数据类型，解释其在底层机器级的实现方法

IA-32/x64指令系统概述

- x86是Intel开发的一类处理器体系结构的泛称
  - 包括 Intel 8086、80286、i386和i486等，因此其架构被称为“x86”
  - 由于数字并不能作为注册商标，因此，后来使用了可注册的名称，如Pentium、PentiumPro、Core 2、Core i7等
  - 现在Intel把32位x86架构的名称x86-32改称为IA-32
- 由AMD首先提出了一个兼容IA-32指令集的64位版本
  - 扩充了指令及寄存器长度和个数等，更新了参数传送方式
  - AMD称其为AMD64，Intel称其为Intl64（不同于IA-64）
  - 命名为“x86-64”，有时也简称为x64

本课程主要介绍IA-32，最后简要介绍x64

回顾：计算机系统核心层之间的关联



ABI是为运行在特定ISA及特定操作系统之上的应用程序规定的一种机器级目标代码层接口

描述了应用程序和操作系统之间、应用程序和所调用的库之间、不同组成部分（如过程或函数）之间在较低层次上的机器级代码接口。

同一个C语言源程序，使用遵循不同ABI规范的编译器进行编译，其执行结果可能不一样。程序员将程序移植到另一个系统时，一定要仔细阅读目标系统的ABI规范。

后端根据ISA规范和应用程序二进制接口（Application Binary Interface，ABI）规范进行设计实现。

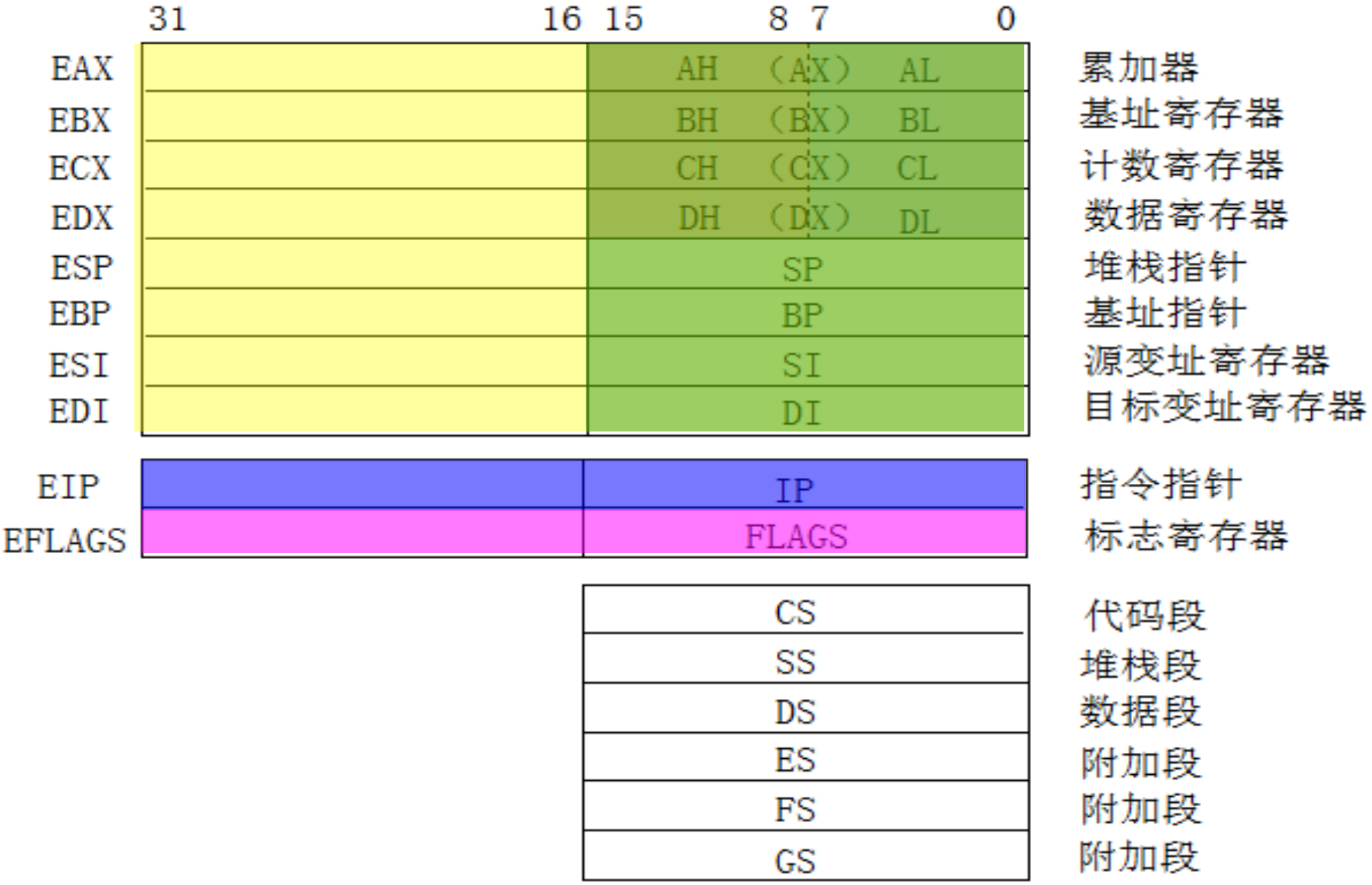
本课程所用平台为IA-32/x86-64+Linux+GCC+C语言，Linux操作系统下一般使用system V ABI

I386 System V ABI规定的数据类型

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

IA-32的定点寄存器组织

IA-32的寄存器组织


31
16
15
8
7
0
EAX
AH
(AX)
AL
累加器
EBX
BH
(BX)
BL
基址寄存器
ECX
CH
(CX)
CL
计数寄存器
EDX
DH
(DX)
DL
数据寄存器
ESP
SP
堆栈指针
EBP
BP
基址指针
ESI
SI
源变址寄存器
EDI
DI
目标变址寄存器
EIP
IP
指令指针
EFLAGS
FLAGS
标志寄存器
CS
代码段
SS
堆栈段
DS
数据段
ES
附加段
FS
附加段
GS
附加段

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

如果要用C语言来模拟IA-32的寄存器组织，该如何做?

```
typedef struct{
union{
    struct {        uint32_t  eax;
                    uint32_t  ecx;
                    uint32_t  edx;
                    uint32_t  ebx;
                    uint32_t  esp;
                    uint32_t  ebp;
                    uint32_t  esi;
                    uint32_t  edi; };
    union {
                    uint32_t  _32;
                    uint16_t  _16;
                    uint8_t   _8[2];
          } gpr[8];
   };
swaddr_t  eip;
} CPU_state;

extern CPU_state cpu;
enum { R_EAX, R_ECX, R_EDX, R_EBX, R_ESP, R_EBP, R_ESI, R_EDI };
enum { R_AX, R_CX, R_DX, R_BX, R_SP, R_BP, R_SI, R_DI };
enum { R_AL, R_CL, R_DL, R_BL, R_AH, R_CH, R_DH, R_BH };
```

PA中模拟的IA-32的寄存器组织

IA-32的标志寄存器

31-22	21	20	19	18	17	16	15	14	13 12	11	10	9	8	7	6	5	4	3	2	1	0
保留	ID	VIP	VIF	AC	VM	RF	0	NT	IOPL	O	D	I	T	S	Z	0	A	0	P	1	C

8086

80286/386

- 6个条件标志
  - OF、SF、ZF、CF各是什么标志（条件码）？
  - AF：辅助进位标志（BCD码运算时才有意义）
  - PF：奇偶标志
- 3个控制标志
  - DF（Direction Flag）：方向标志（自动变址方向是增还是减）
  - IF（Interrupt Flag）：中断允许标志 （仅对外部可屏蔽中断有用）
  - TF（Trap Flag）：陷阱标志（是否是单步跟踪状态）
- ……

IA-32的寻址方式

- 寻址方式
  - 根据指令给定信息得到操作数或操作数地址
- 操作数所在的位置
  - 指令中：立即寻址
  - 寄存器中：寄存器寻址
  - 存储单元中（属于存储器操作数，按字节编址）：其他寻址方式
- 存储器操作数的寻址方式与微处理器的工作模式有关
  - 两种工作模式：实地址模式和保护模式
- 实地址模式（基本用不到）
  - 为与8086/8088兼容而设，加电或复位时
  - 寻址空间为1MB，20位地址：(CS)<<4+(IP)
- 保护模式（需要掌握）
  - 加电后进入，采用虚拟存储管理，多任务情况下隔离、保护
  - 80286以上高档微处理器最常用的工作模式
  - 寻址空间为2^{32}B，32位地址分段（段基址+段内偏移量）

保护模式下的寻址方式

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

存储器操作数的寻址方式

```
int x;
float a[100];
short b[4][4];
char c;
double d[10];
```

a[i]的地址如何计算?

104+i×4

i=99时，104+99×4=500

b[i][j]的地址如何计算?

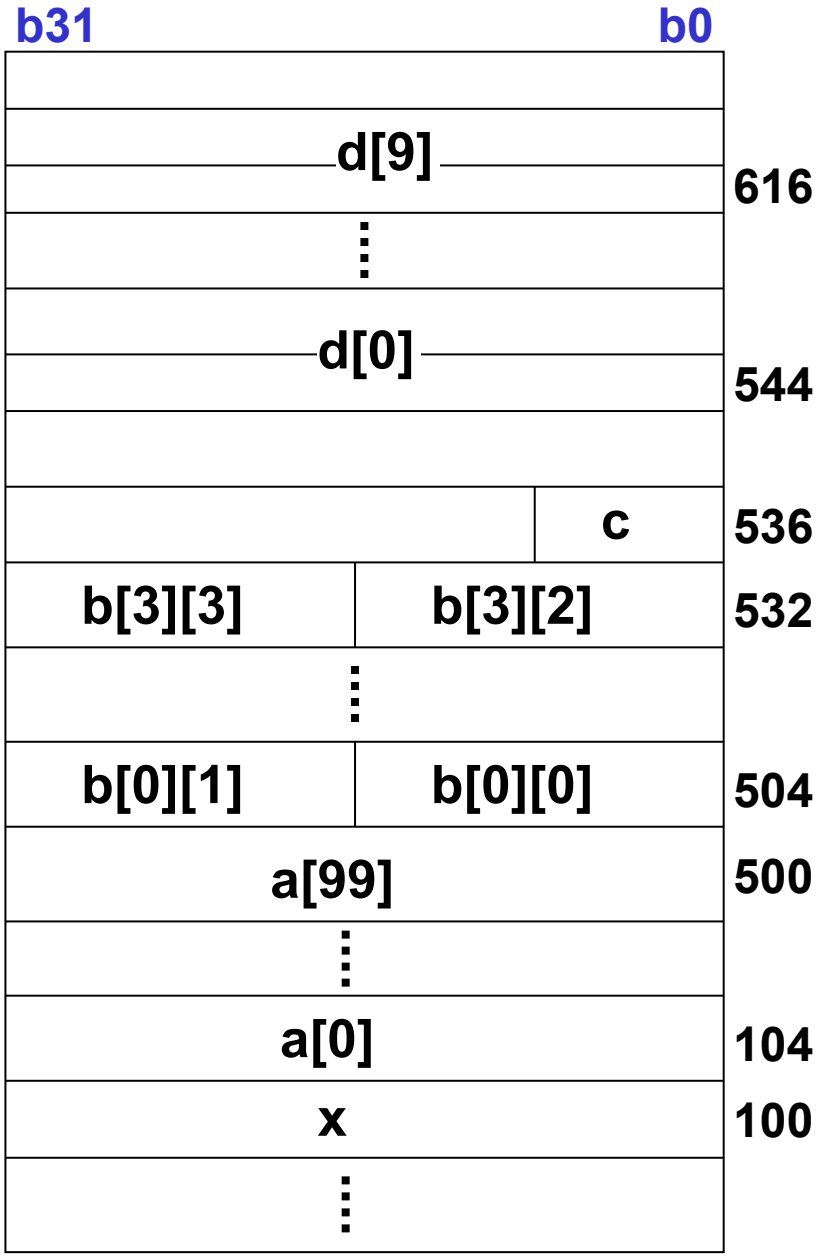
504+i×8+j×2

i=3、j=2时，504+24+4=532

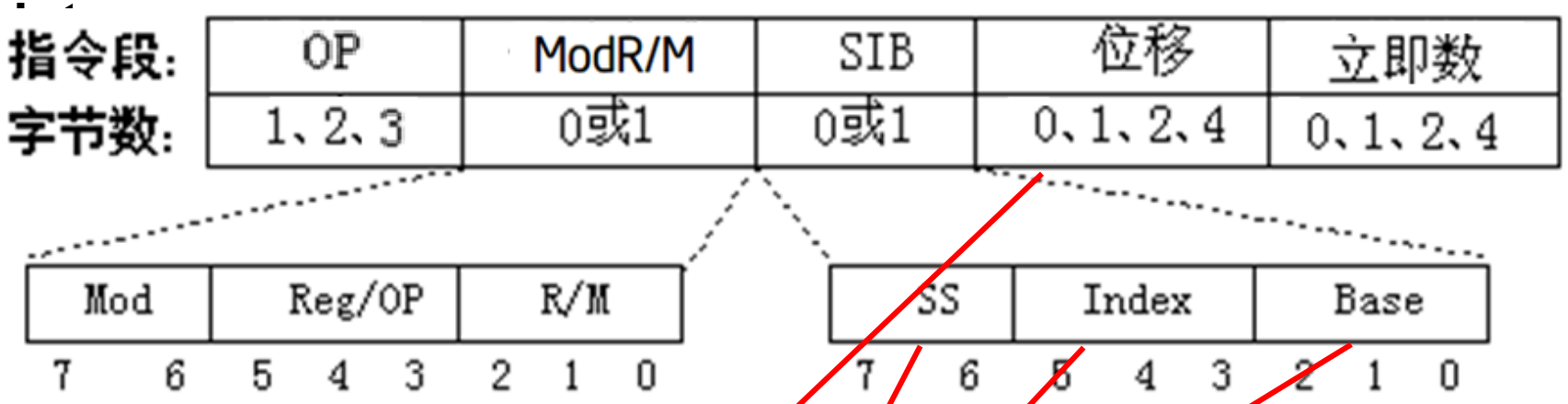
d[i]的地址如何计算?

544+i×8

i=9时，544+9×8=616



存储器操作数的寻址方式



x、c：位移 / 基址

a[i]：104+i×4，比例变址+位移

d[i]：544+i×8，比例变址+位移

b[i][j]：504+i×8+j×2，

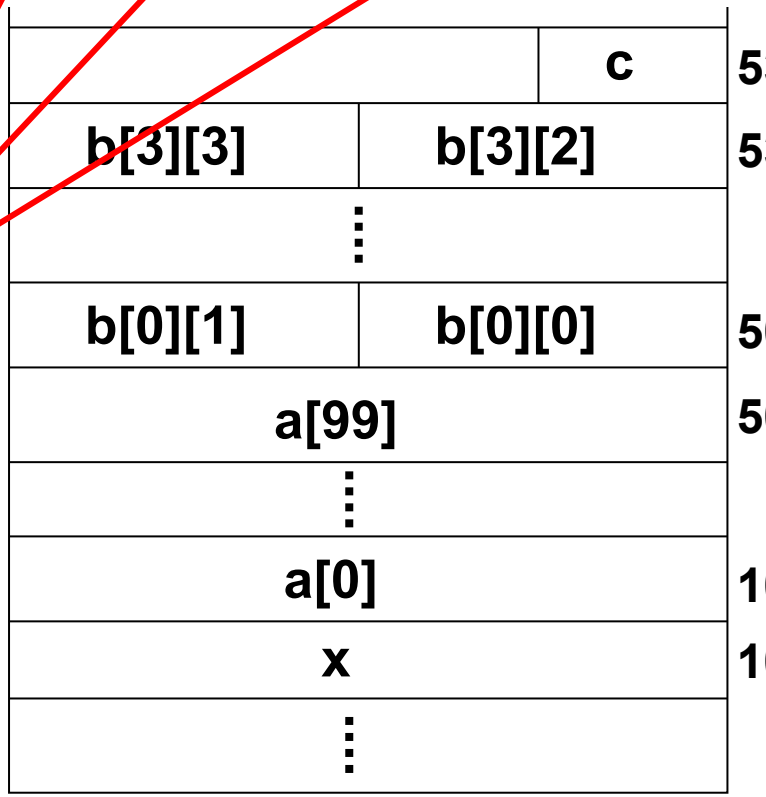
基址+比例变址+位移

将b[i][j]取到AX中的指令可以是：

“movw 504(%ebp,%esi,2), %ax”

其中，i×8在EBP中，j在ESI中，

2为比例因子



IA-32常用指令类型

（1）传送指令

- 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

- 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

- 输入输出指令

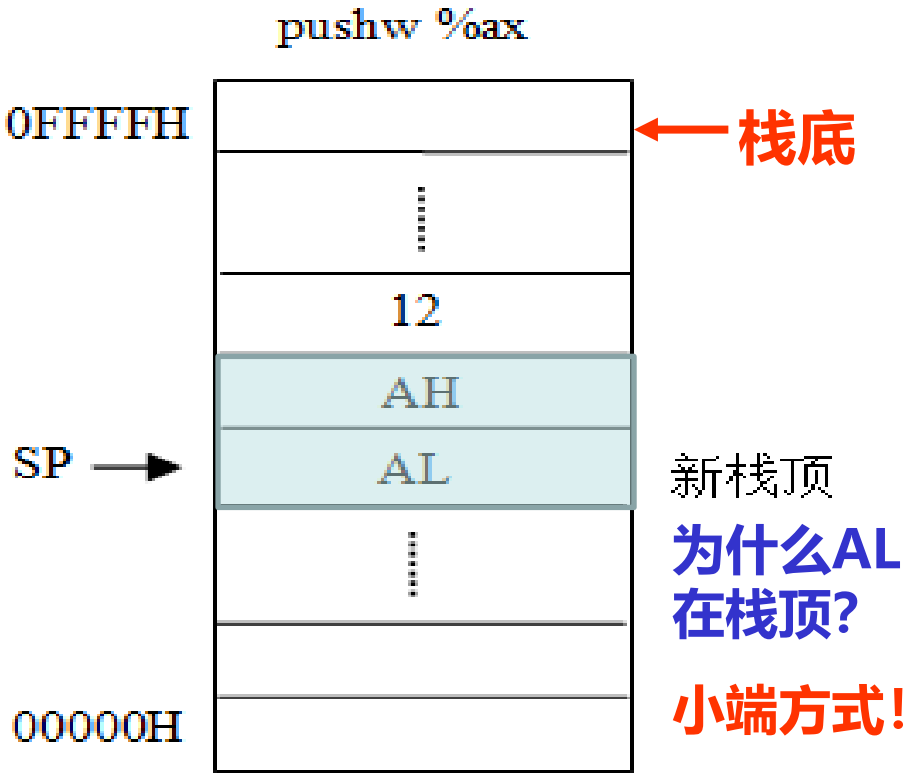
  IN和OUT：I/O端口与寄存器之间的交换

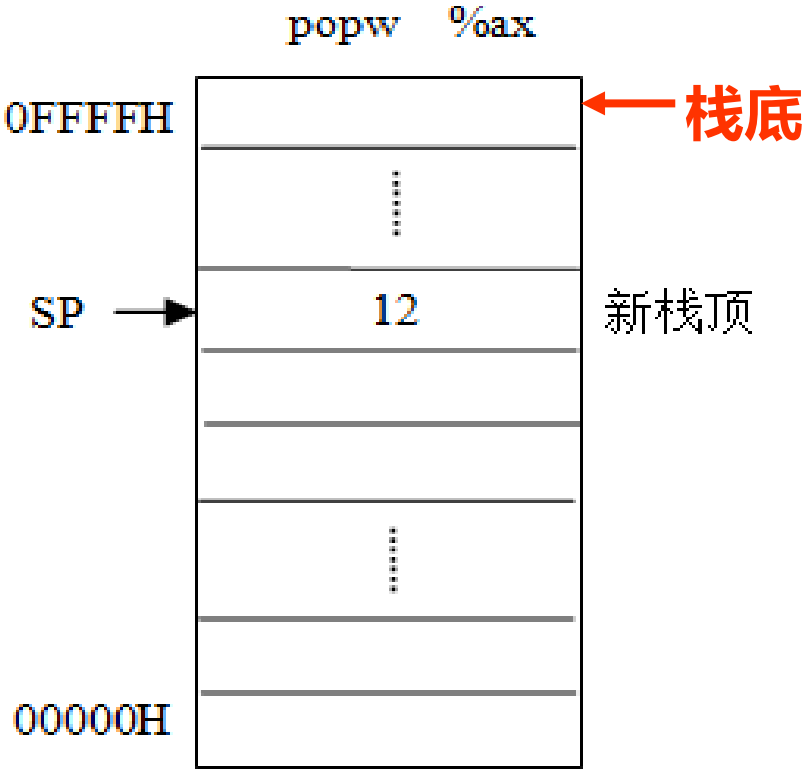
- 标志传送指令

  PUSHF、POPF：将EFLAG压栈，或将栈顶内容送EFLAG

“入栈”和“出栈”操作

- 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
- “栈”不等于“堆栈”（由“堆”和“栈”组成）





R[sp]←R[sp]-2、M[R[sp]]←R[ax]

R[ax]←M[R[sp]]、[sp]←R[sp]+2

传送指令举例

将以下Intel格式指令转换为AT&T格式指令，并说明功能。

```
push    ebp
mov     ebp, esp
mov     edx, DWORD PTR [ebp+8]
mov     bl, 255
mov     ax, WORD PTR [ebp+edx*4+8]
mov     WORD PTR [ebp+20], dx
lea     eax, [ecx+edx*4+8]
```

```
pushl   %ebp                    //R[esp]←R[esp]-4，M[R[esp]] ←R[ebp]，双字
movl    %esp, %ebp              //R[ebp] ←R[esp]，双字
movl    8(%ebp), %edx           //R[edx] ←M[R[ebp]+8]，双字
movb    $255, %bl               //R[bl]←255，字节
movw    8(%ebp,%edx,4), %ax     //R[ax]←M[R[ebp]+R[edx]×4+8]，字
movw    %dx, 20(%ebp)           //M[R[ebp]+20]←R[dx]，字
leal    8(%ecx,%edx,4), %eax    //R[eax]←R[ecx]+R[edx]×4+8，双字
```

IA-32常用指令类型

（2）定点算术运算指令

- 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
- 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
- 取负运算（影响标志、若对0取负，则结果为0且CF=0, 否则CF=1）
  - NEG：取负，包括negb、negw、negl等
- 比较运算（做减法得到标志、不区分无/带符号）
  - CMP：比较，包括cmpb、cmpw、cmpl等
- 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

整数乘除指令

- 乘法指令：可给出一个、两个或三个操作数
  - mul、imul：若给出一个操作数SRC，则另一个源操作数隐含在AL/AX/EAX中，将SRC和累加器内容相乘，结果存放在AX（16位）或DX-AX（32位）或EDX-EAX（64位）中。DX-AX表示32位乘积的高、低16位分别在DX和AX中。 n位× n位=2n位
  - imul：若指令中给出两个操作数DST和SRC，则将DST和SRC相乘，结果在DST中。n位× n位=n位
  - imul：若指令中给出三个操作数REG、SRC和IMM，则将SRC和立即数IMM相乘，结果在REG中。n位× n位=n位
- 除法指令：只明显指出除数
  - 若为8位，则16位被除数在AX寄存器中，商送回AL，余数在AH
  - 若为16位，则32位被除数在DX-AX寄存器中，商送回AX，余数在DX
  - 若为32位，则被除数在EDX-EAX寄存器中，商送EAX，余数在EDX

以上内容不要死记硬背，遇到具体指令时能查阅到并理解即可。

定点算术运算指令汇总

指令	显式操作数	影响的常用标志	操作数类型	AT&T 指令助记符	对应 C 运算符
ADD	2 个	OF、ZF、SF、CF	无/带符号整数	addb、addw、addl	+
SUB	2 个	OF、ZF、SF、CF	无/带符号整数	subb、subw、subl	−
INC	1 个	OF、ZF、SF	无/带符号整数	incb、incw、incl	++
DEC	1 个	OF、ZF、SF	无/带符号整数	decb、decw、decl	−−
NEG	1 个	OF、ZF、SF、CF	无/带符号整数	negb、negw、negl	−
CMP	2 个	OF、ZF、SF、CF	无/带符号整数	cmpb、cmpw、cmpl	<, <=, >, >=
MUL	1 个	OF、CF	无符号整数	mulb、mulw、mull	*
IMUL	1 个	OF、CF	带符号整数	imulb、imulw、imull	*
IMUL	2 个	OF、CF	带符号整数	imulb、imulw、imull	*
IMUL	3 个	OF、CF	带符号整数	imulb、imulw、imull	*
DIV	1 个	无	无符号整数	divb、divw、divl	/, %
IDIV	1 个	无	带符号整数	idivb、idivw、idivl	/, %

定点加法指令举例

- 假设R[ax]=FFFAH，R[bx]=FFF0H，则执行以下指令后

“addw %bx, %ax”

AX、BX中的内容各是什么？标志CF、OF、ZF、SF各是什么？要求分别将操作数作为无符号数和带符号整数解释并验证指令执行结果。

解：功能：R[ax]←R[ax]+R[bx]，指令执行后的结果如下

R[ax]=FFFAH+FFF0H=FFEAH ，BX中内容不变

CF=1，OF=0，ZF=0，SF=1

若是无符号整数运算，则CF=1说明结果溢出

验证：FFFA的真值为65535-5=65530，FFF0的真值为65520

FFEA的真值为65535-21=65514≠65530+65520，即溢出

若是带符号整数运算，则OF=0说明结果没有溢出

验证：FFFA的真值为-6，FFF0的真值为-16

FFEA的真值为-22=-6+(-16)，结果正确，无溢出

定点乘法指令举例

- 假设R[eax]=000000B4H，R[ebx]=00000011H，M[000000F8H]=000000A0H，请问：

  (1) 执行指令“mulb %bl”后，哪些寄存器的内容会发生变化？是否与执行“imulb %bl”指令所发生的变化一样？为什么？请用该例给出的数据验证你的结论。

解：“mulb %bl”功能为 R[ax]←R[al]×R[bl]，执行结果如下

R[ax]=B4H × 11H（无符号整数180和17相乘）

R[ax]=0BF4H，真值为3060=180 × 17

无符号乘：

```
      1011 0100
   x  0001 0001
-----------------
      1011 0100
 1011 0100
-----------------
0000 1011 1111 0100
```

AH=?　　AL=?

“imulb %bl”功能为 R[ax]←R[al]×R[bl]

R[ax]=B4H × 11H（带符号整数-76和17相乘）

R[ax]=0BF4H? 则真值为3060≠-76 × 17

R[al]=F4H, R[ah]=? AH中的值不一样！

R[ax]=FAF4H, 真值为-1292=-76 × 17

对于带符号乘，若积只取低n位，则和无符号相同；若取2n位，则采用“布斯”乘法

定点乘法指令举例

- **布斯乘法：** “imulb %bl”

R[ax]=B4H × 11H=FFB4H+FB40H=FAF4H

```
                  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 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

定点乘法指令举例

- 假设R[eax]=000000B4H，R[ebx]=00000011H，M[000000F8H]=000000A0H，请问：

(2) 执行指令“imull $-16, (%eax,%ebx,4), %eax”后哪些寄存器和存储单元发生了变化？乘积的机器数和真值各是多少？

解：“imull -16, (%eax,%ebx,4),%eax”

功能为 R[eax]←(-16)×M[R[eax]+R[ebx]×4] ，执行结果如下

R[eax]+R[ebx]×4=000000B4H+00000011H<<2=000000F8H

R[eax]=(-16)×M[000000F8H]

=(-16)× 000000A0H（带符号整数乘）

=FFFFFF60H<<4

=FFFFF600H

EAX中的真值为-2560

SKIP

整数乘除指令

- 乘法指令：可给出一个、两个或三个操作数 BACK
  - 若给出一个操作数SRC，则另一个源操作数隐含在AL/AX/EAX中，将SRC和累加器内容相乘，结果存放在AX（16位）或DX-AX（32位）或EDX-EAX（64位）中。DX-AX表示32位乘积的高、低16位分别在DX和AX中。
  - 若指令中给出两个操作数DST和SRC，则将DST和SRC相乘，结果在DST中。
  BACK
  - 若指令中给出三个操作数REG、SRC和IMM，则将SRC和立即数IMM相乘，结果在REG中。
- 除法指令：只明显指出除数，用EDX-EAX中内容除以指定的除数
  - 若为8位，则16位被除数在AX寄存器中，商送回AL，余数在AH
  - 若为16位，则32位被除数在DX-AX寄存器中，商送回AX，余数在DX
  - 若为32位，则被除数在EDX-EAX寄存器中，商送EAX，余数在EDX

以上内容不要死记硬背，遇到具体指令时能查阅到并理解即可。

IA-32常用指令类型

（3）按位运算指令

- 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
  - TEST：做“与”操作测试，仅影响标志
- 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

以上内容不要死记硬背，遇到具体指令时能查阅到并理解即可。

按位运算指令举例

假设short型变量x被编译器分配在寄存器AX中，R[ax]=FF80H，则以下汇编代码段执行后变量x的机器数和真值分别是多少?

```
movw %ax, %dx
salw  $2, %ax      1111 1111 1000 0000<<2
addl  %dx, %ax     1111 1111 1000 0000+1111 1110 0000 0000
sarw  $1, %ax      1111 1101 1000 0000>>1=1111 1110 1100 0000
```

解：$2和$1分别表示立即数2和1 。

x是short型变量，故都是算术移位指令，并进行带符号整数加。

假设上述代码段执行前R[ax]=x，则执行((x<<2)+x)>>1后，R[ax]=5x/2。算术左移时，AX中的内容在移位前、后符号未发生变化，故OF=0，没有溢出。最终AX的内容为FEC0H，解释为short型整数时，其值为-320。验证：x=-128，5x/2=-320。经验证，结果正确。

逆向工程：从汇编指令推断出高级语言程序代码

移位指令举例

```c
#include <stdio.h>
void main()
{
    int a = 0x80000000;
    unsigned int b = 0x80000000;
    printf("a=   0x%X\n", a >>1);
    printf("b=   0x%X\n", b >>1);
}
```

```
                                    push   %ebp
                                    mov    %esp,%ebp
                                    and    $0xfffffff0,%esp
                                    sub    $0x20,%esp
                                    movl   $0x80000000,0x1c(%esp)
                                    movl   $0x80000000,0x18(%esp)
19:   8b 44 24 1c                   mov    0x1c(%esp),%eax
1d:   d1 f8                         sar    %eax
1f:   89 44 24 04                   mov    %eax,0x4(%esp)
23:   c7 04 24 00 00 00 00          movl   $0x0,(%esp)
2a:   e8 fc ff ff ff                call   2b <main+0x2b>
2f:   8b 44 24 18                   mov    0x18(%esp),%eax
33:   d1 e8                         shr    %eax
35:   89 44 24 04                   mov    %eax,0x4(%esp)
39:   c7 04 24 0b 00 00 00          movl   $0xb,(%esp)
40:   e8 fc ff ff ff                call   41 <main+0x41>
45:   c9                            leave
46:   c3                            ret
```

算术

逻辑

IA-32常用指令类型

（4）控制转移指令

指令执行可按顺序 或 跳转到转移目标指令处执行

- 无条件转移指令

  JMP DST：无条件转移到目标指令DST处执行

- 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

- 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 ）

- 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

- 中断指令 （详见第7、8章）

以上内容不要死记硬背，遇到具体指令时能查阅到并理解即可。

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

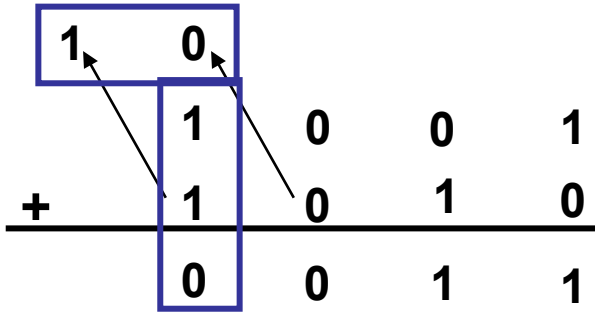
(3)按带符号整数比较转移

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

标志信息是干什么的？

Ex1: -7- 6 = -7 + (-6) = +3

9 - 6 = 3

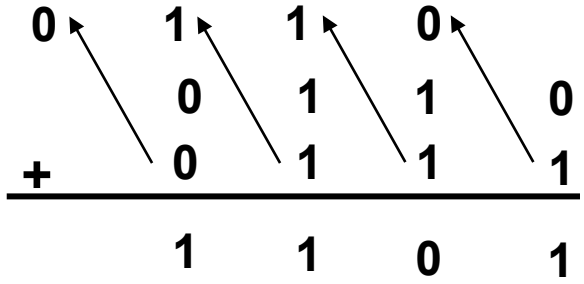


OF=1、ZF=0

SF=0、借位CF=0

6 - (-7) = 6 + 7 = -3

6 - 9 = 13



OF=1、ZF=0

SF=1、借位CF=1

做减法以比较大小，规则：

Unsigned: CF=0时，大于

Signed：OF=SF时，大于

例子：C表达式类型转换顺序

unsigned long long

↑

long long

↑

unsigned

↑

int

↑

(unsigned)char,short

```
#include <stdio.h>
void main()
{
        unsigned int a = 1;
        unsigned short b = 1;
        char c = -1;
        int d;

        d = (a > c) ? 1:0;
        printf("%d\n",d);
        d = (b > c) ? 1:0;
        printf("%d\n",d);
}
```

猜测：各用哪种条件设置指令?

条件设置指令SETcc DST:

将条件码cc保存到DST（通常是一个8位寄存器）

```
0804841c <main>:
 804841c:        55                              push   %ebp
 804841d:        89 e5                           mov    %esp,%ebp
 804841f:        83 e4 f0                        and    $0xfffffff0,%esp
 8048422:        83 ec 20                        sub    $0x20,%esp
 8048425:        c7 44 24 1c 01 00 00 00         movl   $0x1,0x1c(%esp)
 804842d:        66 c7 44 24 1a 01 00            movw   $0x1,0x1a(%esp)
 8048434:        c6 44 24 19 ff                  movb   $0xff,0x19(%esp)
 8048439:        0f be 44 24 19                  movsbl 0x19(%esp),%eax
 804843e:        3b 44 24 1c                     cmp    0x1c(%esp),%eax
 8048442:        0f 92 c0                        setb   %al
 8048445:        0f b6 c0                        movzbl %al,%eax
 8048448:        89 44 24 14                     mov    %eax,0x14(%esp)
 804844c:        8b 44 24 14                     mov    0x14(%esp),%eax
 8048450:        89 44 24 04                     mov    %eax,0x4(%esp)
 8048454:        c7 04 24 20 85 04 08            movl   $0x8048520,(%esp)
 804845b:        e8 a0 fe ff ff                  call   8048300 <printf@plt>
 8048460:        0f b7 54 24 1a                  movzwl 0x1a(%esp),%edx
 8048465:        0f be 44 24 19                  movsbl 0x19(%esp),%eax
 804846a:        39 c2                           cmp    %eax,%edx
 804846c:        0f 9f c0                        setg   %al
 804846f:        0f b6 c0                        movzbl %al,%eax
 8048472:        89 44 24 14                     mov    %eax,0x14(%esp)
 8048476:        8b 44 24 14                     mov    0x14(%esp),%eax
 804847a:        89 44 24 04                     mov    %eax,0x4(%esp)
 804847e:        c7 04 24 20 85 04 08            movl   $0x8048520,(%esp)
 8048485:        e8 76 fe ff ff                  call   8048300 <printf@plt>
 804848a:        c9                              leave
 804848b:        c3                              ret
```

unsigned int a=1;

unsigned short b=1;

char c=-1;

d=(a>c)?1:0

无符号

d=(b>c)?1:0

带符号

例子：程序的机器级表示与执行

```
int sum(int a[ ], unsigned len)
{
   int  i，sum = 0;
   for (i = 0; i <= len–1; i++)
        sum += a[i];
   return sum;
}
```

```
sum:
    …
.L3:
    …
   movl  -4(%ebp),  %eax
   movl  12(%ebp),  %edx
   subl   $1,  %edx
   cmpl  %edx,  %eax
   jbe    .L3
    …
```

当参数len为0时，返回值应该是0，但是在机器上执行时，却发生了存储器访问异常。 Why?

i 和 len 分别存放在哪个寄存器中？ %eax？ %edx?

i 在%eax中，len在%edx中

%eax: 0000 …… 0000

%edx: 0000 …… 0000

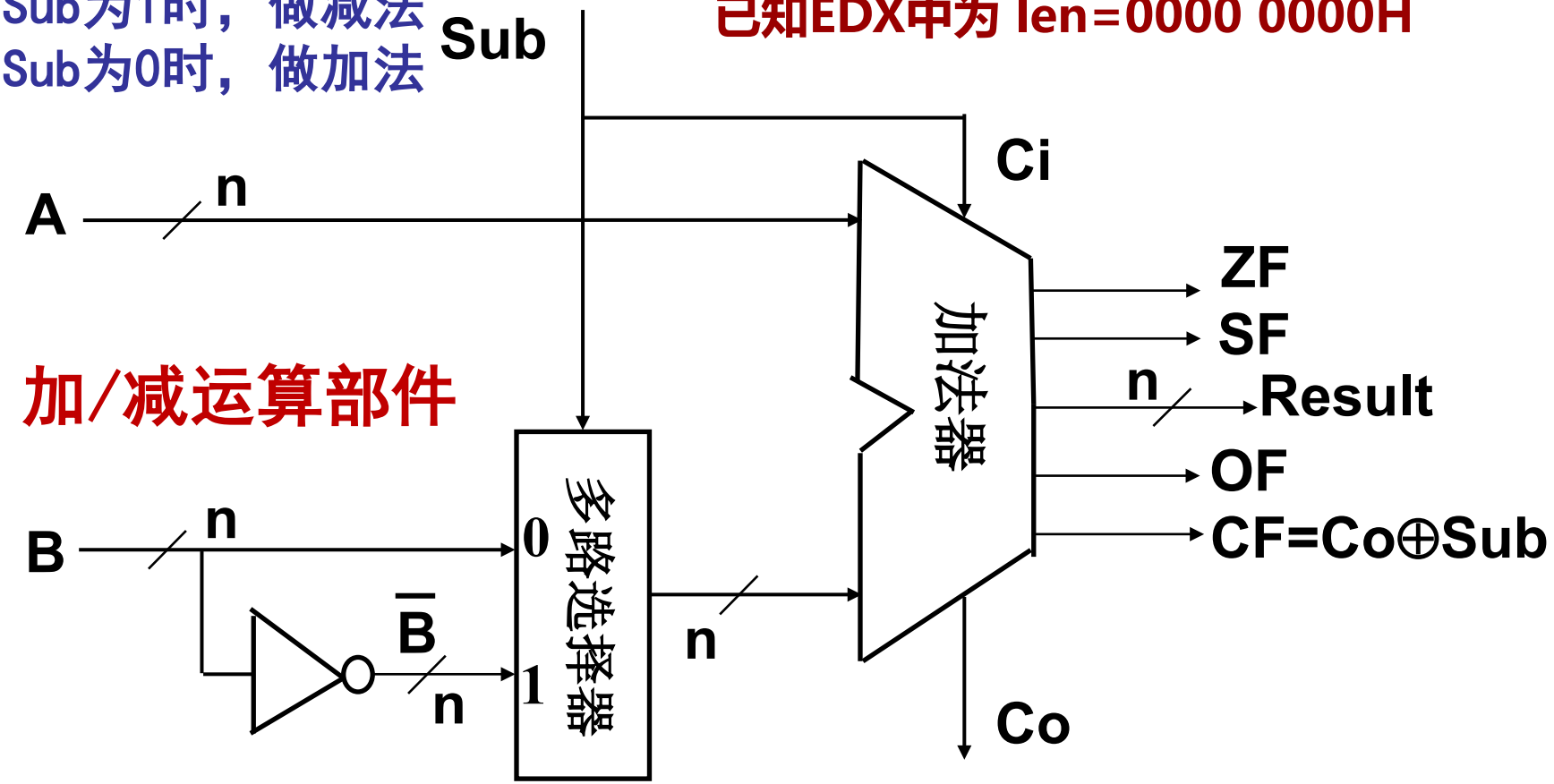
subl 指令的执行结果是什么?

cmpl 指令的执行结果是什么?

subl $1, %edx指令的执行结果

当Sub为1时，做减法
当Sub为0时，做加法

已知EDX中为 len=0000 0000H

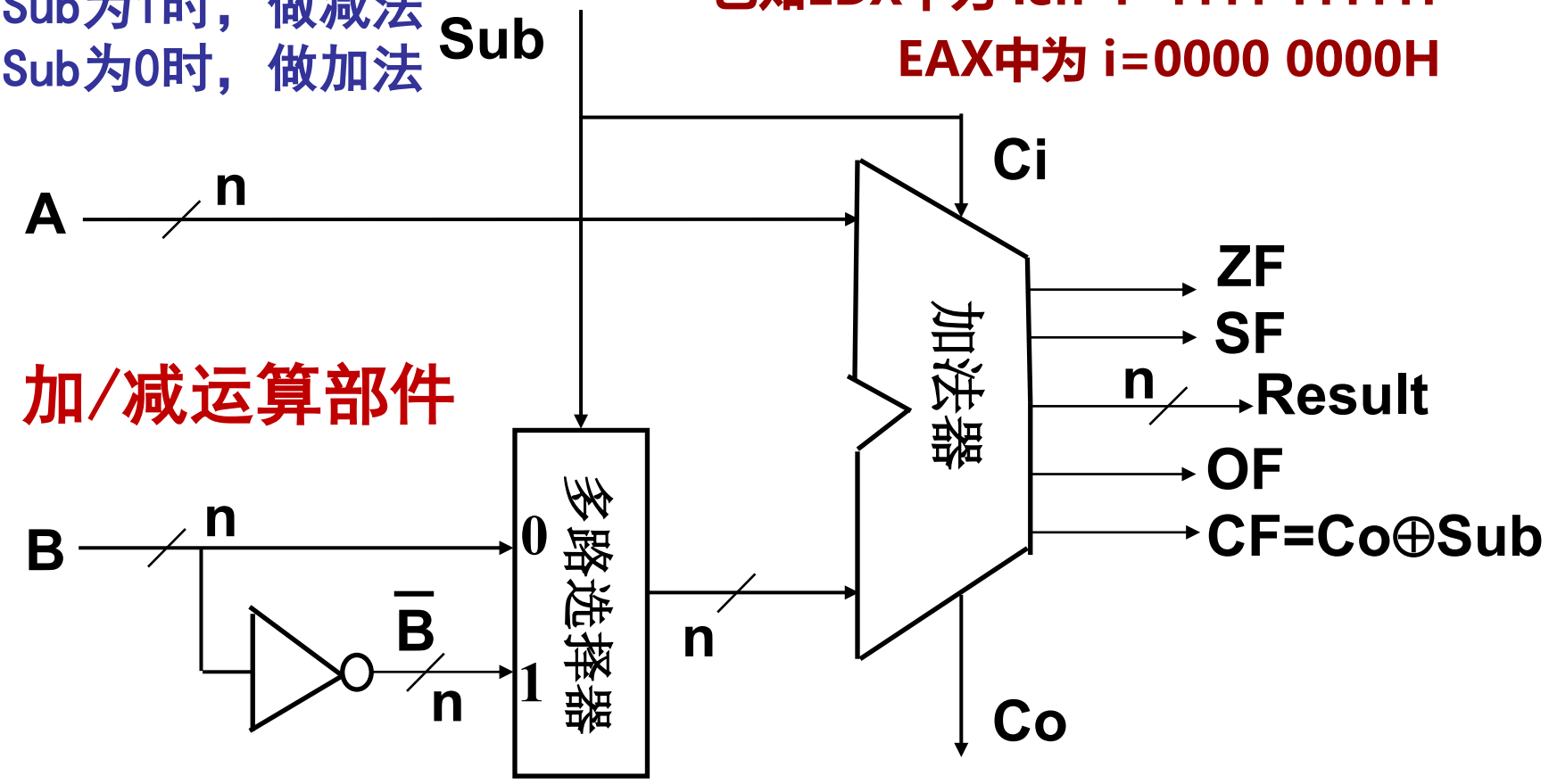


“subl $1, %edx”执行时：A=0000 0000H，B为0000 0001H，Sub=1，因此Result是32个1。

cpml %edx,%eax指令的执行结果

当Sub为1时，做减法
当Sub为0时，做加法

已知EDX中为 len-1=FFFF FFFFH
EAX中为 i=0000 0000H



“cmpl %edx,%eax”执行时：A=0000 0000H，B为FFFF FFFFH，Sub=1，因此Result是0…01, CF=1, ZF=0, OF=0, SF=0

jbe .L3指令的执行结果

指令	转移条件	说明
JA/JNBE label	CF=0 AND ZF=0	无符号数A > B
JAE/JNB label	CF=0 OR ZF=1	无符号数A≥B
JB/JNAE label	CF=1 AND ZF=0	无符号数A < B
JBE/JNA label	CF=1 OR ZF=1	无符号数A≤B
JG/JNLE label	SF=OF AND ZF=0	有符号数A > B
JGE/JNL label	SF=OF OR ZF=1	有符号数A≥B
JL/JNGE label	SF≠OF AND ZF=0	有符号数A < B
JLE/JNG label	SF≠OF OR ZF=1	有符号数A≤B

“cmpl %edx,%eax”执行结果是 CF=1, ZF=0, OF=0, SF=0，说明满足条件，应转移到.L3执行！ 显然，对于每个 i 都满足条件，因为任何无符号数都比32个1小，因此循环体被不断执行，最终导致数组访问越界而发生存储器访问异常。

例子：程序的机器级表示与执行

例：

```
int sum(int a[ ], int len)
{
   int  i，sum = 0;
   for (i = 0; i <= len–1; i++)
        sum += a[i];
   return sum;
}
```

正确的做法是将参数len声明为int型。 Why?

i 和 len 分别存放在哪个寄存器中？ %eax？ %edx?

```
sum:
    …
.L3:
    …
    movl  -4(%ebp),  %eax
    movl  12(%ebp),  %edx
    subl    $1,  %edx
    cmpl  %edx,  %eax
    jle      .L3
    …
```

i 在%eax中，len在%edx中

%eax: 0000 …… 0000

%edx: 0000 …… 0000

subl 指令的执行结果是什么?

cmpl 指令的执行结果是什么?

jle .L3指令的执行结果

指令	转移条件	说明
JA/JNBE label	CF=0 AND ZF=0	无符号数A > B
JAE/JNB label	CF=0 OR ZF=1	无符号数A≥B
JB/JNAE label	CF=1 AND ZF=0	无符号数A < B
JBE/JNA label	CF=1 OR ZF=1	无符号数A≤B
JG/JNLE label	SF=OF AND ZF=0	有符号数A > B
JGE/JNL label	SF=OF OR ZF=1	有符号数A≥B
JL/JNGE label	SF≠OF AND ZF=0	有符号数A < B
JLE/JNG label	SF≠OF OR ZF=1	有符号数A≤B

“cmpl %edx,%eax”执行结果是 CF=1, ZF=0, OF=0, SF=0，

说明不满足条件，应跳出循环执行，执行结果正常。

X87浮点指令、MMX和SSE指令

- IA-32的浮点处理架构有两种：
  - 浮点协处理器x87架构（x87 FPU）
  - ✓ 8个80位寄存器ST(0) ~ ST(7)（采用栈结构），栈顶为ST(0)
  - 由MMX发展而来的SSE架构
  - ✓ MMX指令使用8个64位寄存器MM0~MM7，借用8个80位寄存器ST(0)~ST(7)中64位尾数所占的位，可同时处理8个字节，或4个字，或2个双字，或一个64位的数据
  - ✓ MMX指令并没带来3D游戏性能的显著提升，故相继推出SSE指令集，它们都采用SIMD（单指令多数据，也称数据级并行）技术
  - ✓ SSE指令集将80位浮点寄存器扩充到128位多媒体扩展通用寄存器XMM0~XMM7，可同时处理16个字节，或8个字，或4个双字（32位整数或单精度浮点数），或两个四字的数据，而且从SSE2开始，还支持128位整数运算或同时并行处理两个64位双精度浮点数

IA-32中通用寄存器中的编号

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

反映了体系结构发展的轨迹，字长不断扩充，指令保持兼容

ST（0）~ ST（7）是80位，MM0 ~MM7使用其低64位

SSE指令（SIMD操作）

- 用简单的例子来比较普通指令与数据级并行指令的执行速度
  - ✓为使比较结果不受访存操作影响，下例中的运算操作数在寄存器中
  - ✓为使比较结果尽量准确，例中设置的循环次数较大: 0x4000000=2^{26}
  - ✓例子只是为了说明指令执行速度的快慢，并没有考虑结果是否溢出

以下是普通指令写的程序

```
080484f0 <dummy_add>:
 80484f0:  55              push  %ebp
 80484f1:  89 e5           mov  %esp, %ebp
 80484f3:  b9 00 00 00 04  mov  $0x4000000, %ecx
 80484f8:  b0 01           mov  $0x1, %al
 80484fa:   b3 00          mov   $0x0, %bl
 80484fc: 00 c3            add   %al, %bl
 80484fe: e2 fc            loop  80484fc <dummy_add+0xc>
8048500:  5d               pop            %ebp
 8048501: c3               ret
```

所用时间约为22.643816s

循环400 0000H=2^{26}次，每次只有一个数（字节）相加

SSE指令（SIMD操作）

以下是SIMD指令写的程序

所用时间约为1.411588s

```
08048510 <dummy_add_sse>:
 8048510:  55                   push  %ebp
 8048511:  b8 00 9d 04 10       mov $0x10049d00, %eax
 8048516:  89 e5                mov   %esp, %ebp
 8048518:  53                   push   %ebx
 8048519:   bb 20 9d 04 14      mov   $0x14049d20, %ebx
 804851e:  b9 00 00 40 00       mov   $0x400000, %ecx
 8048523:  66 0f 6f 00          movdqa  (%eax), %xmm0
 8048527:  66 0f 6f 0b          movdqa  (%ebx), %xmm1
804852b: 66 0f fc c8            paddb    %xmm0, %xmm1
804852f: e2 fa                  loop   804852b <dummy_add_sse+0x1b>
 8048531:  5b                   pop    %ebx
 8048532:  5d                   pop    %ebp
 8048533:  c3                   ret
```

SIDM指令

22.643816s/ 1.411588s ≈16.041378,与预期结果一致!

SIMD指令并行执行效率高!

dqa：两个对齐四字

循环400000H=2^{22}次，每次同时有128/8=16个数（字节）相加

浮点操作与SIMD指令

- 浮点操作与SIMD指令
  - IA-32的浮点处理架构有两种

    (1) x86配套的浮点协处理器x87FPU架构，80位浮点寄存器栈

    (2) 由MMX发展而来的SSE指令集架构，采用的是单指令多数据（Single Instruction Multi Data，SIMD）技术

    对于IA-32架构， gcc默认生成x87 FPU 指令集代码，如果想要生成SEE指令集代码，则需要设置适当的编译选项

  - IA-32采用80位双精度浮点数扩展格式

    1位符号位s、15位阶码e（偏置常数为16 383）、1位显式首位有效位（explicit leading significant bit）j 和 63位尾数f。它与IEEE 754单精度和双精度浮点格式的一个重要的区别是，它没有隐藏位，有效位数共64位。

X87 FPU指令

- **数据传送类**

  栈顶为ST(0) ，带P结尾指令表示操作数会出栈，即ST(1)将变成ST(0)

(1) 装入

FLD：将数据装入浮点寄存器栈顶

FILD：将数据从int型转换为浮点格式后，装入浮点寄存器栈顶

(2) 存储

FSTx：x为s/l时，将栈顶ST(0)转换为单/双精度格式，然后存入存储单元

FSTPx：弹出栈顶元素，并完成与FSTx相同的功能

FISTx：将栈顶数据从浮点格式转换为int型后，存入存储单元

FISTPx：弹出栈顶元素，并完成与FISTx相同的功能

不作要求，大概了解一下

X87 FPU指令

- **数据传送类**

(3) 交换

FXCH：交换栈顶和次栈顶两元素

(4) 常数装载到栈顶

FLD1 ： 装入常数1.0

FLDZ ： 装入常数0.0

FLDPI ： 装入常数pi (=3.1415926...)

FLDL2E ： 装入常数log(2)e

FLDL2T ： 装入常数log(2)10

FLDLG2 ： 装入常数log(10)2

FLDLN2 ： 装入常数Log(e)2

X87 FPU指令

- **算术运算类**

(1) 加法

FADD/FADDP： 相加 / 相加后弹出

FIADD：按int型相加

(2) 减法

FSUB/FSUBP ： 相减 / 相减后弹出

FSUBR/FSUBRP：调换次序相减 / 相减后弹出

FISUB：按int型相减

FISUBR：按int型相减，调换相减次序

若指令未带操作数，则默认操作数为ST(0) 、ST(1)

带R后缀指令是指操作数顺序变反，例如：

fsub执行的是x-y，fsubr执行的就是y-x

X87 FPU指令

- **算术运算类**

(3) 乘法

FMUL/FMULP: 相乘/相乘后出栈

FIMUL：按int型相乘

(4) 除法

FDIV/FDIVP : 相除/相除后出栈

FIDIV：按int型相除

FDIVR/FDIVRP

FIDIVR

例如： fidivl 0x8(%ebp)将指定存储单元中的操作数M[R[ebp]+8]按int型数的值转换为double型，再将ST(0)除以该数，并将结果存入ST(0)中

IA-32浮点操作举例

问题：使用老版本gcc –O2编译时，程序一输出0，程序二输出却是1，是什么原因造成的？ **f(10)的值是多少？机器数是多少？**

程序一：

```
#include <stdio.h>
double f(int x) {
        return 1.0 / x ;
}
void main() {
        double a, b;
        int i ;
        a = f(10) ;
        b = f(10) ;
        i = a == b ;
        printf( "%d\n" , i ) ;
}
```

程序二：

```
#include <stdio.h>
double f(int x) {
        return 1.0 / x ;
}
void main() {
        double a, b, c;
        int i ;
        a = f(10) ;
        b = f(10) ;
        c = f(10) ;
        i = a == b ;
        printf( "%d\n" , i ) ;
}
```

IA-32浮点操作举例

```
double f(int x)
{
    return 1.0 / x ;
}
```

```
8048328:    55          push   %ebp
8048329:    89 e5       mov    %esp,%ebp
804832b:    d9 e8       fld1
804832d:    da 75 08    fidivl 0x8(%ebp)
8048330:    c9          leave
8048331:    c3          ret
```

两条重要指令的功能如下。

fld1：将常数1压入栈顶ST(0)

fidivl：将指定存储单元操作数M[R[ebp]+8]按int型数转换为double型，再将ST(0)除以该数，并将结果存入ST(0)中

f(10)=0.1

0.1=0.00011[0011]B= 0.00011 0011 0011 0011 0011 0011 0011...B

IA-32浮点操作举例

```
...
a = f(10) ;
b = f(10) ;
i = a == b;
...
```

```
08048334 <main>:
 8048334:      55                    push   %ebp
 8048335:      89 e5                 mov    %esp,%ebp
 8048337:      83 ec 08              sub    $0x8,%esp
 804833a:      83 e4 f0              and    $0xfffffff0,%esp
 804833d:      83 ec 0c              sub    $0xc,%esp
 8048340:      6a 0a                 push   $0xa
 8048342:      e8 e1 ff ff ff        call   8048328 <f> //计算a=f(10)
 8048347:      dd 5d f8              fstpl  0xfffffff8(%ebp) //a存入内存
 804834a:      c7 04 24 0a 00 00 00    movl   $0xa,(%esp)
 8048351:      e8 d2 ff ff ff        call   8048328 <f> //计算b=f(10)
 8048356:      dd 45 f8              fldl   0xfffffff8(%ebp) //a入栈顶
 8048359:      58                    pop    %eax
 804835a:      da e9                 fucompp      //比较ST(0)a和ST(1)b
 804835c:      df e0                 fnstsw %ax  //把FPU状态字送到AX
 804835e:      80 e4 45              and    $0x45,%ah
 8048361:      80 fc 40              cmp    $0x40,%ah
 8048364:      0f 94 c0              sete   %al
 8048367:      5a                    pop    %edx
 8048368:      0f b6 c0              movzbl %al,%eax
 804836b:      50                    push   %eax
 804836c:      68 d8 83 04 08        push   $0x80483d8
 8048371:      e8 f2 fe ff ff        call   8048268 <_init+0x38>
 8048376:      c9                    leave
 8048377:      c3                    ret
```

80位→64位

64位→80位

0.1是无限循环小数，无法精确表示，比较时，a舍入过而b没有舍入过，故 a≠b

IA-32浮点操作举例

```
...
a = f(10) ;
b = f(10) ;
c = f(10) ;
i = a == b;
...
```

```
8048342:    e8 e1 ff ff ff     call   8048328 <f> //计算a
8048347:    dd 5d f8           fstpl  0xfffffff8(%ebp) //把a存回内存
                                      //a产生精度损失
804834a:    c7 04 24 0a 00 00 00    movl   $0xa,(%esp,1)
8048351:    e8 d2 ff ff ff     call   8048328 <f> //计算b
8048356:    dd 5d f0           fstpl  0xfffffff0(%ebp) //把b存回内存
                                      //b产生精度损失
8048359:    c7 04 24 0a 00 00 00    movl   $0xa,(%esp,1)
8048360:    e8 c3 ff ff ff     call   8048328 <f> //计算c
8048365:    dd d8              fstp   %st(0)
8048367:    dd 45 f8           fldl   0xfffffff8(%ebp) //从内存中载入a
804836a:    dd 45 f0           fldl   0xfffffff0(%ebp) //从内存中载入b
804836d:    d9 c9              fxch   %st(1)
804836f:    58                 pop    %eax
8048370:    da e9              fucompp //比较a , b
8048372:    df e0              fnstsw %ax
```

0.1是无限循环小数，因而无法精确表示，比较时，a和b都是舍入过的，故 a=b!

IA-32浮点操作举例

- 从这个例子可以看出
  - 编译器的设计和硬件结构紧密相关。
  - 对于编译器设计者来说，只有真正了解底层硬件结构和真正理解指令集体系结构，才能够翻译出没有错误的目标代码，并为程序员完全屏蔽掉硬件实现的细节，方便应用程序员开发出可靠的程序。
  - 对于应用程序开发者来说，也只有真正了解底层硬件的结构，才有能力编制出高效的程序，能够快速定位出错的地方，并对程序的行为作出正确的判断。

第一、二讲总结

- 高级语言程序总是转换为机器代码才能在机器上执行
- 转换过程：预处理、编译、汇编、链接
- 机器代码是二进制代码，可DUMP为汇编代码表示
- ISA规定了一台机器的指令系统涉及到的所有方面，例如：
  - 所有指令的指令格式、功能
  - 通用寄存器的个数、位数、编号和功能
  - 存储地址空间大小、编址方式、大/小端
  - 指令寻址方式
- IA-32是典型的CISC（复杂指令集计算机）风格ISA
  - Intel格式汇编、AT&T格式汇编（本课程使用）
  - 指令类型（传送、算术、位操作、控制、浮点、…）
  - 寻址方式
    - 立即、寄存器、存储器（SR:[B]+[I]*s+A）

程序的机器级表示

- 分以下五个部分介绍
  - 第一讲：程序转换概述
    - 机器指令和汇编指令
    - 机器级程序员感觉到的属性和功能特性
    - 高级语言程序转换为机器代码的过程
  - 第二讲：IA-32 /x86-64指令系统
  - 第三讲： C语言程序的机器级表示
    - 过程调用的机器级表示
    - 选择语句的机器级表示
    - 循环结构的机器级表示
  - 第四讲：复杂数据类型的分配和访问
    - 数组的分配和访问
    - 结构体数据的分配和访问
    - 联合体数据的分配和访问
    - 数据的对齐
  - 第五讲：越界访问和缓冲区溢出

从高级语言程序出发，用其对应的机器级代码以及内存（栈）中信息的变化来说明底层实现

围绕C语言中的语句和复杂数据类型，解释其在底层机器级的实现方法

过程调用的机器级表示

以下过程（函数）调用对应的机器级代码是什么？

如何将t1(125)、t2(80)分别传递给add中的形式参数x、y

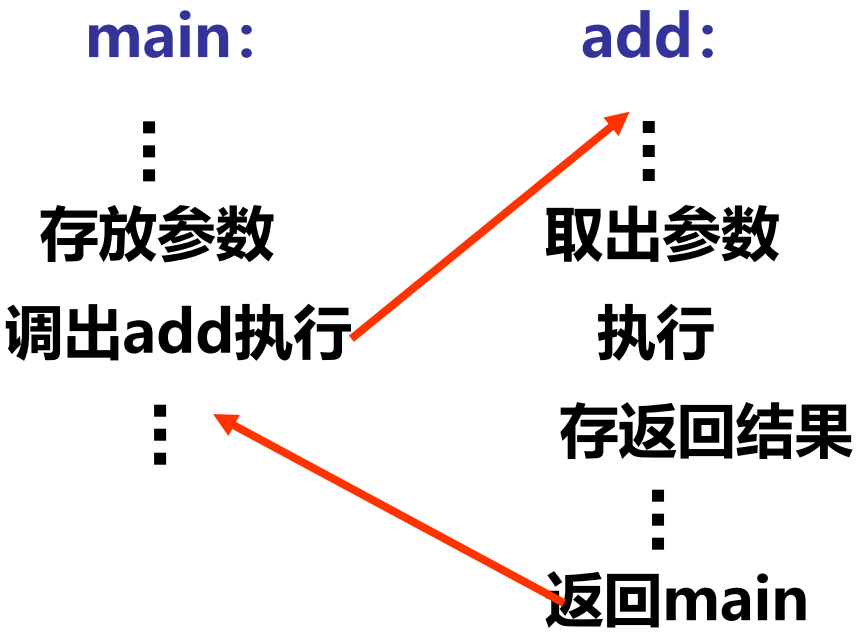
add函数执行的结果如何返回给caller?

```
int add ( int x, int y ) {
    return x+y;
}
int main ( ) {
    int t1 = 125;
     int t2 = 80;
    int sum = add (t1, t2);
    return sum;
}
```

add

↑

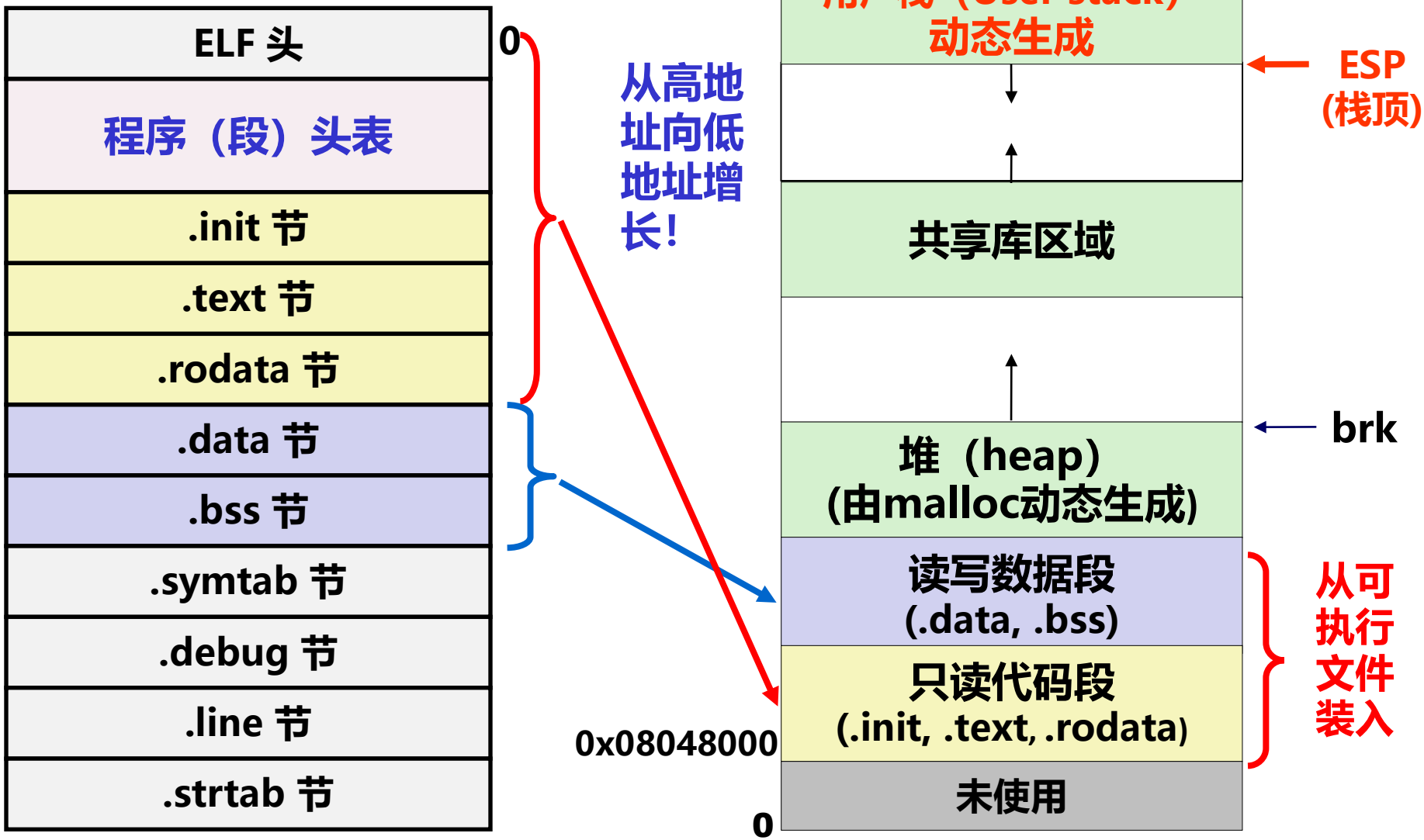
main



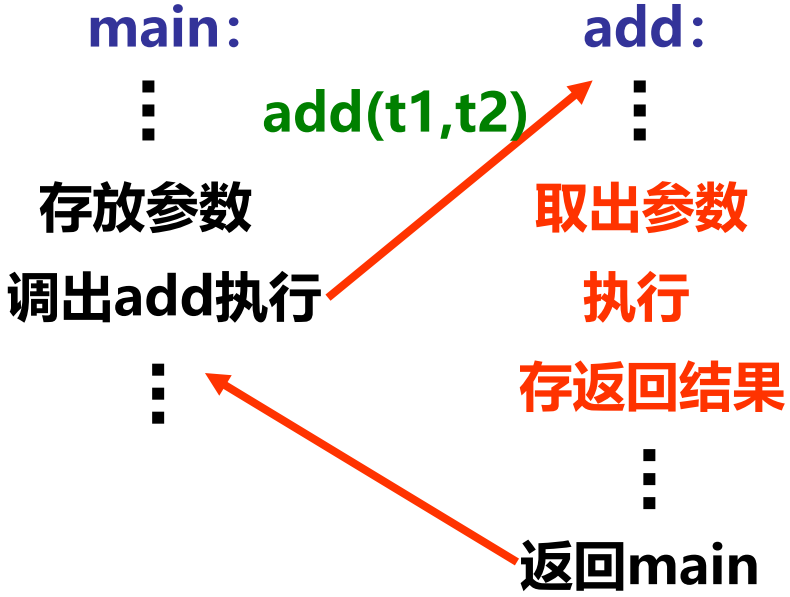
为了统一，模块代码之间必须遵循调用接口约定，称为调用约定(calling convention)，具体由ABI规范定义，编译器强制执行，汇编语言程序员也必须强制按照这些约定执行，包括寄存器的使用、栈帧的建立和参数传递等。

可执行文件的存储器映像

IA-32中参数通过栈来传递
栈（stack）在哪里?



过程调用的机器级表示



何为现场?

通用寄存器的内容!

为何要保存现场?

因为所有过程共享一套通用寄存器

想象：妈妈做菜过程中，让你来完成其中一个工序时共用一套盘子的情况

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将返回地址放到Q能取到的地方，然后将控制转移到Q；

P过程

CALL指令

(3) Q保存P的现场，并为非静态局部变量分配空间；

准备阶段

(4) 执行Q的过程体（函数体）；处理阶段

(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。RET指令

结束阶段

Q过程

过程调用的机器级表示

- i386 System V ABI规范约定
  - 调用者保存寄存器：EAX、EDX、ECX　相当于妈妈腾空的盘子

    当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　相当于妈妈还要用的盘子

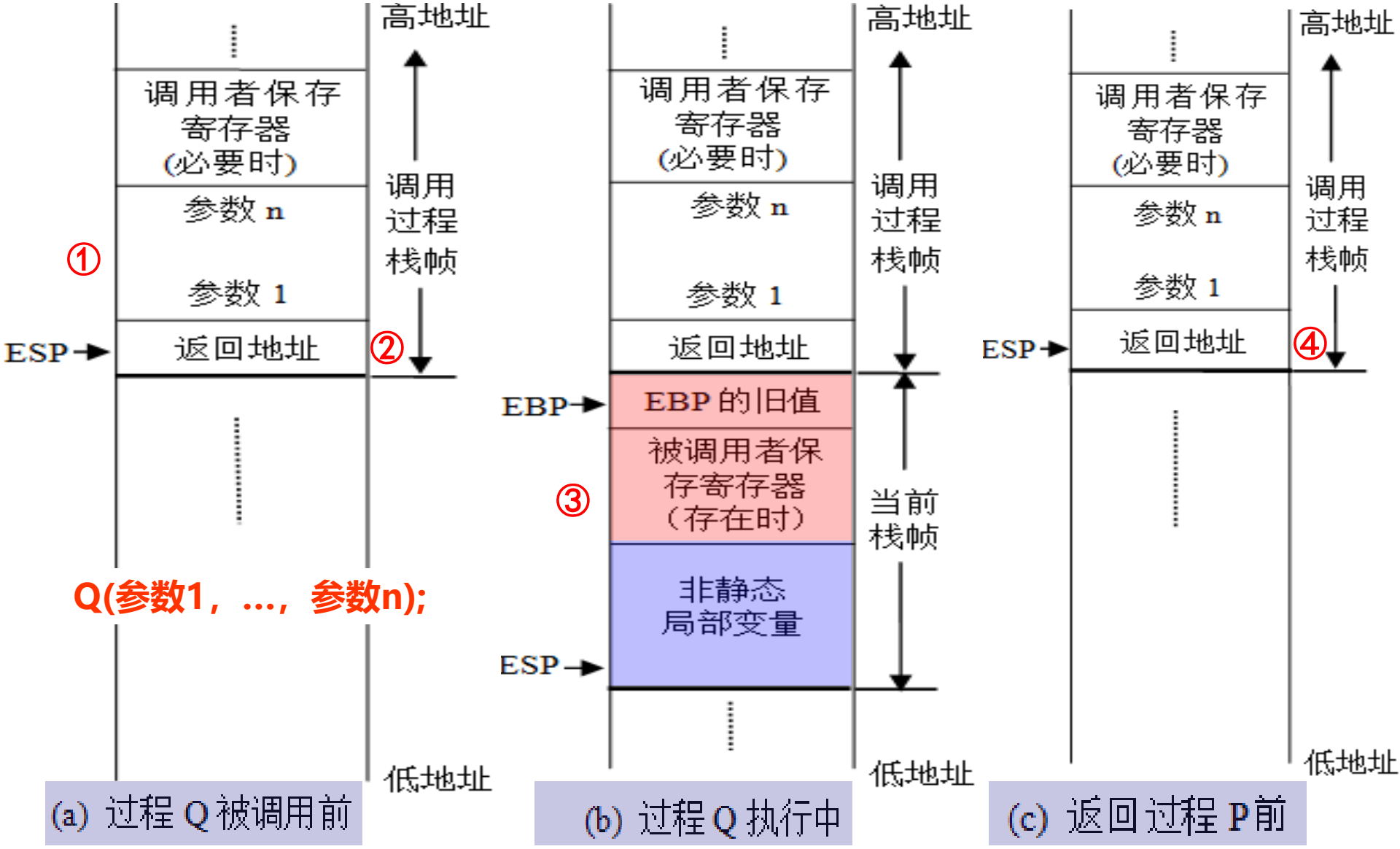
    Q必须先将它们的值保存到栈中再使用它们，并在返回P之前恢复它们的值。
  - EBP和ESP分别是帧指针寄存器和栈指针寄存器，分别用来指向当前栈帧的底部和顶部。

问题：为减少准备和结束阶段的开销，每个过程应先使用哪些寄存器？

EAX、ECX、EDX!

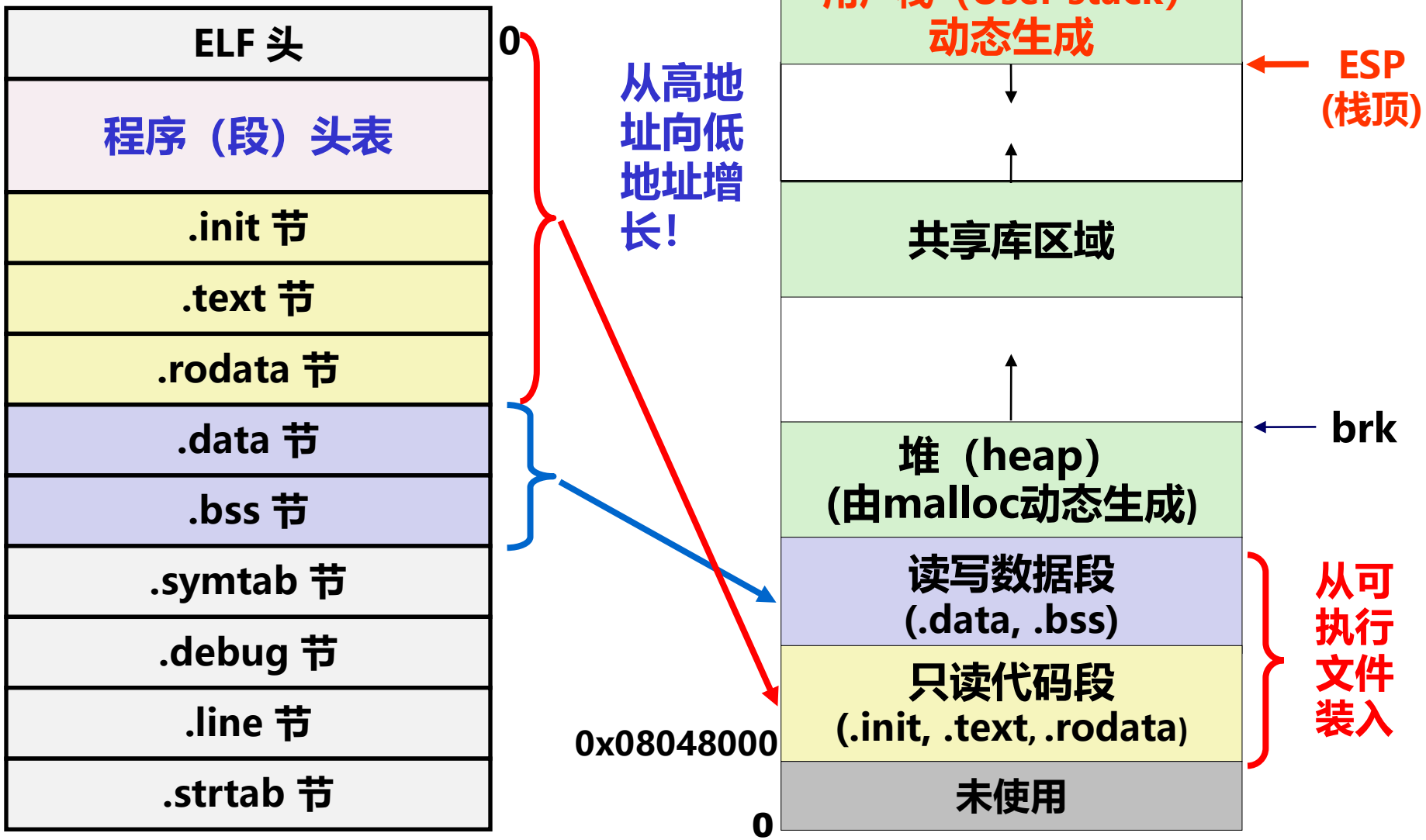
过程调用的机器级表示

- 过程调用过程中**栈和栈帧**的变化 (Q为被调用过程)



Linux可执行文件的存储映像

程序(段)头表描述如何映射



```
int add ( int x, int y ) {
      return x+y;
}
int  caller ( ) {
      int   t1 = 125;
      int   t2 = 80;
      int   sum = add (t1, t2);
      return sum;
}
```

add

caller

P

caller:

```
pushl   %ebp
movl    %esp, %ebp
subl    $24, %esp
movl    $125, -12(%ebp)
movl    $80, -8(%ebp)
movl    -8(%ebp), %eax
movl    %eax, 4(%esp)
movl    -12(%ebp), %eax
movl    %eax, (%esp)
call    add
movl    %eax, -4(%ebp)
movl    -4(%ebp), %eax
leave
ret
```

准备阶段

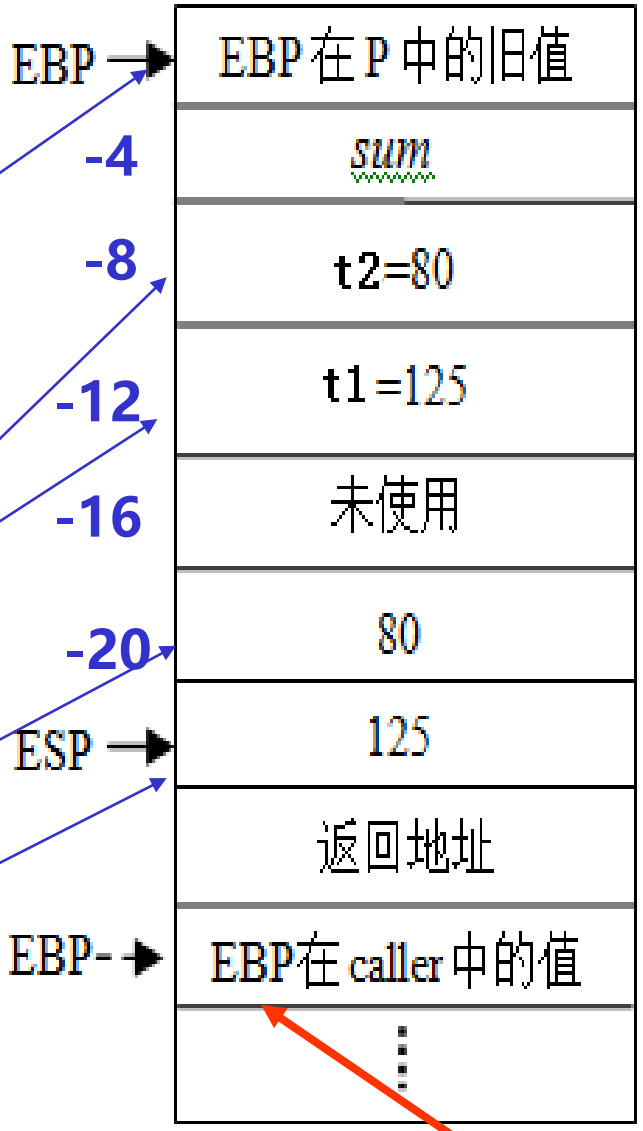
分配局部变量

准备入口参数

返回参数总在EAX中

准备返回参数

结束阶段



caller帧底

caller栈帧

ESP+4

add栈帧底部

movl %ebp, %esp
popl %ebp

add函数开始是什么?

pushl %ebp

movl %esp, %ebp

过程（函数）的结构

- 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）：push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

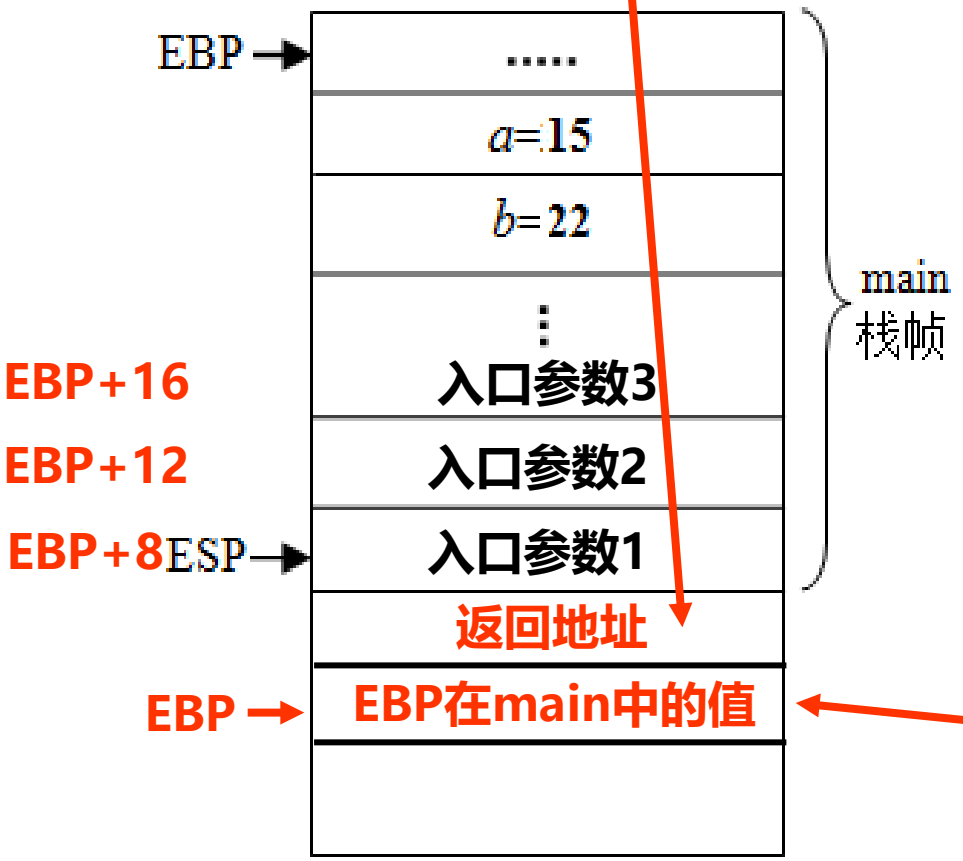
入口参数的位置

```
movl  参数3，8(%esp)
.....
movl  参数1, (%esp)
call   add   R[esp]←R[esp]-4
             M[R[esp]]←返回地址
             R[eip]←add函数首地址
```

准备入口参数

返回地址是什么?

call指令的下一条指令的地址!



i386 System V ABI规范规定，栈中参数按4字节对齐

- 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
- 每个过程开始两条指令

```
pushl %ebp
movl %esp, %ebp
```

过程调用参数传递举例

程序一

```
#include <stdio.h>
main ( )
{
   int a=15, b=22;
   printf (“a=%d\tb=%d\n”, a, b);
   swap (&a, &b);
   printf (“a=%d\tb=%d\n”, a, b);
}
swap (int *x, int *y )
{
    int t=*x;
    *x=*y;
    *y=t;
}
```

按地址传递参数

执行结果？为什么？

程序一的输出：

a=15　b=22

a=22　b=15

程序二

```
#include <stdio.h>
main ( )
{
   int a=15, b=22;
   printf (“a=%d\tb=%d\n”, a, b);
   swap (a, b);
   printf (“a=%d\tb=%d\n”, a, b);
}
swap (int x, int y )
{
    int t=x;
    x=y;
    y=t;
}
```

按值传递参数

程序二的输出：

a=15　b=22

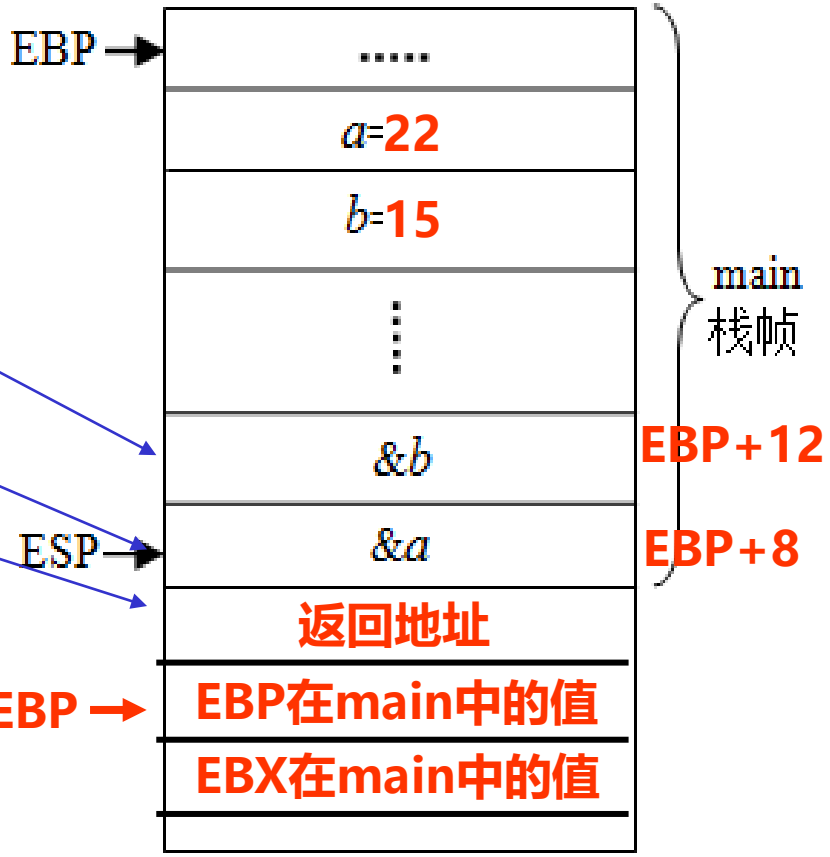
a=15　b=22

过程调用参数传递举例

按地址传递参数swap (&a, &b)

```
main:    ......
      leal    -8(%ebp), %eax
      movl    %eax, 4(%esp)
      leal    -4(%ebp), %eax
      movl    %eax, (%esp)
      call    swap
          ......
      ret
swap:
      pushl   %ebp
      movl    %esp, %ebp
      pushl   %ebx
      movl    8(%ebp), %edx
      movl    (%edx), %ecx
      movl    12(%ebp), %eax
      movl    (%eax), %ebx
      movl    %ebx, (%edx)
      movl    %ecx, (%eax)
```

```
int t=*x;
*x=*y;
*y=t;
```



EBP → EBP在main中的值

EBX是被调用者保存

R[ecx]←M[&a]=15

R[ebx]←M[&b]=22

M[&a] ← R[ebx] =22

M[&b] ← R[ecx] = 15

局部变量a和b进行了交换

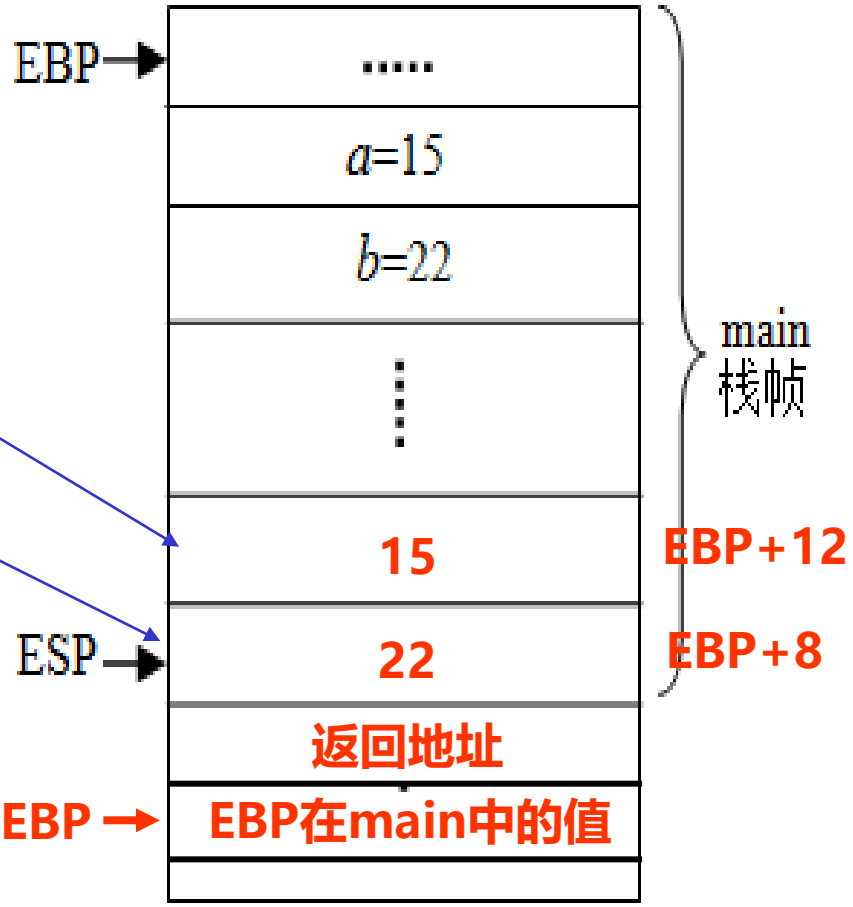
过程调用参数传递举例

按值传递参数swap (a, b)

```
main:  ......
      movl   -8(%ebp), %eax
      movl   %eax, 4(%esp)
      movl   -4(%ebp), %eax
      movl   %eax, (%esp)
      call   swap
          ......
     ret
swap:
```

```
int t=x;
    x=y;
    y=t;
```

```
      pushl   %ebp
      movl    %esp, %ebp
      movl    8(%ebp), %edx
      movl    12(%ebp), %eax
      movl    %eax, 8(%ebp)
      movl    %edx, 12(%ebp)
```



R[edx]←15

R[eax]←22

M[R[ebp]+8] ← R[eax] =22

M[R[ebp]+12] ← R[edx] =15

局部变量a和b没有交换，交换的仅是入口参数

过程调用

```
1  void test ( int x, int *ptr )
2  {
3       if  ( x>0 && *ptr>0 )
4       *ptr+=x;
5   }
6                  100   200
7  void caller (int a, int y )
8  {
9        int x = a>0 ? a : a+100;
10       test (x, &y);
11  }
```

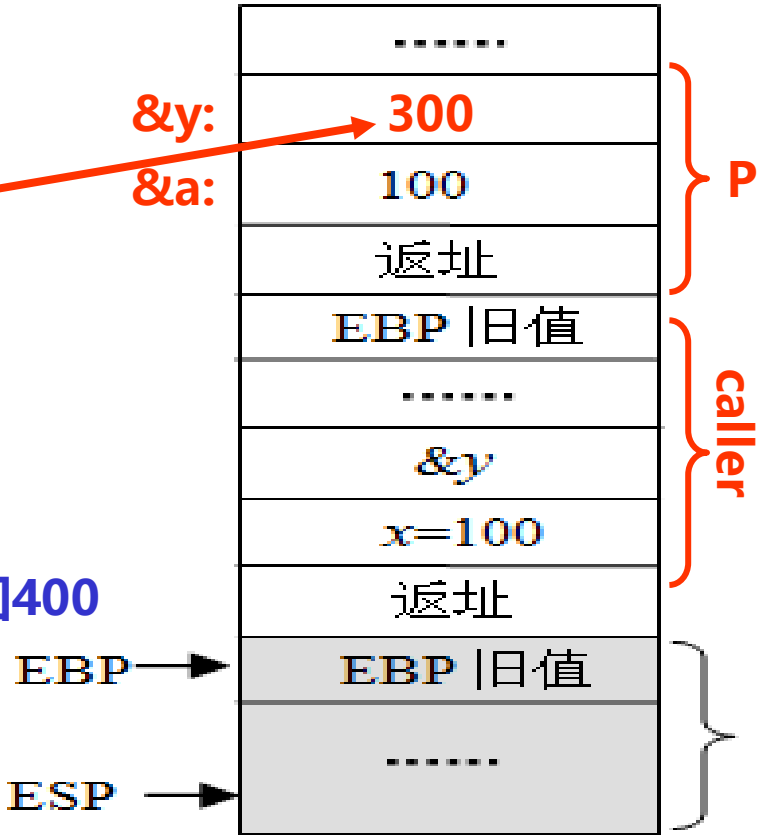
test

caller

P

则函数返回400

若return x+y;



调用caller的过程为P，P中给出形参a和y的实参分别是100和200，画出相应栈帧中的状态，

(1) test的形参是按值传递还是按地址传递？test的形参ptr对应的实参是一个什么类型的值？ 前者按值、后者按地址。一定是一个地址

(2) test中被改变的*ptr的结果如何返回给它的调用过程caller？

第10行执行后，P帧中200变成300，test退帧后，caller中通过y引用该值300

(3) caller中被改变的y的结果能否返回给过程P？为什么？

第11行执行后caller退帧并返回P，因P中无变量与之对应，故无法引用该值300

```
int  nn_sum ( int n)
{
    int result;
    if  (n<=0 )
        result=0;
    else
        result=n+nn_sum(n-1)
    return  result;
}
```

```
pushl   %ebp
movl    %esp, %ebp
pushl   %ebx
subl    $4, %esp
movl    8(%ebp), %ebx
movl    $0, %eax
cmpl    $0, %ebx
jle     .L2
leal    -1(%ebx), %eax
movl    %eax, (%esp)
call    nn_sum
addl    %ebx, %eax
.L2
addl    $4, %esp
popl    %ebx
popl    %ebp
ret
```

R[ebx]←n

R[eax]←0

if (n≤0) 转L2

R[eax]←n-1

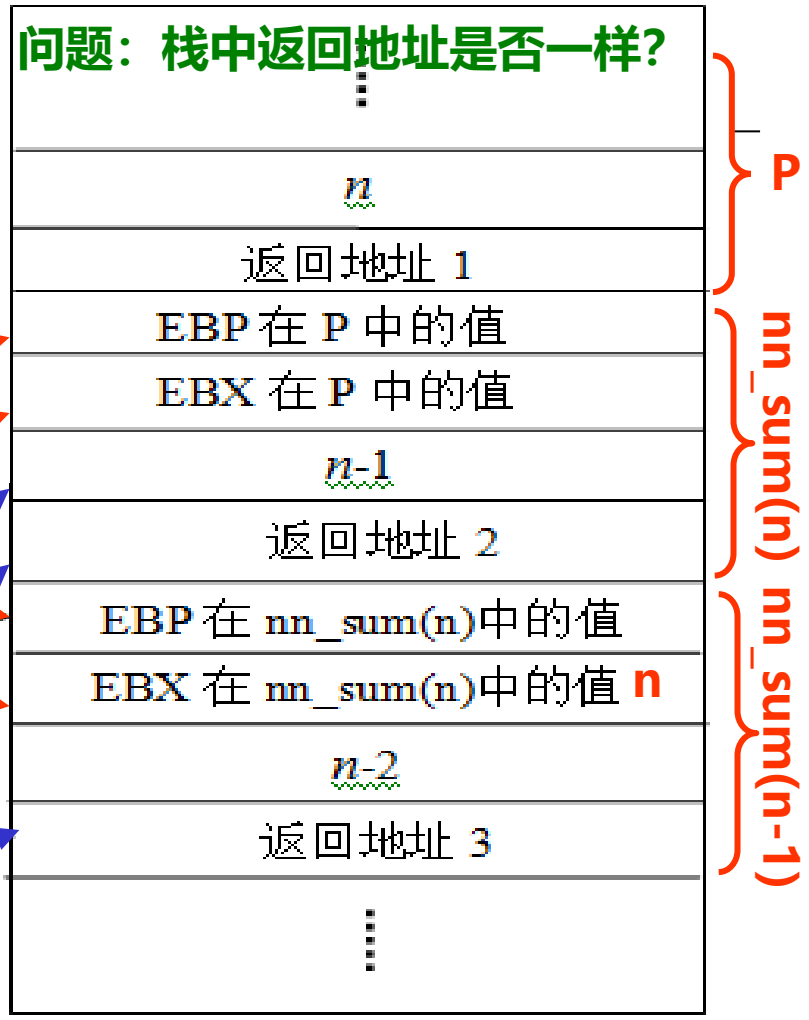
R[eax] ← 0+1+2+…+(n-1)+n

nn_sum(n-1)

nn_sum(n)

P

问题：栈中返回地址是否一样?

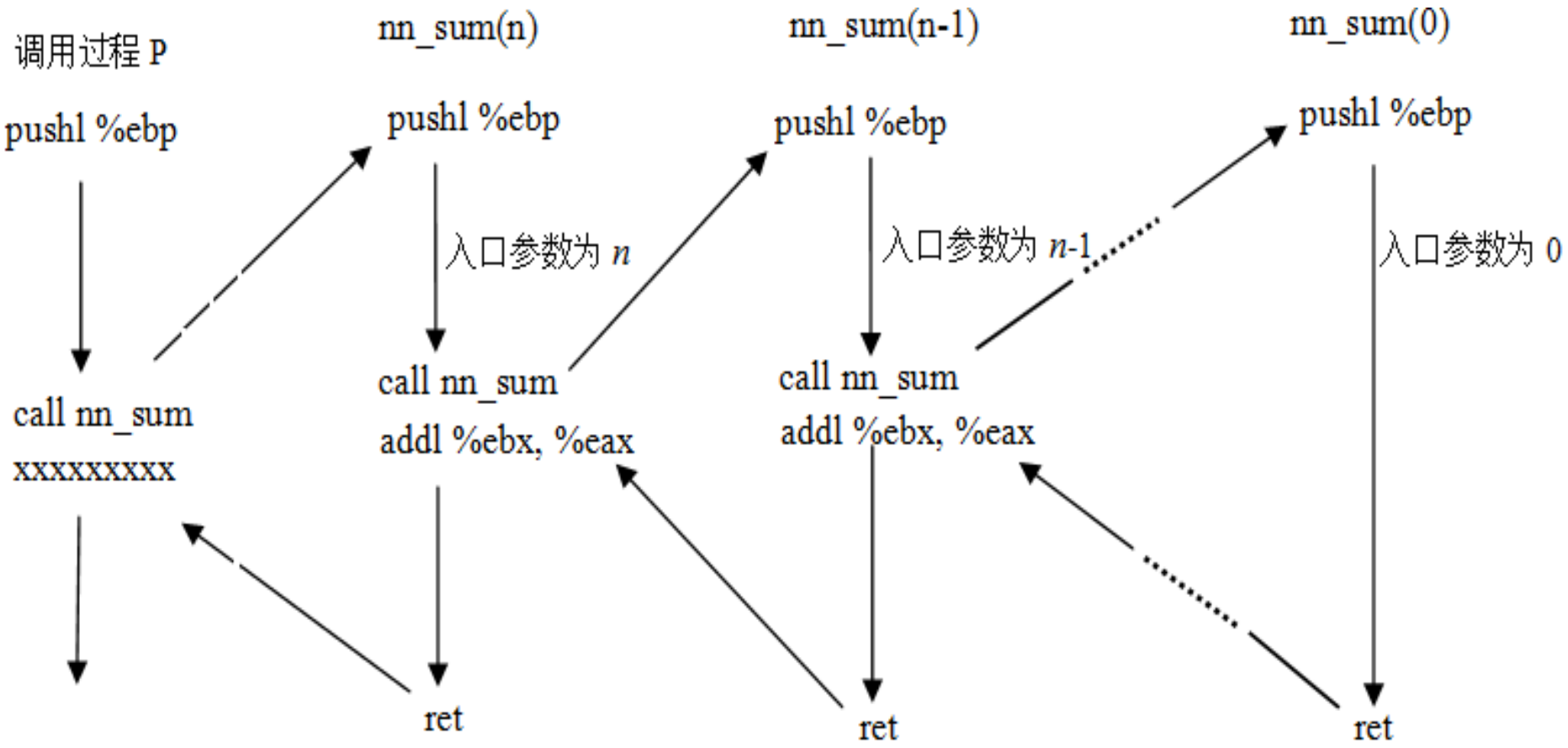


操作系统为程序分配的栈会有默认的大小限制

每次递归调用都会增加一个栈帧（该例为16B），所以空间开销很大。当n很大时会发生栈溢出!

过程调用的机器级表示

- **递归函数nn_sum的执行流程**



为支持过程调用，每个过程包含准备阶段和结束阶段。因而每增加一次过程调用，就要增加许多条包含在准备阶段和结束阶段的额外指令，它们对程序性能影响很大，应尽量避免不必要的过程调用，特别是递归调用。

过程调用举例

例：应始终返回d[0]中的3.14，但并非如此。Why?

```
double fun(int i)
{
  volatile double d[1] = {3.14};
  volatile long int a[2];
  a[i] = 1073741824; /* Possibly out of bounds */
  return d[0];
}
```

fun(0) → 3.14
fun(1) → 3.14
fun(2) → 3.1399998664856
fun(3) → 2.00000061035156
fun(4) → 3.14, 然后存储保护错

为何每次返回不一样?

为什么会引起保护错?

栈帧中的状态如何?

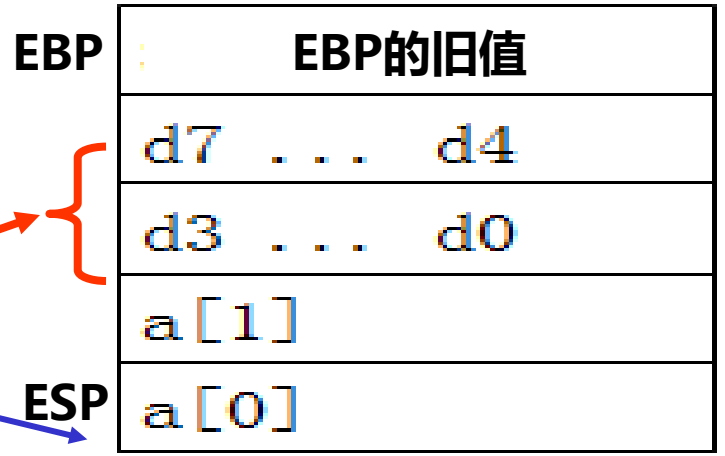
不同系统上执行结果可能不同

例如，编译器对局部变量分配方式可能不同

```
double fun(int i)
{
  volatile double d[1] = {3.14};
  volatile long int a[2];
  a[i] = 1073741824;
  return d[0];
}
```

当i=0或1，OK
当i=2, d3~d0=0x40000000
低位部分（尾数）被改变
当i=3, d7~d3=0x40000000
高位部分被改变
当i=4, EBP被改变

```
<fun>:
    push   %ebp
    mov    %esp,%ebp
    sub    $0x10,%esp
    fldl   0x8048518
    fstpl  -0x8(%ebp)
    mov    0x8(%ebp),%eax
    movl   $0x40000000,-0x10(%ebp,%eax,4)
    fldl   -0x8(%ebp)
    leave
    ret
```



a[i]=1073741824;

0x40000000
$=2^{30}=1073741824$

return d[0];

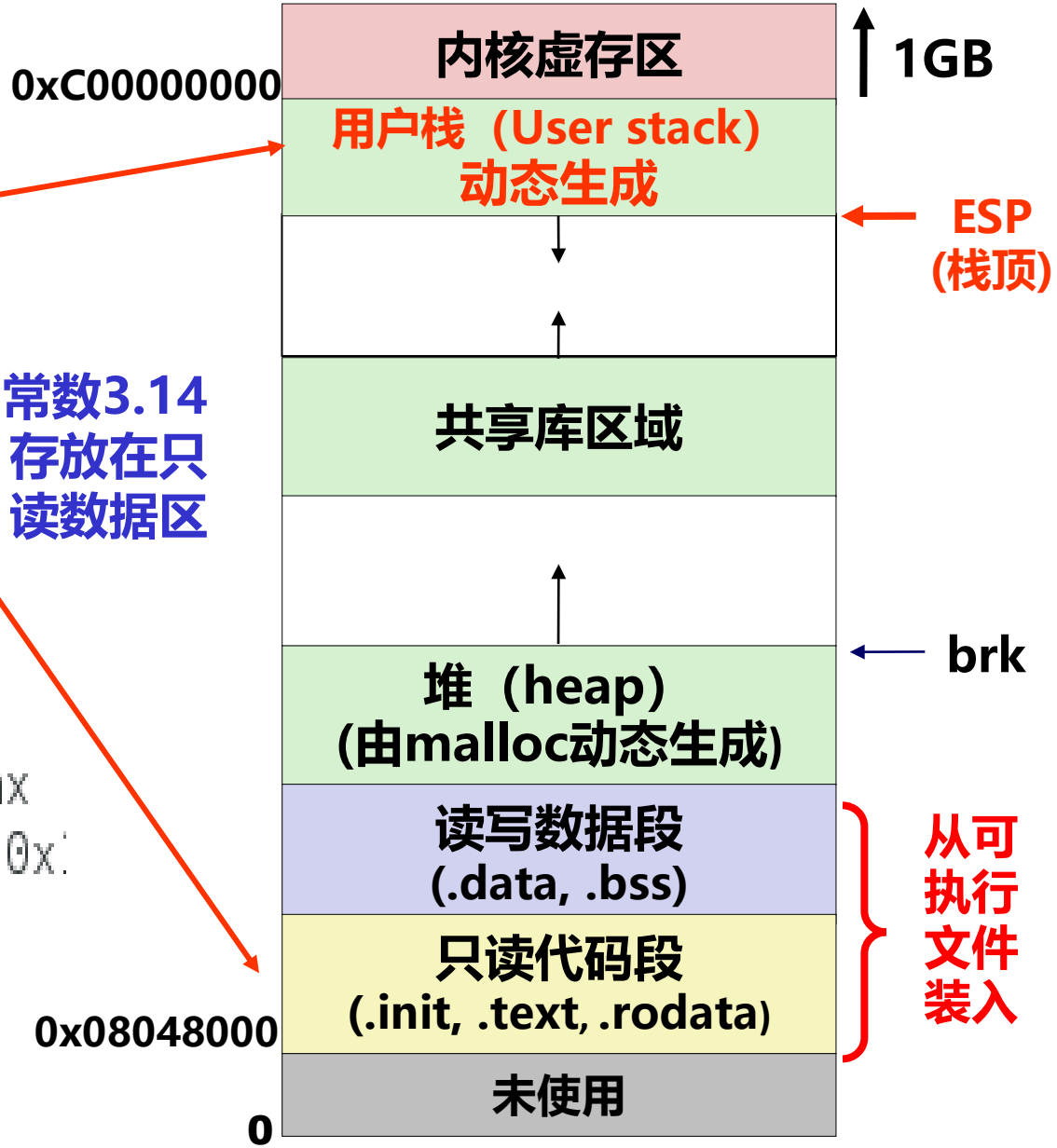
fun(2) = 3.1399998664856
fun(3) = 2.00000061035156
fun(4) = 3.14, 然后存储保护错

IA-32/Linux的存储映像

只读代码段地址的特点：
0x8048xxx
栈区地址特点：
0xbfffxxxx

```
<fun>:
    push   %ebp
    mov    %esp,%ebp
    sub    $0x10,%esp
    fldl   0x8048518
    fstpl  -0x8(%ebp)
    mov    0x8(%ebp),%eax
    movl   $0x40000000,-0x
    fldl   -0x8(%ebp)
    leave
    ret
```

常数3.14
存放在只
读数据区



Windows/Linux中的存储映像

说明了什么？注意：每个存储区地址的特征！

```
1  #include<stdio.h>
2  int func(int param1, int param2, int param3)
3  {
4      int var1 = param1;
5      int var2 = param2;
6      int var3 = param3;
7      printf("0x%p\n", &param1);
8      printf("0x%p\n", &param2);
9      printf("0x%p\n\n", &param3);
10     printf("0x%p\n", &var1);
11     printf("0x%p\n", &var2);
12     printf("0x%p\n\n", &var3);
13     return 0;
14 }
15 int main()
16 {
17     func(1, 2, 3);
18     return 0;
19 }
```

局部变量和参数都存放在：栈区

Linux

```
0x0xffff2b50
0x0xffff2b54
0x0xffff2b58

0x0xffff2b34
0x0xffff2b38
0x0xffff2b3c
```

参数的地址总比局部变量的地址大！

因为栈的生长方向：高地址→低地址

Linux总是最右边参数的地址最大，因为参数入栈顺序为：右→左

Windows中的存储映像

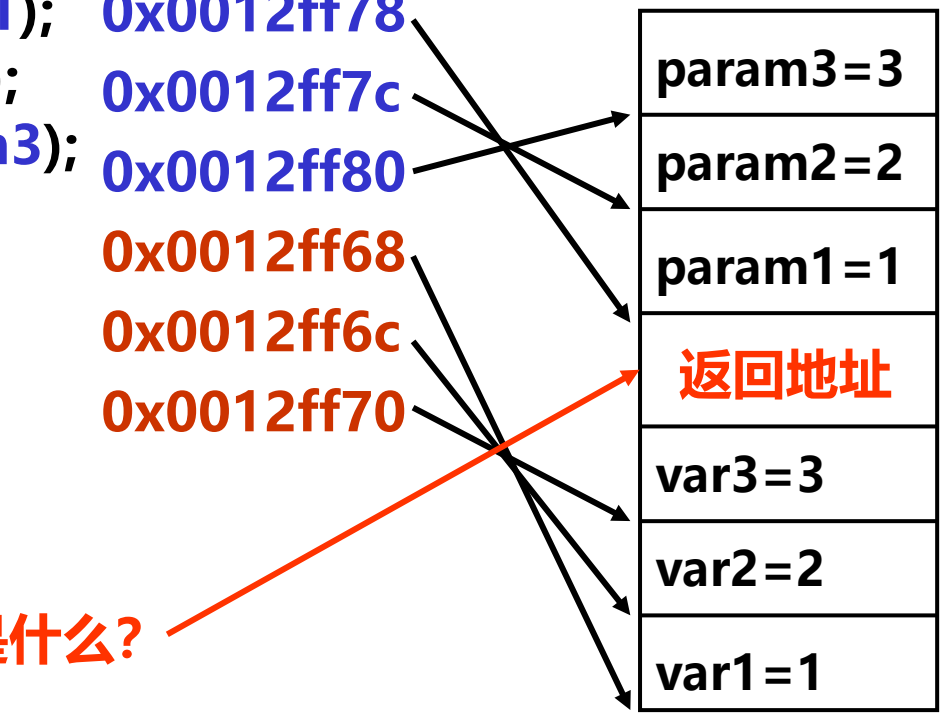
```
#include  ……
void __stdcall func(int param1,int param2,int param3)
{
    int var1=param1;
    int var2=param2;
    int var3=param3;
    printf( “0x%08x\n” ,&param1);
    printf("0x%08x\n", &param2);
    printf("0x%08x\n\n", &param3);
    printf("0x%08x\n",&var1);
    printf("0x%08x\n",&var2);
    printf("0x%08x\n\n",&var3);
    return;
}
int main()
{
    func(1,2,3);
    return 0;
}
```

说明了什么?

Windows中栈区也是高地址向低地址生长!

执行结果如下:

0x0012ff78
0x0012ff7c
0x0012ff80
0x0012ff68
0x0012ff6c
0x0012ff70



猜猜这里是什么?

这里与Linux的差别是什么? EBP未压栈!

Windows中的存储映像

```
#include  ....
int g1=0, g2=0, g3=0;
int main()
{
    static int s1=0, s2=0, s3=0;
    int v1=0, v2=0, v3=0;
     printf("0x%08x\n",&v1);
    printf("0x%08x\n",&v2);
    printf("0x%08x\n\n",&v3);
    printf("0x%08x\n",&g1);
    printf("0x%08x\n",&g2);
    printf("0x%08x\n\n",&g3);
    printf("0x%08x\n",&s1);
     printf("0x%08x\n",&s2);
    printf("0x%08x\n\n",&s3);
    return 0;
}
```

说明了什么?

注意：每个存储区地址的特征!

执行结果如下：

0x0012ff78
0x0012ff7c
0x0012ff80

局部变量存放在另一个存储区：栈区

0x004068d0
0x004068d4
0x004068d8
0x004068dc
0x004068e0
0x004068e4

全局变量和静态变量连续存放在同一个存储区：可读写数据区

Windows/Linux中的存储映像

说明了什么？注意：每个存储区地址的特征！

```
1  #include <stdio.h>
2  int g1 = 0, g2 = 0, g3 = 0;
3  int main()
4  {
5      static int s1 = 0, s2 = 0, s3 = 0;
6      int v1 = 0, v2 = 0, v3 = 0;
7      printf("0x%p\n", &v1);
8      printf("0x%p\n", &v2);
9      printf("0x%p\n\n", &v3);
10     printf("0x%p\n", &g1);
11     printf("0x%p\n", &g2);
12     printf("0x%p\n\n", &g3);
13     printf("0x%p\n", &s1);
14     printf("0x%p\n", &s2);
15     printf("0x%p\n\n", &s3);
16     return 0;
17 }
```

全局、静态变量不一定按顺序分配

Windows	Linux
0x0013FD28	0x0xff8a5864
0x0013FD24	0x0xff8a5868
0x0013FD20	0x0xff8a586c
0x00C43384	0x0x804a024
0x00C43380	0x0x804a028
0x00C4337C	0x0x804a02c
0x00C43374	0x0x804a030
0x00C43378	0x0x804a034
0x00C43388	0x0x804a038

局部变量存放在另一个存储区：栈区

全局变量和静态变量连续存放在同一个存储区：可读写数据区

Windows中的分配顺序

```
#include  ……
int main()
{
    int a;
    char b;
    int c;
    printf(  “a: 0x%08x\n",&a);
    printf(  “b: 0x%08x\n",&b);
    printf(  “c: 0x%08x\n",&c);
    return 0;
}
```

a、b、c不一定
按顺序分配！

用VC编译后的执行结果：

a: 0x0012ff7c

b: 0x0012ff7b

c: 0x0012ff80

顺序：b(1B)-a(4B)-c(4B)

用Dev-C++编译后的执行结果：

a: 0x0022ff7c

b: 0x0022ff7b

c: 0x0022ff74

顺序：c(4B)-隔3B-b(1B)-a(4B)

用lcc编译后的执行结果：

a: 0x0012ff6c

b: 0x0012ff6b

c: 0x0012ff64

顺序：同上（大地址->小地址）

变量的存储分配

非静态局部变量占用的空间分配在本过程的栈帧中

全局、静态变量在可读可写数据区分配

- C标准中，没有规定必须按顺序分配，不同的编译器有不同的处理方式。
- C标准明确指出，对不同变量的地址进行除==和!=之外的关系运算，都属未定义行为（undefined behavior）

  如，语句“if (&var1 < &var2) {...};”属于未定义行为
- 编译优化的情况下，会把属于简单数据类型的变量分配在通用寄存器中
- 对于复杂数据类型变量，如数组、结构和联合等数据类型变量，一定会分配在存储器中

有关“过程调用”的练习

以下是一个C语言程序代码：

```
int add(int x, int y)
{
    return x+y;
}
int caller( )
{
    int t1=100 ;
    int t2=200;
    int sum=add(t1, t2);
    return sum;
}
```

C

以下关于上述代码在 IA-32+Linux上执行情况的叙述中，错误的是（ ）。

A. 变量t1、t2和sum被分配在寄存器或caller函数的栈帧中

B. 传递参数时t2和t1的值从高地址到低地址依次存入栈中

C. 入口参数t1和t2的值被分配在add函数的栈帧中

D. add函数返回时返回值存放在EAX寄存器中

```
int add ( int x, int y ) {
      return x+y;
}
int  caller ( ) {
      int   t1 = 125;
      int   t2 = 80;
      int   sum = add (t1, t2);
      return sum;
}
```

add

caller

P

caller:

```
pushl   %ebp
movl    %esp, %ebp
subl    $24, %esp
movl    $125, -12(%ebp)
movl    $80, -8(%ebp)
movl    -8(%ebp), %eax
movl    %eax, 4(%esp)
movl    -12(%ebp), %eax
movl    %eax, (%esp)
call    add
movl    %eax, -4(%ebp)
movl    -4(%ebp), %eax
leave
ret
```

准备阶段

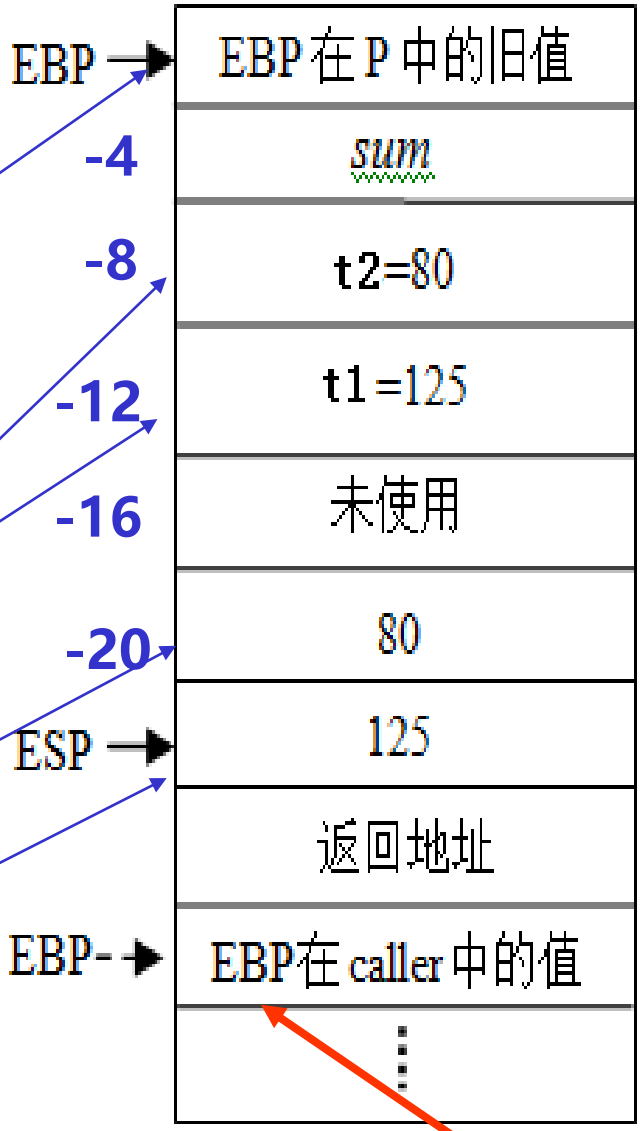
分配局部变量

准备入口参数

返回参数总在EAX中

准备返回参数

结束阶段



caller帧底

caller栈帧

ESP+4

add栈帧底部

movl %ebp, %esp
popl %ebp

add函数开始是什么?

pushl %ebp

movl %esp, %ebp

有关“过程调用”的练习 SKIP

以下是一个C语言程序代码：

```
int add(int *xp, int *yp)
{
    return *xp+*yp;
}
void caller( )
{
    static int t1=100;
    static int t2=200;
    int sum=add(&t1, &t2);
    int diff=sub(&t1, &t2);
    printf( “sum=%d, diff=%d", sum, diff);
}
```

思考题：

若改为以下语句，则怎样？

```
int diff;
sub(&diff,&t1,&t2);
```

B

D?

翻转题目一：

用更复杂的例子来说明和演示有多个被调用过程的情况。

以下关于上述代码在 IA-32+Linux上执行情况的叙述中，错误的是（ ）。

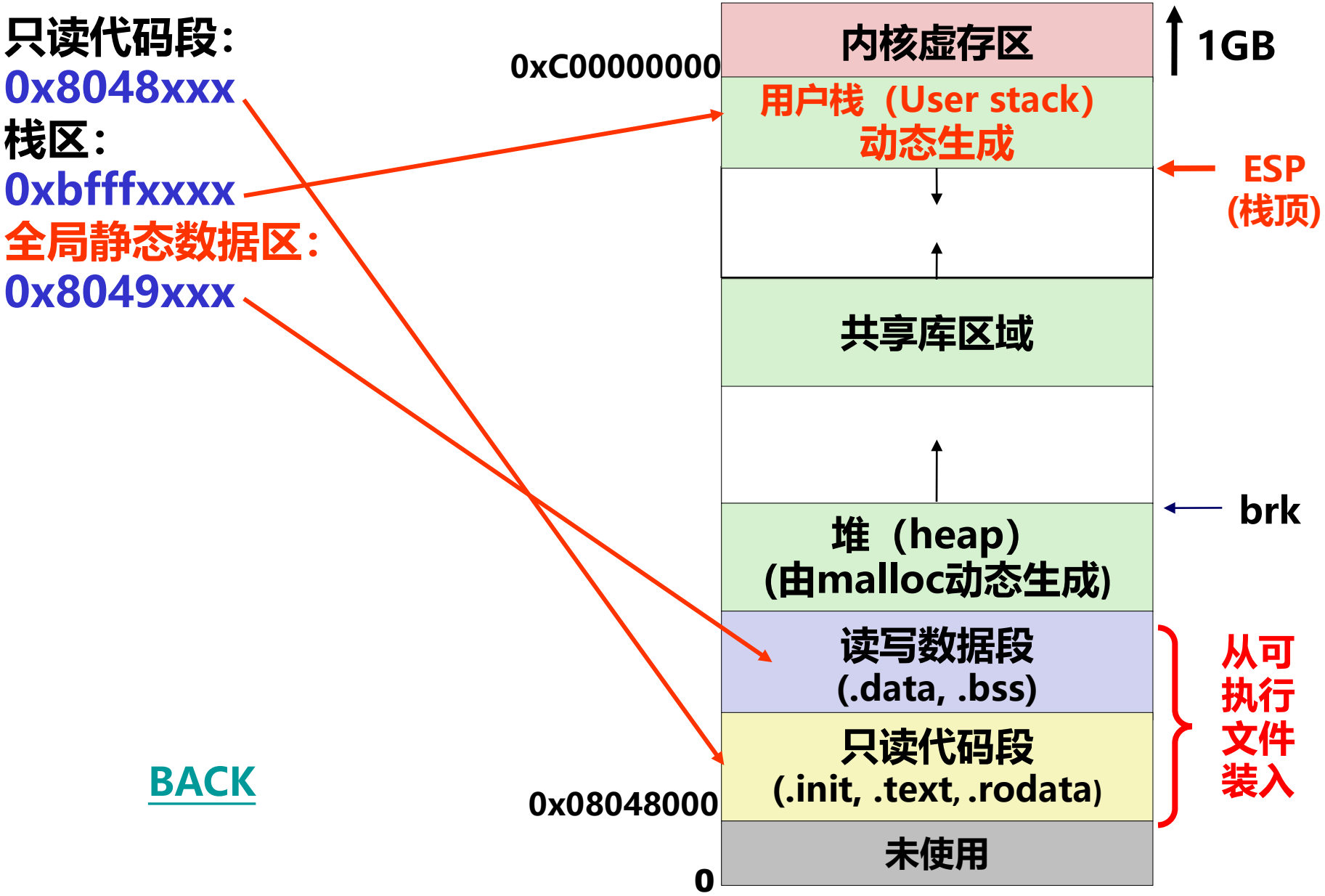
A. 变量t1、t2被分配在可读可写的全局静态数据区中

B. 存入栈中的入口参数可能是0xbfff0004、0xbfff0000

C. 在caller中执行leave指令后，入口参数的值还在存储器中

D. add函数和sub函数的栈帧底部在完全相同的位置处?

IA-32/Linux的存储映像



```
int add ( int x, int y ) {
      return x+y;
}
int  caller ( ) {
      ......
      int   sum = add (t1, t2);
      int  diff= sub (t1, t2);
      ......
}
```

add

caller

P

```
caller:
 pushl    %ebp
 movl     %esp, %ebp
 subl     $24, %esp
 ......
 ......
 movl     -12(%ebp), %eax
 movl     %eax, (%esp)
 call     add
 .......
 call     sub
 ......
 leave
 ret
```

准备阶段

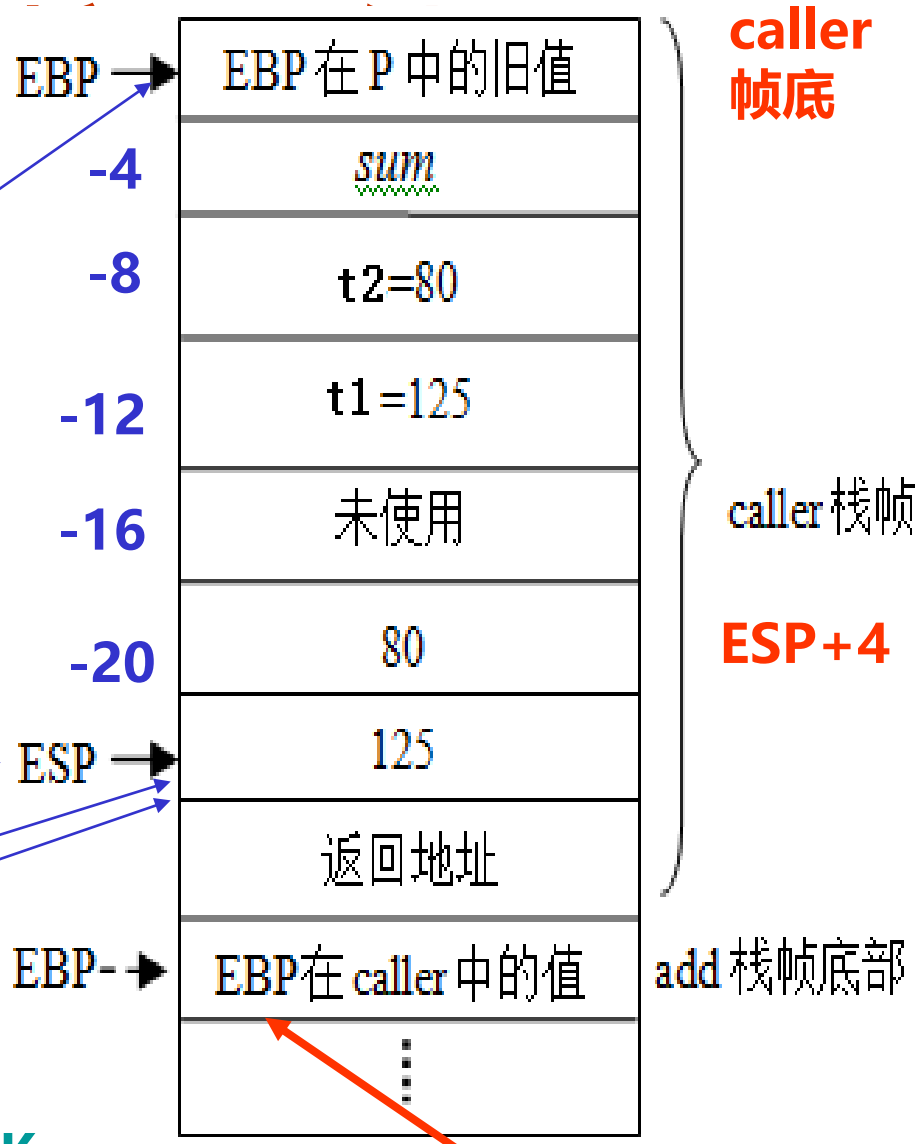
准备入口参数

准备入口参数

结束阶段

BACK

```
movl %ebp, %esp
popl  %ebp
```



add函数开始是什么?

```
pushl   %ebp
movl   %esp, %ebp
```

有关“过程调用”的讨论

为什么以下程序输出结果是x=-1217400844而不是x=100？在你的机器上执行结果是什么？每次执行结果都一样吗？反汇编后的机器级代码如何支持你的分析？

```
int x=100;
void main ( )
{    int x;
     printf( “x=%d\n” , x);
}
```

稍作修改后输出结果是什么？

```
int x=100;
void main ( )
{   int x=10;
    printf( “x=%d\n” , x);
}
```

```
int x=100;
void main ( )
{     static int x;
      printf( “x=%d\n” , x);
}
```

```
void main ( )
{   int x;
    printf( “x=%d\n” , x);
}
```

字符串“x=%d\n”属于只读数据

```
0804841c <main>:
 804841c:       55                              push   %ebp
 804841d:       89 e5                           mov    %esp,%ebp
 804841f:       83 e4 f0                        and    $0xfffffff0,%esp
 8048422:       83 ec 20                        sub    $0x20,%esp
 8048425:       8b 44 24 1c                     mov    0x1c(%esp),%eax
 8048429:       89 44 24 04                     mov    %eax,0x4(%esp)
 804842d:       c7 04 24 d0 84 04 08            movl   $0x80484d0,(%esp)
 8048434:       e8 c7 fe ff ff                  call   8048300 <printf@plt>
 8048439:       c9                              leave
 804843a:       c3                              ret
```

参考答案：

（1）程序中有两个变量x，一个是全局变量x，初值为100，另一个是局部变量x，没有赋初值。这里打印出来的x的值应该是局部变量x的值，局部变量x所占的空间是栈中的4个单元，栈中存储单元的内容不会进行初始化，除非局部变量赋初值，因而**局部变量x的值是一个随机数**（例如，我运行该程序两次得到的结果分别是 x=-1217400844 和 x=-1217273868）。而全局变量所占空间的初值一定是确定的，要么是程序所赋予的初值，要么是0（未赋初值时）。

（2）main函数反汇编后的结果如下，这里局部变量x所占空间的首地址为R[esp]+0x1c，没有任何一条指令对该空间的4个字节赋值，而是直接将4个字节取出，作为printf()函数的参数，存入了首地址为R[esp]+4的空间。

有关“过程调用”的讨论

翻转题目二：以下是网上的一个帖子，请将程序的可执行文件反汇编（基于IA-32），并对汇编代码进行分析以正确回答该贴中的问题。

该贴给出的结果是在Linux还是Windows上得到的?

C/C++ code

```
1  #include "stdafx.h"
2  int main(int argc, char* argv[])
3  {
4      int a=10;
5      double *p=(double*)&a;
6      printf("%f\n",*p);              //结果为0.000000
7      printf("%f\n",(double(a)));     //结果为10.000000
8
9      return 0;
10 }
11 为什么printf("%f",*p)和printf("%f",(double)a)结果不一样呢？
```

SKIP

不都是强制类型转换吗？怎么会不一样

有关“过程调用”的讨论

在32位Linux系统中反汇编结果：

Windows下结果如何?
p: 0x0012ffxx
a: 0xa

```
int a = 10;
 8048425:  c7 44 24 28 0a 00 00 00    movl   $0xa,0x28(%esp)
```

打印出来的是0!

+2c	p : &a=0xbfff0028
+28	a : 0xa
	.....
+8	p : &a=0xbfff0028
+4	a : 0xa
ESP	0x8048500 (指向 “%f\n” 的指针)

假定R[esp]=0xbfff0000

```
                                     lea    0x28(%esp),%eax
                                     mov    %eax,0x2c(%esp)
                                  层次并没有体现出来，都是直接 mov 过去

                                     mov    0x2c(%esp),%eax
                                     fldl   (%eax)
                                  精度加载到浮点栈顶 ST(0)）
                                     fstpl  0x4(%esp)
                                  *p 的类型是 double，故按 64 位压栈）
                                     movl   $0x8048500,(%esp)
                                     call   8048300 <printf@plt>
```

打印出来的是一个负数

```
                                     mov    0x28(%esp),%eax
                                     mov    %eax,0x1c(%esp)
由于没有优化，这里有一些冗余的 mov 操作，把变量 a 的值移来移去
 8048453:  db 44 24 1c               fildl  0x1c(%esp)
把 10 转换成 double 型，注意这里用的是 fildl 指令，和上面用的 fldl 指令不一样!
```

BACK

有关“过程调用”的讨论

- 从上述代码可以看出，对于double *p=(double *)&a，只是把a的地址直接传送到p所存放的空间，然后把p中的内容，也就是a的地址送到了EAX中，随后用指令“fldl (%eax)”将a的地址处开始的8个字节的机器数（xx…x0000000AH）直接加载到ST(0)中，其中前4个字节xx…x表示R[esp]+0x28，在Linux系统中它应该是一个很大的数，如BFFF…，然后再用指令“fstpl 0x4(%esp)”把ST(0)中的内容（即xx…x0000000AH）作为printf函数的参数送到R[esp]+4的位置，printf(“%lf\n”,*p)函数将其作为double类型（%lf）的数打印出来。显然，这个打印的值不会是10.000000，而是一个负数。
- 因为Linux和Windows两种系统所设置的栈底所在地址不同，所以ESP寄存器中的内容不同，因而打印出来的值也肯定不同。通常，Linux中栈底在靠近C0000000H的位置，而在Windows中栈的大致位置是0012FFxxH。因此，可以判断出题目中给出的结果应该是在Windows中执行的结果，打印的值应该是0012 FFxx 0000 000AH 或者 0000 000A 0012 FFxxH对应的double类型的值，前者值为+1.0010….1010×2^{-1022}，后者为+0.0…1… ×2^{-1023}，显然都是接近0的值，正如题目中程序注释所示，结果为0.000000。
- 对于printf(“%lf\n, (double)a) 函数，使用的指令为fildl，该指令先将a作为int型变量（值为10）等值转换为double类型，再加载到ST(0)中。这样再作为double类型（%lf）的数打印时，打印的值就是10.000000。

有关“过程调用”的讨论

翻转题目三：

```
#include <stdio.h>
main()
{
    double a = 10;
    printf("a = %d\n", a);
}
```

在IA-32和x86-64上运行时，结果各是什么？分别考察Windows和Linux系统。

$10=1010B=1.01\times2^3$

阶码e=1023+3=10000000010B

10的double型表示为：

0 10000000010 0100…0B

即4024 0000 0000 0000H

先执行fldl，再执行fstpl

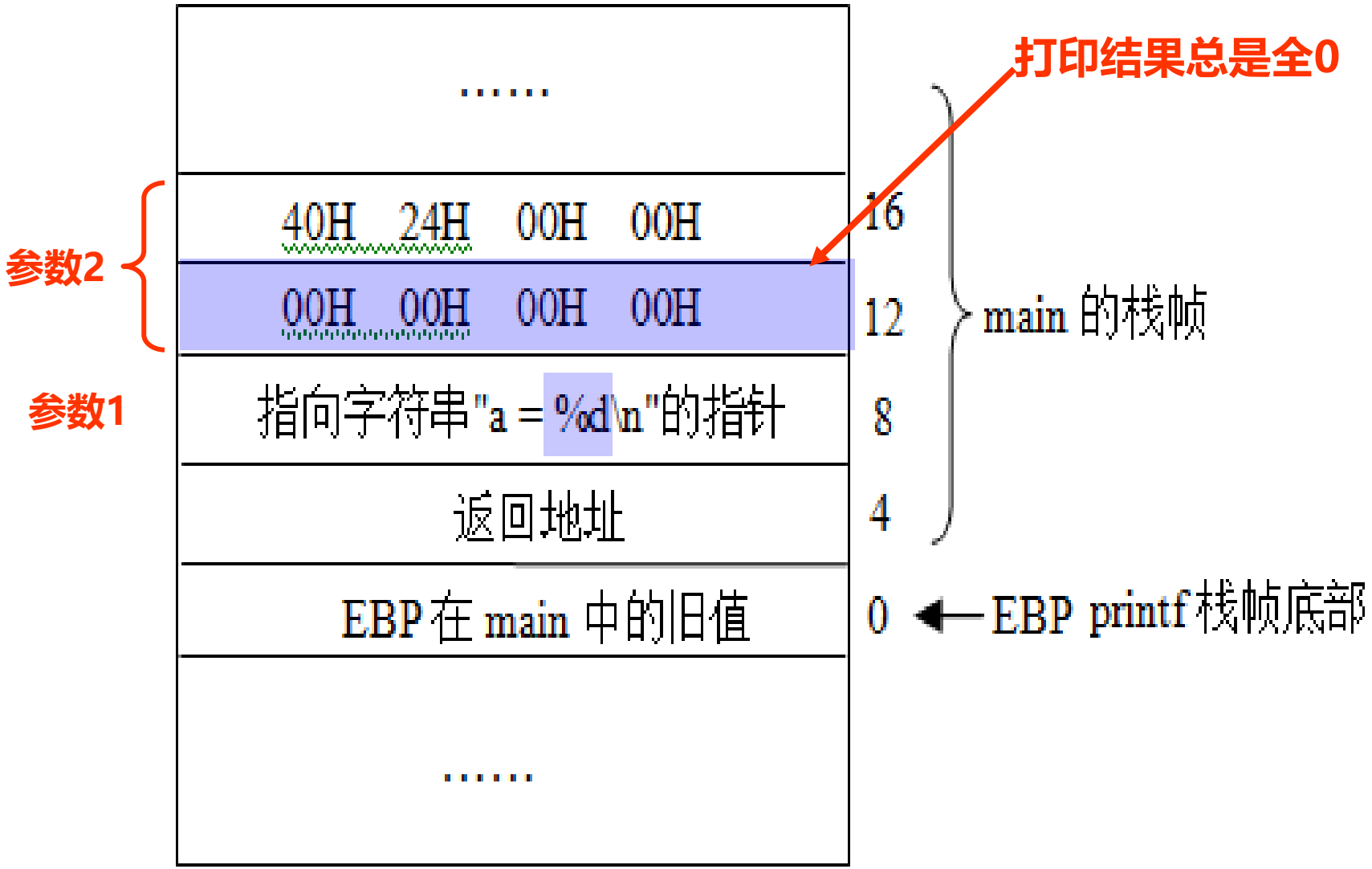
fldl：局部变量区→ST(0)

fstpl：ST(0) →参数区

在IA-32中a为float型又怎样呢？先执行flds，再执行fstpl

即：flds将32位单精度转换为80位格式入浮点寄存器栈，fstpl再将80位转换为64位送存储器栈中，故实际上与a是double效果一样！

IA-32过程调用参数传递



a的机器数对应十六进制为：40 24 00 00 00 00 00 00H

X86-64过程调用参数传递

```
main()
{
    double a = 10;
    printf("a = %d\n", a);
}
```

操作数宽度（字节）	入口参数						返回参数
	1	2	3	4	5	6	
8	RDI	RSI	RDX	RCX	R8	R9	RAX
4	EDI	ESI	EDX	ECX	R8D	R9D	EAX
2	DI	SI	DX	CX	R8W	R9W	AX
1	DIL	SIL	DL	CL	R8B	R9B	AL

```
.LC1:
        .string "a = %d\n "
......
movsd   .LC0(%rip), %xmm0 //a送xmm0
movl    $.LC1, %edi //RDI 高32位为0
movl    $1, %eax    //向量寄存器个数
call     printf
addq    $8, %rsp
ret
......
.LC0:
    .long   0
    .long   1076101120
```

因为printf第2个参数为double型，故向量寄存器个数为1

00000000H

40240000H

小端方式！0存在低地址上

printf中为%d，故将从ESI中取打印参数进行处理；但a是double型数据，在x86-64中，a的值被送到XMM寄存器中而不会送到ESI中。故在printf执行时，从ESI中读取的并不是a的低32位，而是一个不确定的值。

选择结构的机器级表示

- if ~ else语句的机器级表示

```
if (cond_expr)
    then_statement
else
    else_statement
```

```
    c=cond_expr;
    if (!c)
        goto false_label;
    then_statement
    goto done;
false_label:
    else_statement
done:
```

Jcc指令

JMP指令

```
    c=cond_expr;
    if (c)
        goto true_label;
    else_statement
    goto done;
true_label:
    then_statement
done:
```

If-else语句举例

```
int  get_cont( int *p1, int *p2 )
{
      if  ( p1 > p2 )
            return *p2;
      else
            return *p1;
}
```

p1和p2对应实参的存储地址分别为R[ebp]+8、R[ebp]+12，EBP指向当前栈帧底部，结果存放在EAX。

为何这里是”jbe”指令?

```
  movl  8(%ebp), %eax    //R[eax]←M[R[ebp]+8], 即 R[eax]=p1
  movl  12(%ebp), %edx   //R[edx]←M[R[ebp]+12], 即 R[edx]=p2
  cmpl  %edx, %eax       //比较 p1 和 p2, 即根据p1-p2结果置标志
  jbe   .L1              //若 p1<=p2, 则转 L1 处执行
  movl  (%edx), %eax     //R[eax]←M[R[edx]], 即 R[eax]=M[p2]
  jmp   .L2              //无条件跳转到 L2 执行
.L1:
  movl  (%eax), %eax     // R[eax]←M[R[eax]], 即 R[eax]=M[p1]
.L2
```

switch-case语句举例

```
int sw_test(int a, int b, int c)
{
   int result;
   switch(a) {
   case 15:
      c=b&0x0f;
   case 10:
      result=c+50;
      break;
   case 12:
   case 17:
      result=b+50;
      break;
   case 14:
      result=b
      break;
   default:
      result=a;
   }
   return result;
}
```

a在10和17之间

```
  movl  8(%ebp), %eax
  subl  $10, %eax
  cmpl  $7, %eax
  ja    .L5
  jmp   *.L8( , %eax, 4)
.L1:
  movl  12(%ebp), %eax
  andl  $15, %eax
  movl  %eax, 16(%ebp)
.L2:
  movl  16(%ebp), %eax
  addl  $50, %eax
  jmp  .L7
.L3:
  movl  12(%ebp), %eax
  addl  $50, %eax
  jmp   .L7
.L4:
  movl  12(%ebp), %eax
  jmp  .L7
.L5:
  addl  $10, %eax
.L7:
```

R[eax]=a-10=i

if (a-10)>7 转 L5

转.L8+4*i 处的地址

跳转表在目标文件的只读节中，按4字节边界对齐。

```
  .section  .rodata
  .align 4
.L8                a=
  .long   .L2  10
  .long   .L5  11
  .long   .L3  12
  .long   .L5  13
  .long   .L4  14
  .long   .L1  15
  .long   .L5  16
  .long   .L3  17
```

循环结构的机器级表示

- do~while循环的机器级表示

```
do  loop_body_statement
      while (cond_expr);
```

```
loop:
    loop_body_statement
    c=cond_expr;
    if (c) goto loop;
```

红色处为条件转移指令！

不一定有无条件跳转指令

- for循环的机器级表示

```
for (begin_expr; cond_expr; update_expr)
        loop_body_statement
```

- while循环的机器级表示

```
while (cond_expr)
      loop_body_statement
```

```
    c=cond_expr;
    if (!c) goto done;
loop:
    loop_body_statement
    c=cond_expr;
    if (c) goto loop;
done:
```

```
        begin_expr;
        c=cond_expr;
        if (!c) goto done;
    loop:
        loop_body_statement
        update_expr;
        c=cond_expr;
        if (c) goto loop;
    done:
```

循环结构与递归的比较

递归函数nn_sum仅为说明原理，实际上可直接用公式，为说明循环的机器级表示，这里用循环实现。

```
int  nn_sum ( int n)
{
        int i;
        int result=0;
        for (i=1; i <=n; i++)
            result+=i;
        return result;
}
```

```
movl  8(%ebp), %ecx
movl  $0, %eax
movl  $1, %edx
cmpl  %ecx, %edx
 jg    .L2
.L1:
addl  %edx, %eax
addl  $1, %edx
cmpl  %ecx, %edx
 jle   .L1
.L2
```

局部变量 i 和 result 被分别分配在EDX和EAX中。

通常复杂局部变量被分配在栈中，而这里都是简单变量

SKIP

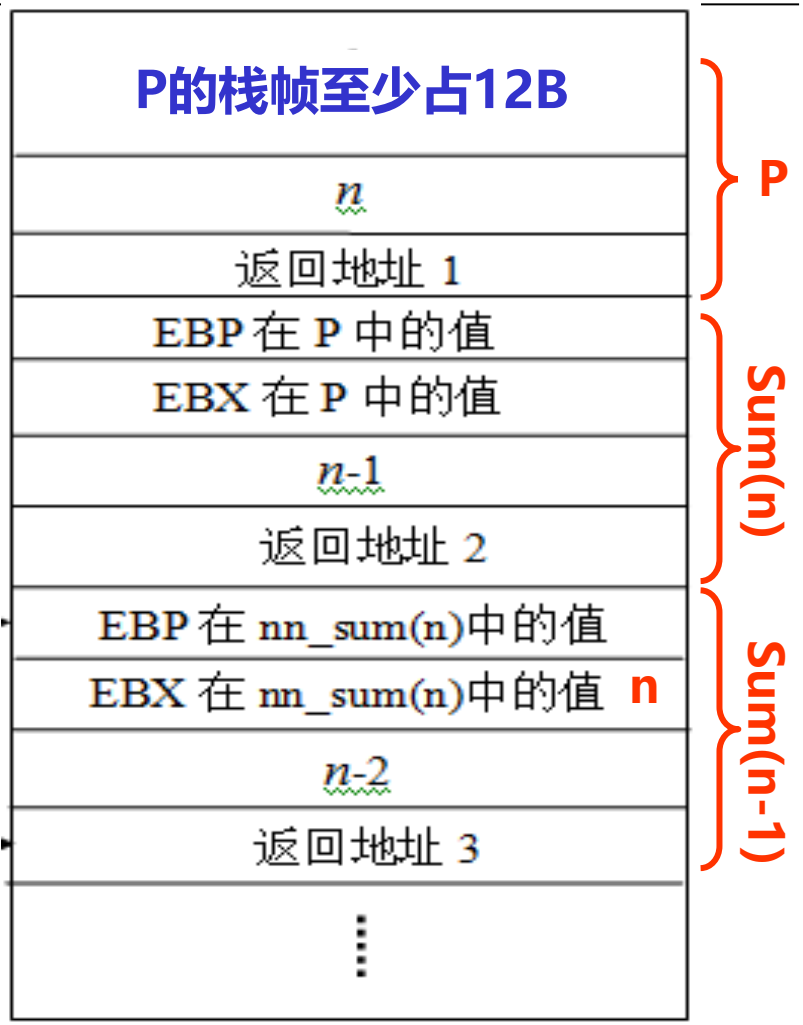
过程体中没用到被调用过程保存寄存器。因而，该过程栈帧中仅需保留EBP，即其栈帧仅占用4字节空间，而递归方式则占用了(16n+12)字节栈空间，多用了(16n+8)字节，每次递归调用都要执行16条指令，一共多了n次过程调用，因而，递归方式比循环方式至少多执行了16n条指令。由此可以看出，为了提高程序的性能，若能用非递归方式执行则最好用非递归方式。

递归过程调用举例

```
int  nn_sum ( int n)
{
        int result;
        if  (n<=0 )
            result=0;
        else
            result=n+nn_sum(n-1);
        return  result;
}
```

```
 pushl   %ebp
 movl    %esp, %ebp
 pushl    %ebx
 subl    $4, %esp
 movl    8(%ebp), %ebx
 movl    $0, %eax
 cmpl    $0, %ebx
 jle     .L2
 leal    -1(%ebx), %eax
 movl    %eax, (%esp)
 call    nn_sum
 addl    %ebx, %eax
.L2
 addl    $4, %esp
 popl    %ebx
 popl    %ebp
 ret
```

BACK



时间开销：每次递归执行16条指令，共16n条指令

空间开销：一次调用增加16B栈帧，共16n+12

逆向工程举例

```
int function_test( unsigned x)
{
    int result=0;
    int i;
    for ( ____①____ ; ____②____ ; ____③____ ) {
            ____________④____________;
     }
     return result;
}
```

```
    movl   8(%ebp), %ebx
    movl   $0, %eax
    movl   $0, %ecx
.L12:
    leal   (%eax,%eax), %edx
    movl   %ebx, %eax
    andl   $1, %eax
    orl    %edx, %eax
    shrl   %ebx
    addl   $1, %ecx
    cmpl   $32, %ecx
    jne    .L12
```

该函数有几个参数? 1个

处理结构是怎样的? 循环结构!

① 处为i=0，② 处为i≠32，③ 处为i++。

入口参数 x 在EBX中，返回参数 result 在EAX中。LEA实现“2*result”，即：将result左移一位；第6和第7条指令则实现“x&0x01”；第8条指令实现“result=(result<<1) | (x & 0x01)”，第9条指令实现“x>>=1”。综上所述，④ 处的C语言语句是“result=(result<<1) | (x & 0x01); x>>=1;”。

程序的机器级表示

- 分以下五个部分介绍
  - 第一讲：程序转换概述
    - 机器指令和汇编指令
    - 机器级程序员感觉到的属性和功能特性
    - 高级语言程序转换为机器代码的过程
  - 第二讲：IA-32 /x86-64指令系统
  - 第三讲： C语言程序的机器级表示
    - 过程调用的机器级表示
    - 选择语句的机器级表示
    - 循环结构的机器级表示
  - 第四讲：复杂数据类型的分配和访问
    - 数组的分配和访问
    - 结构体数据的分配和访问
    - 联合体数据的分配和访问
    - 数据的对齐
  - 第五讲：越界访问和缓冲区溢出

从高级语言程序出发，用其对应的机器级代码以及内存（栈）中信息的变化来说明底层实现

围绕C语言中的语句和复杂数据类型，解释其在底层机器级的实现方法

数组的分配和访问

- **数组元素在内存的存放和访问**
  - **例如，定义一个具有4个元素的静态存储型 short 数据类型数组A，可以写成“static short A[4];”**
  - **第 i（0≤i≤3）个元素的地址计算公式为&A[0]+2*i。**
  - **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 **比例因子是2!**

  其中，ECX为变址（索引）寄存器，在循环体中增量

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

数组元素在内存的存放和访问

- 分配在静态区的数组的初始化和访问

```
int buf[2] = {10, 20};
int main ( )
{
    int i, sum=0;
    for (i=0; i<2; i++)
        sum+=buf[i];
    return sum;
}
```

buf是在静态区分配的数组，链接后，buf在可执行目标文件的可读写数据段中分配了空间

```
08049908 <buf>:
08049908:   0A 00 00 00 14 00 00 00
```

此时，buf=&buf[0]=0x08049908

编译器通常将其先存放到寄存器(如EDX)中

假定 i 被分配在ECX中，sum被分配在EAX中，则“sum+=buf[i];”和 i++ 可用什么指令实现?

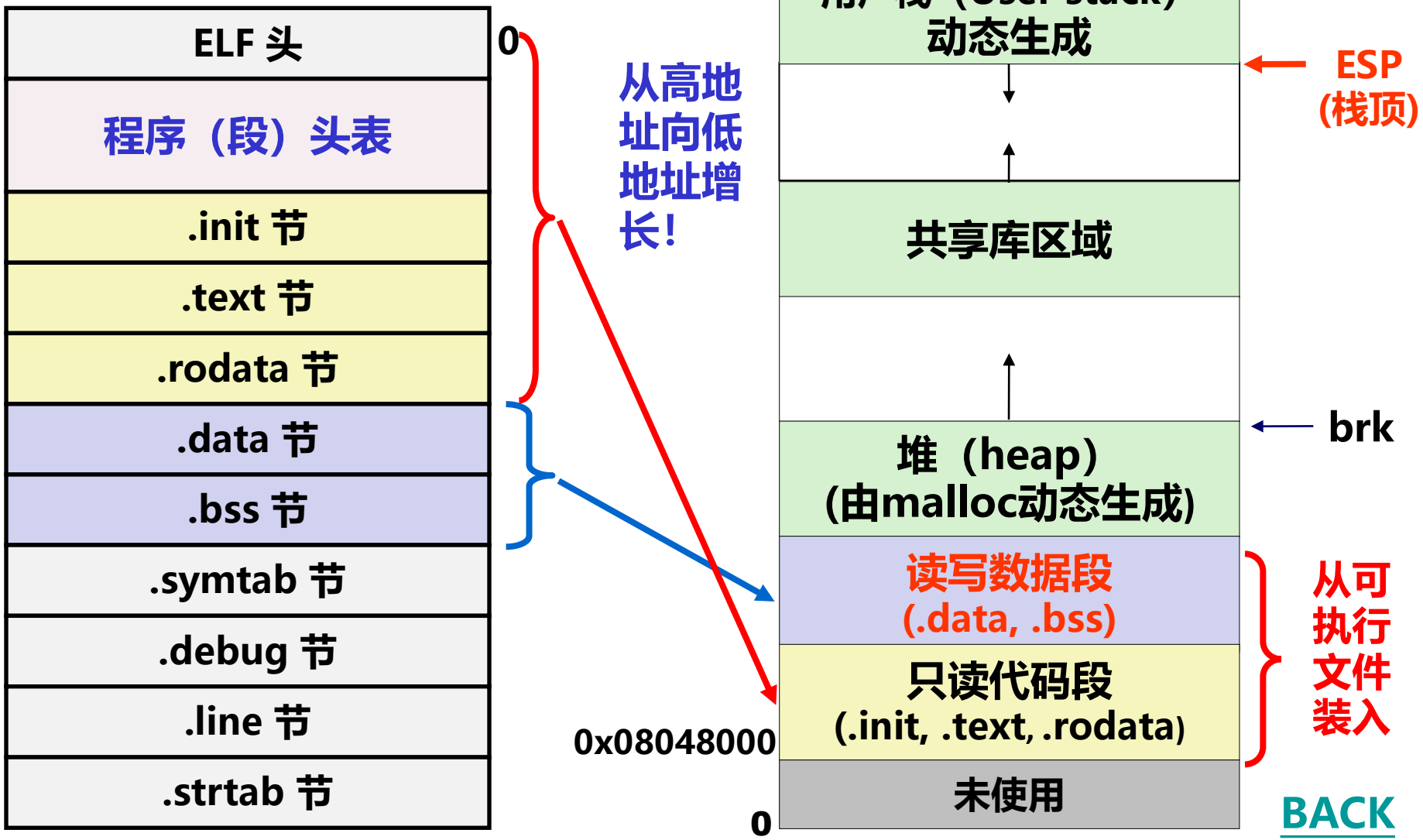
addl buf(, %ecx, 4), %eax 或 addl 0(%edx , %ecx, 4), %eax

addl $1, %ecx

SKIP

可执行文件的存储器映像

程序(段)头表描述如何映射



BACK

数组元素在内存的存放和访问

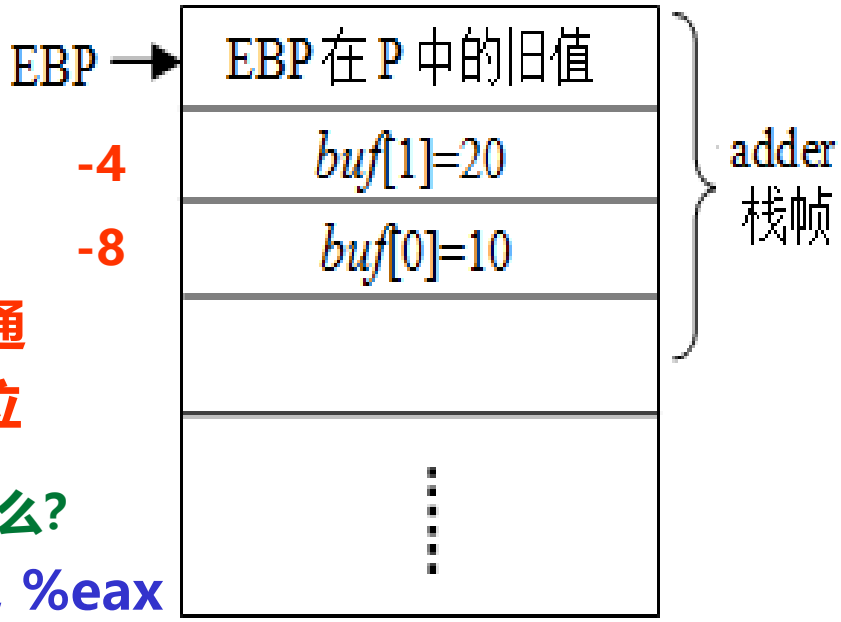
- auto型数组的初始化和访问

```
int adder ( )
{
    int buf[2] = {10, 20};
    int i, sum=0;
    for (i=0; i<2; i++)
        sum+=buf[i];
    return sum;
}
```

分配在栈中，故数组首址通过EBP来定位

EDX、ECX各是什么?

```
addl (%edx, %ecx, 4), %eax
```



对buf进行初始化的指令是什么?

```
movl $10, -8(%ebp)    //buf[0]的地址为R[ebp]-8，将10赋给buf[0]
movl $20, -4(%ebp)    //buf[1]的地址为R[ebp]-4，将20赋给buf[1]
```

若buf首址在EDX中，则获得buf首址的对应指令是什么?

```
leal -8(%ebp), %edx  //buf[0]的地址为R[ebp]-8，将buf首址送EDX
```

数组元素在内存的存放和访问

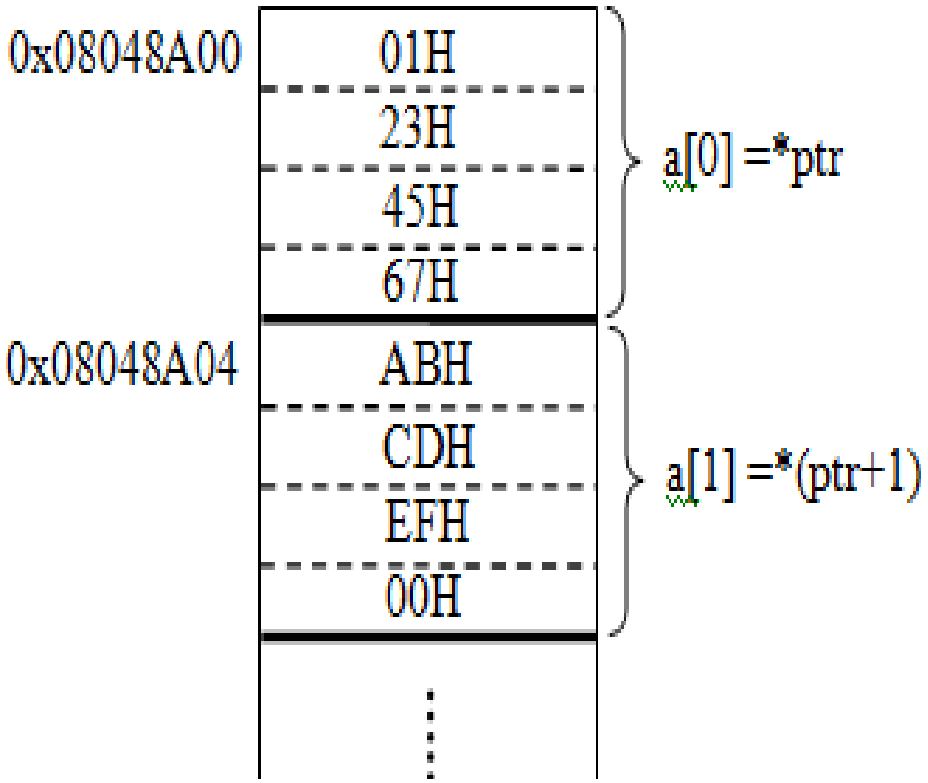
- **数组与指针**

✓ **在指针变量目标数据类型与数组类型相同的前提下，指针变量可以指向数组或数组中任意元素**

✓ **以下两个程序段功能完全相同，都是使ptr指向数组a的第0个元素a[0]。a的值就是其首地址，即 a=&a[0]，因而 a=ptr，从而有 &a[i]=ptr+i=a+i 以及 a[i]=ptr[i]=*(ptr+i)=*(a+i)。**

(1) int a[10];
int *ptr=&a[0];

(2) int a[10], *ptr;
ptr=&a[0];



小端方式下a[0]=?,a[1]=?
a[0]=0x67452301, a[1]=0x0efcdab
数组首址0x8048A00在ptr中，ptr+i并不是用0x8048A00加 i 得到，而是等于0x8048A00+4*i

数组元素在内存的存放和访问

- **数组与指针**

序号	表达式	类型	值的计算方式	汇编代码
1	A	int *		
2	A[0]	int		
3	A[i]	int		
4	&A[3]	int *		
5	&A[i]-A	int		
6	*(A+i)	int		
7	*(&A[0]+i-1)	int		
8	A+i	int *		

问题：

假定数组A的首址SA在ECX中，i在EDX中，表达式结果在EAX中，各表达式的计算方式以及汇编代码各是什么？

2、3、6和7对应汇编指令都需访存，指令中源操作数的寻址方式分别是“基址”、“基址加比例变址”、“基址加比例变址”和“基址加比例变址加位移”的方式，因为数组元素的类型为int型，故比例因子为4。

数组元素在内存的存放和访问

- **数组与指针** **假设A首址SA在ECX，i 在EDX，结果在EAX**

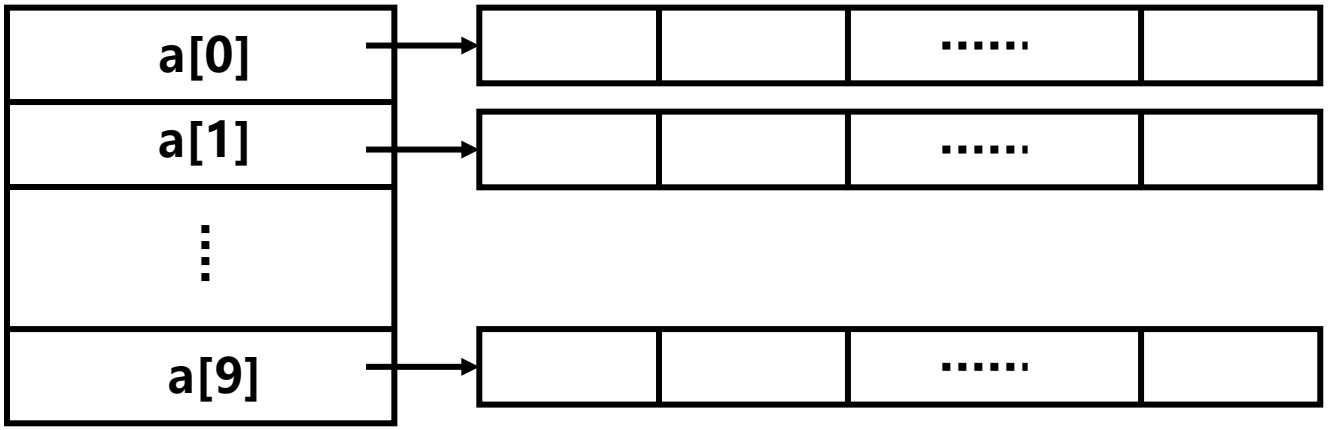
序号	表达式	类型	值的计算方式	汇编代码
1	A	int *	*SA*	leal (%ecx), %eax
2	A[0]	int	M[*SA*]	movl (%ecx), %eax
3	A[i]	int	M[*SA*+4**i*]	movl (%ecx, %edx, 4), %eax
4	&A[3]	int *	*SA*+12	leal 12(%ecx), %eax
5	&A[i]−A	int	(*SA*+4**i*−*SA*)/4=*i*	movl %edx, %eax
6	*(A+i)	int	M[*SA*+4**i*]	movl (%ecx, %edx, 4), %eax
7	*(&A[0]+i−1)	int	M[*SA*+4**i*−4]	movl −4(%ecx, edx, 4), %eax
8	A+i	int *	*SA*+4**i*	leal (%ecx, %edx, 4), %eax

2、3、6和7对应汇编指令都需访存，指令中源操作数的寻址方式分别是“基址”、“基址加比例变址”、“基址加比例变址”和“基址加比例变址加位移”的方式，因为数组元素的类型为int型，故比例因子为4 。

数组元素在内存的存放和访问

- 指针数组和多维数组
  - 由若干指向同类目标的指针变量组成的数组称为指针数组。
  - 其定义的一般形式如下：

    存储类型 数据类型 *指针数组名[元素个数]；
  - 例如，“int *a[10];”定义了一个指针数组a，它有10个元素，每个元素都是一个指向int型数据的指针。
    - 一个指针数组可以实现一个二维数组。



数组元素在内存的存放和访问

- **指针数组和多维数组**　　按行优先方式存放数组元素　　SKIP
  - 计算一个两行四列整数矩阵中每一行数据的和。

```
main ( )
{
   static short num[ ][4]={ {2, 9, -1, 5},
                            {3, 8, 2, -6}};
   static short *pn[ ]={num[0], num[1]};
   static short s[2]={0, 0};
   int i, j;
   for (i=0; i<2; i++) {
      for (j=0; j<4; j++)
           s[i]+=*pn[i]++;
      printf (sum of line %d:  %d\n" , i+1, s[i]);
   }
}
```

当i=1时，pn[i]=*(pn+i)=M[pn+4*i]=0x8049308

若处理“s[i]+=*pn[i]++;”时 i 在ECX，s[i]在AX，pn[i]在EDX，则对应指令序列可以是什么?

```
movl   pn(,%ecx,4), %edx
addw   (%edx), %ax
addl   $2, pn(, %ecx, 4)
```

pn[i]+”1”→pn[i]

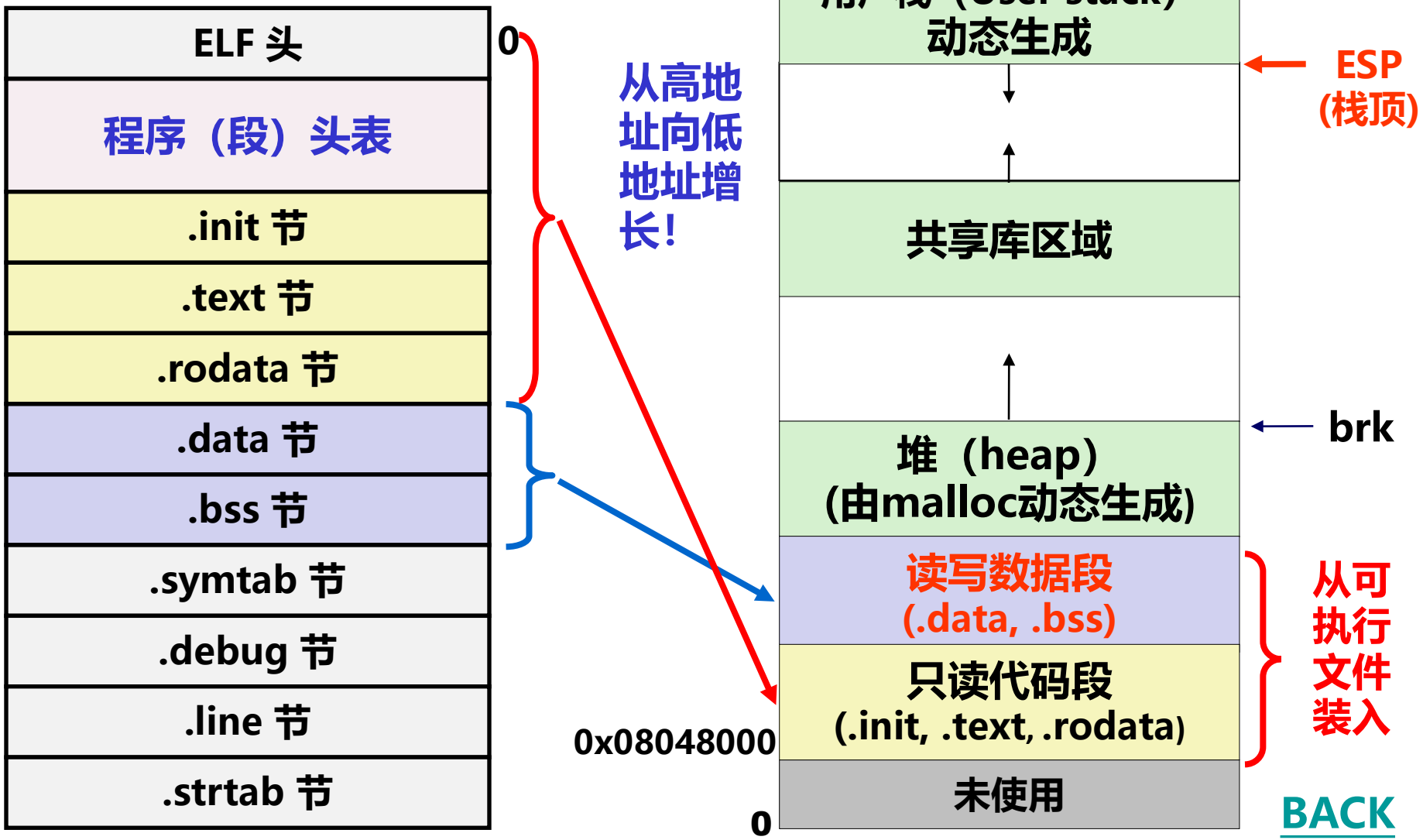
若num=0x8049300,则num、pn和s在存储区中如何存放?

```
08049300 <num>:    num=num[0]=&num[0][0]=0x8049300
08049300:    02 00 09 00 ff ff 05 00 03 00 08 00 02 00 fa ff
08049310 <pn>:
08049310:    00 93 04 08 08 93 04 08
08049318<s>:
08049318:    00 00 00 00
```

pn=&pn[0]=0x8049310
pn[0]=num[0]=0x8048300
pn[1]=num[1]=0x8048308

可执行文件的存储器映像

程序(段)头表描述如何映射



BACK

结构体数据的分配和访问

- 结构体成员在内存的存放和访问
  - 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

```
struct cont_info {
        char id[8];
        char name [12];
        unsigned post;
        char address[100];
        char phone[20];
};
```

若变量x分配在地址0x8049200开始的区域，那么
x=&(x.id)=0x8049200 (若x在EDX中)
&(x.name)= 0x8049200+8=0x8049208
&(x.post)= 0x8049200+8+12=0x8049214
&(x.address)=0x8049200+8+12+4=0x8049218
&(x.phone)=0x8049200+8+12+4+100=0x804927C

struct cont_info x={ “0000000”, “ZhangS”, 210022, “273 long street, High Building #3015”, “12345678”};

x初始化后，在地址0x8049208到0x804920D处是字符串“ZhangS”，0x804920E处是字符‘\0’，从0x804920F到0x8049213处都是空字符。

“unsigned xpost=x.post;”对应汇编指令为“movl 20(%edx), %eax”

结构体数据的分配和访问

- **结构体数据作为入口参数**

按地址调用

```
void stu_phone1 ( struct cont_info *s_info_ptr)          stu_phone1(&x)
{
   printf ( "%s phone number: %s", (*s_info_ptr).name, (*s_info_ptr).phone);
}
void stu_phone2 ( struct cont_info s_info)    按值调用  stu_phone2(x)
{
   printf ( "%s phone number: %s", s_info.name, s_info.phone);
}
```

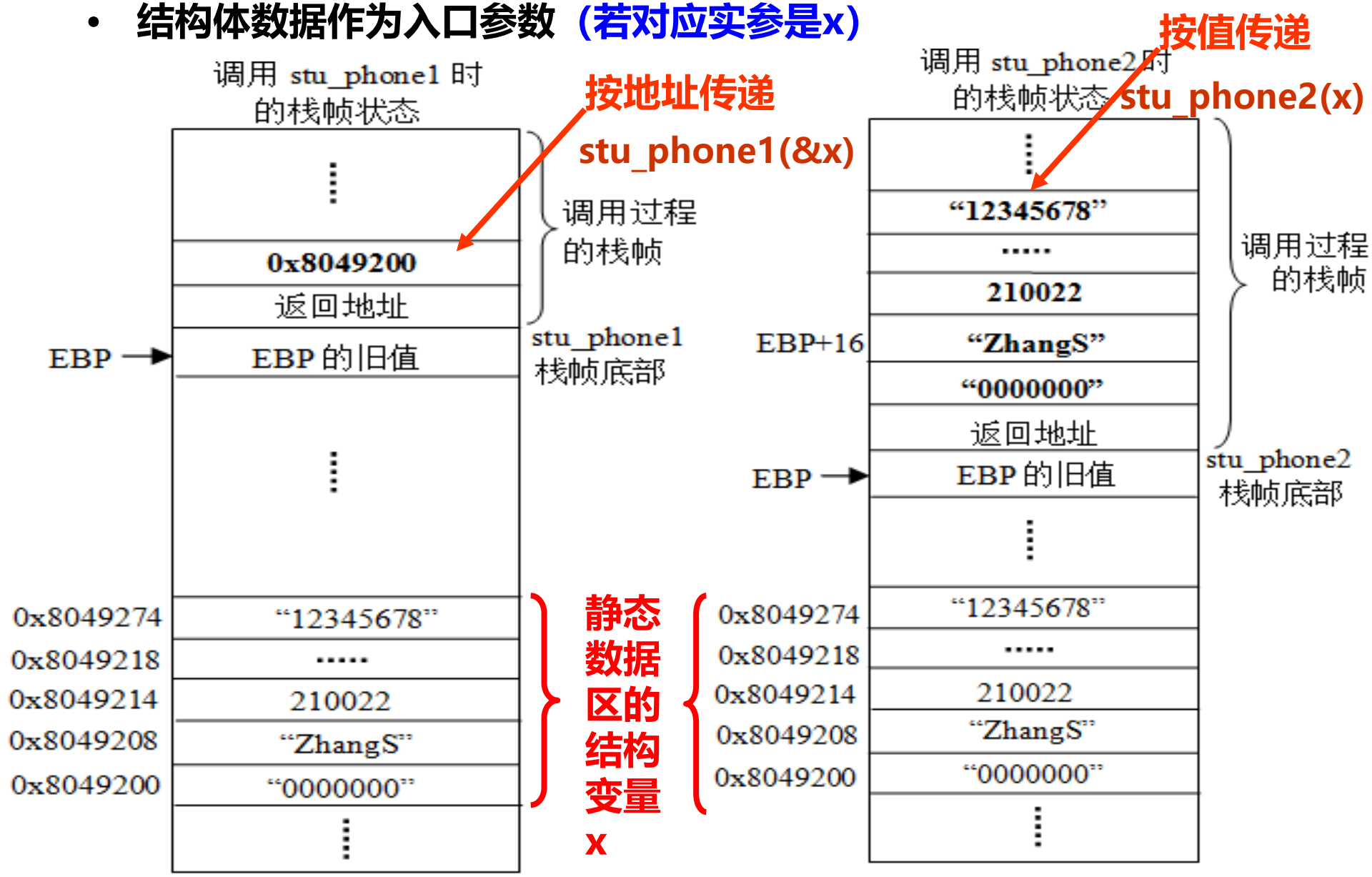
  - 当结构体变量需要作为一个函数的形参时，形参和调用函数中的实参应具有相同结构

    struct cont_info x={ "0000000", "ZhangS", 210022, "273 long street, High Building #3015", "12345678" };

  - 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，且更新后的数据无法在调用过程使用(如前面的swap(a,b)例子)
  - 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

结构体数据的分配和访问

- **结构体数据作为入口参数（若对应实参是x）**



结构体数据的分配和访问

- **按地址传递参数** stu_phone1(&x)

  (*s_info_ptr).name可写成s_info_ptr->name，

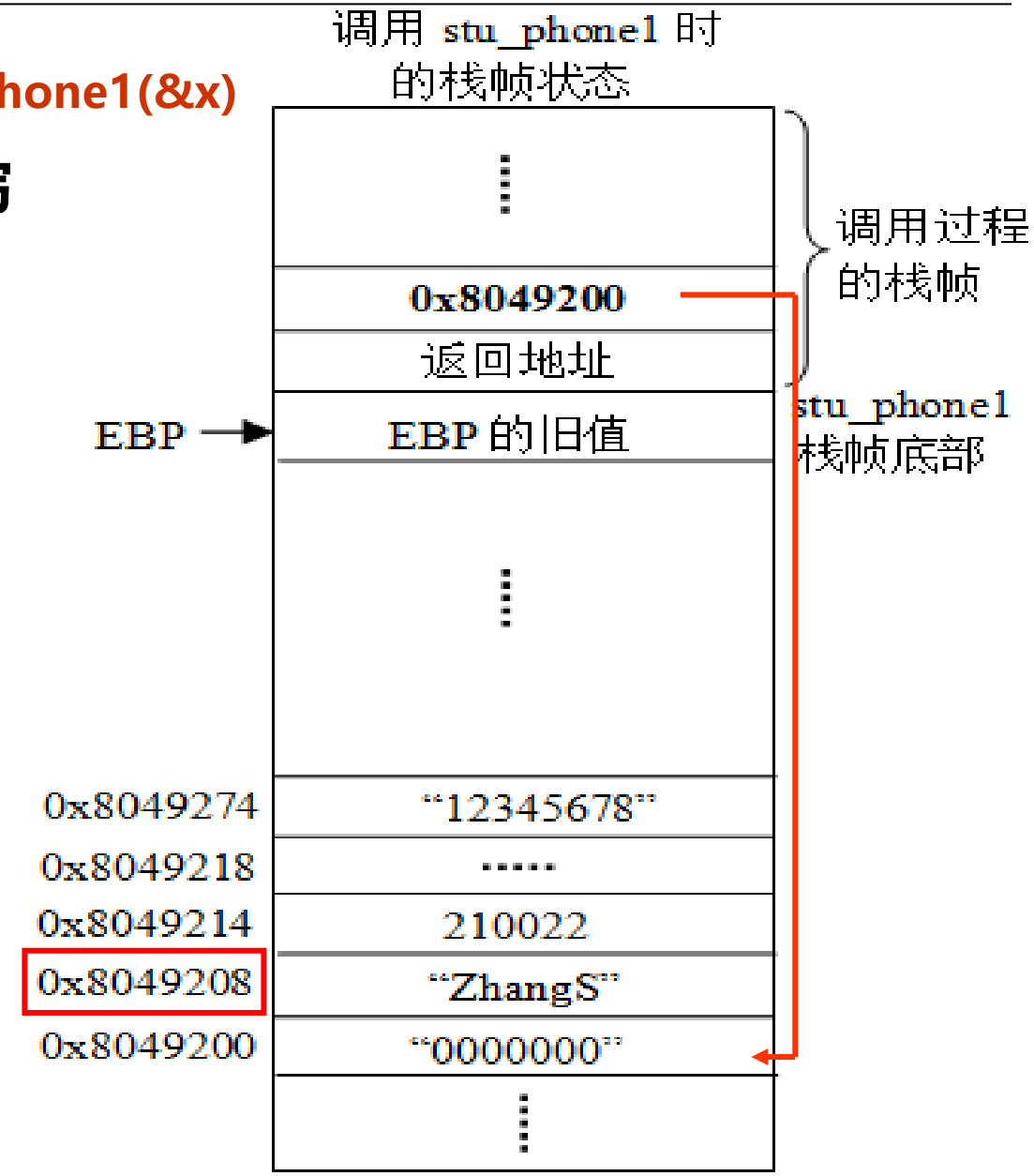
  执行以下两条指令后:

  movl 8(%ebp), edx

  leal 8(%edx), eax

  EAX中存放的是字符串“ZhangS”在静态存储区内的首地址

  0x8049208



结构体数据的分配和访问

- **按值传递参数** stu_phone2(x)

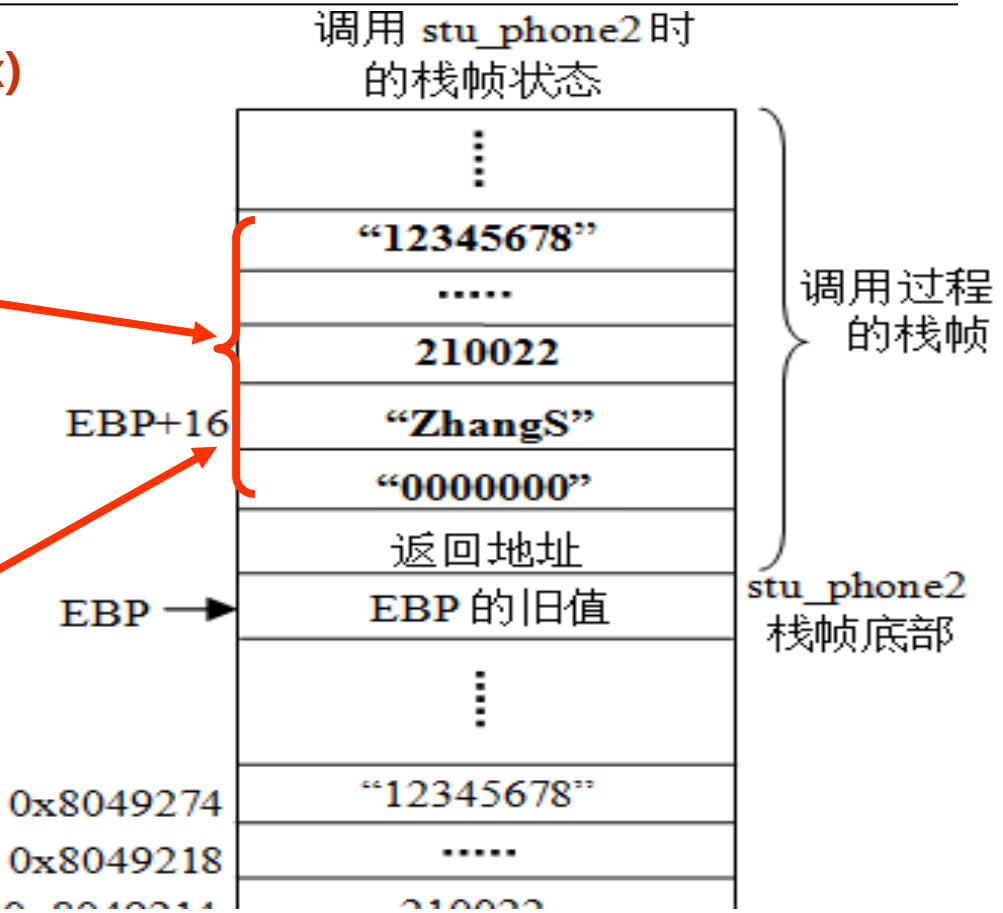
  x所有成员值作为实参存到参数区。

  s_info.name送EAX的指令序列为：

  leal 8(%ebp), edx

  leal 8(%edx), eax

  EAX中存放的是“ZhangS”的栈内参数区首址。



  - stu_phone1和stu_phone2功能相同，但后者开销大，因为它需对结构体成员整体从静态区复制到栈中，需要很多条mov或其他指令，从而执行时间更长，并占更多栈空间和代码空间
  - 特别是，按值传递时，无法获得更新后的结果

联合体数据的分配和访问

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

```
union uarea {
        char  c_data;
        short s_data;
        int     i_data;
        long   l_data;
};
```

IA-32中编译时，long和int长度一样，故uarea所占空间为4个字节。而对于与uarea有相同成员的结构型变量来说，其占用空间大小至少有11个字节，对齐的话则占用更多。

- 通常用于特殊场合，如，当事先知道某种数据结构中的不同字段的使用时间是互斥的，就可将这些字段声明为联合，以减少空间。
- 但有时会得不偿失，可能只会减少少量空间却大大增加处理复杂性。

联合体数据的分配和访问

- 还可实现对相同位序列进行不同数据类型的解释

```
unsigned
float2unsign( float f)
{
    union {
            float f;
            unsigned u;
     } tmp_union;
     tmp_union.f=f;
     return tmp_union.u;
}
```

函数形参是float型，按值传递参数，因而传递过来的实参是float型数据，赋值给非静态局部变量（联合体变量成员）

过程体为:

```
movl 8(%ebp), %eax
movl %eax, -4(%ebp)
movl -4(%ebp) , %eax
```

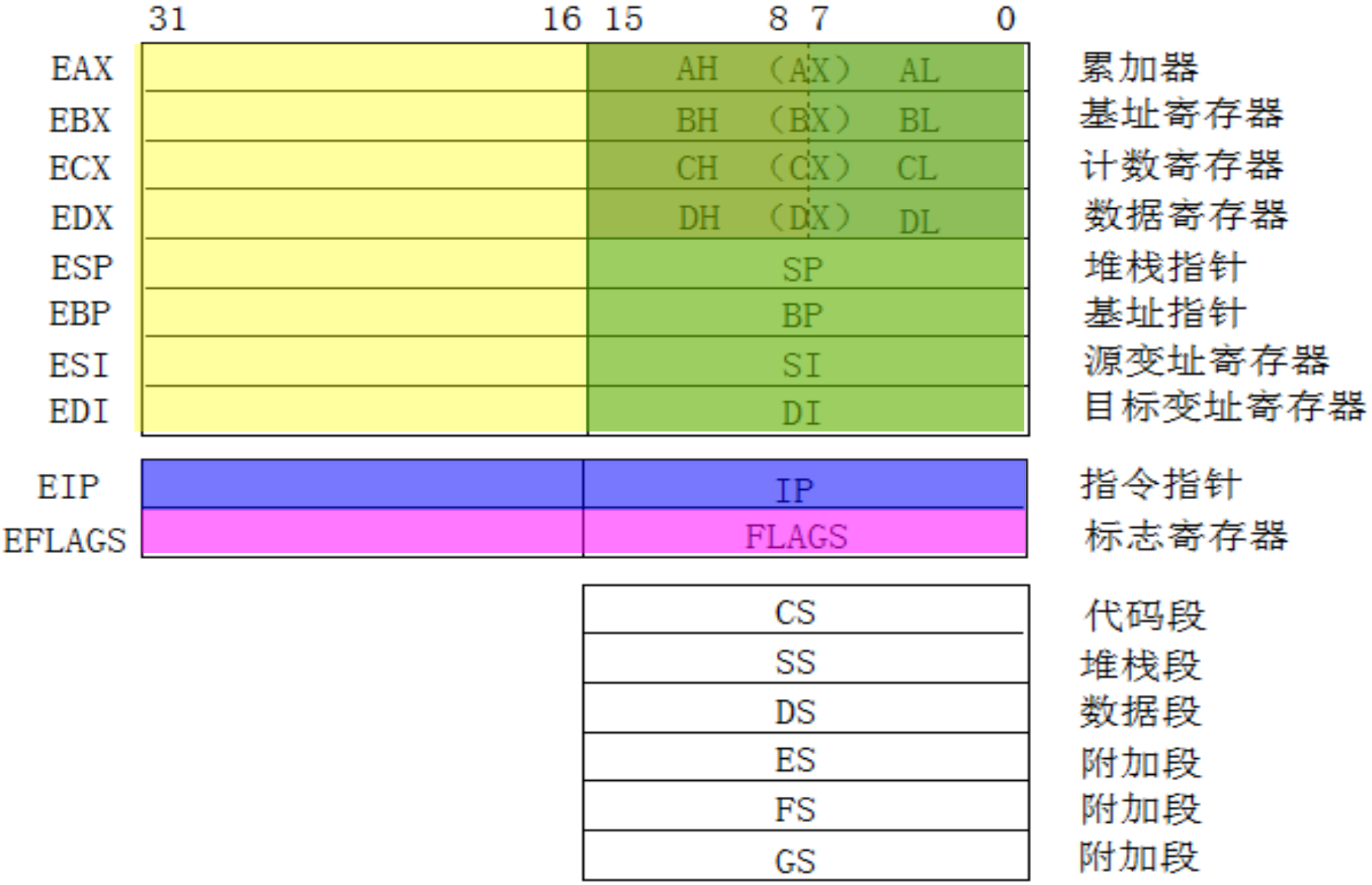
} 可优化掉!

将存放在地址R[ebp]+8处的入口参数 f 送到EAX（返回值）

问题：float2unsign(10.0)=?　$2^{30}+2^{24}+2^{21}=1092616192$

从该例可看出：机器级代码并不区分所处理对象的数据类型，不管高级语言中将其说明成float型还是int型或unsigned型，都把它当成一个0/1序列来处理。

IA-32的寄存器组织


31
16
15
8
7
0
EAX
AH
(AX)
AL
累加器
EBX
BH
(BX)
BL
基址寄存器
ECX
CH
(CX)
CL
计数寄存器
EDX
DH
(DX)
DL
数据寄存器
ESP
SP
堆栈指针
EBP
BP
基址指针
ESI
SI
源变址寄存器
EDI
DI
目标变址寄存器
EIP
IP
指令指针
EFLAGS
FLAGS
标志寄存器
CS
代码段
SS
堆栈段
DS
数据段
ES
附加段
FS
附加段
GS
附加段

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

如果要用C语言来模拟IA-32的寄存器组织，该如何做?

```
typedef struct{
union{
    struct {        uint32_t  eax;
                    uint32_t  ecx;
                    uint32_t  edx;
                    uint32_t  ebx;
                    uint32_t  esp;
                    uint32_t  ebp;
                    uint32_t  esi;
                    uint32_t  edi; };
    union {
                    uint32_t  _32;
                    uint16_t  _16;
                    uint8_t  _8[2];
            } gpr[8];
    };
swaddr_t  eip;
} CPU_state;

extern CPU_state cpu;
enum { R_EAX, R_ECX, R_EDX, R_EBX, R_ESP, R_EBP, R_ESI, R_EDI };
enum { R_AX, R_CX, R_DX, R_BX, R_SP, R_BP, R_SI, R_DI };
enum { R_AL, R_CL, R_DL, R_BL, R_AH, R_CH, R_DH, R_BH };
```

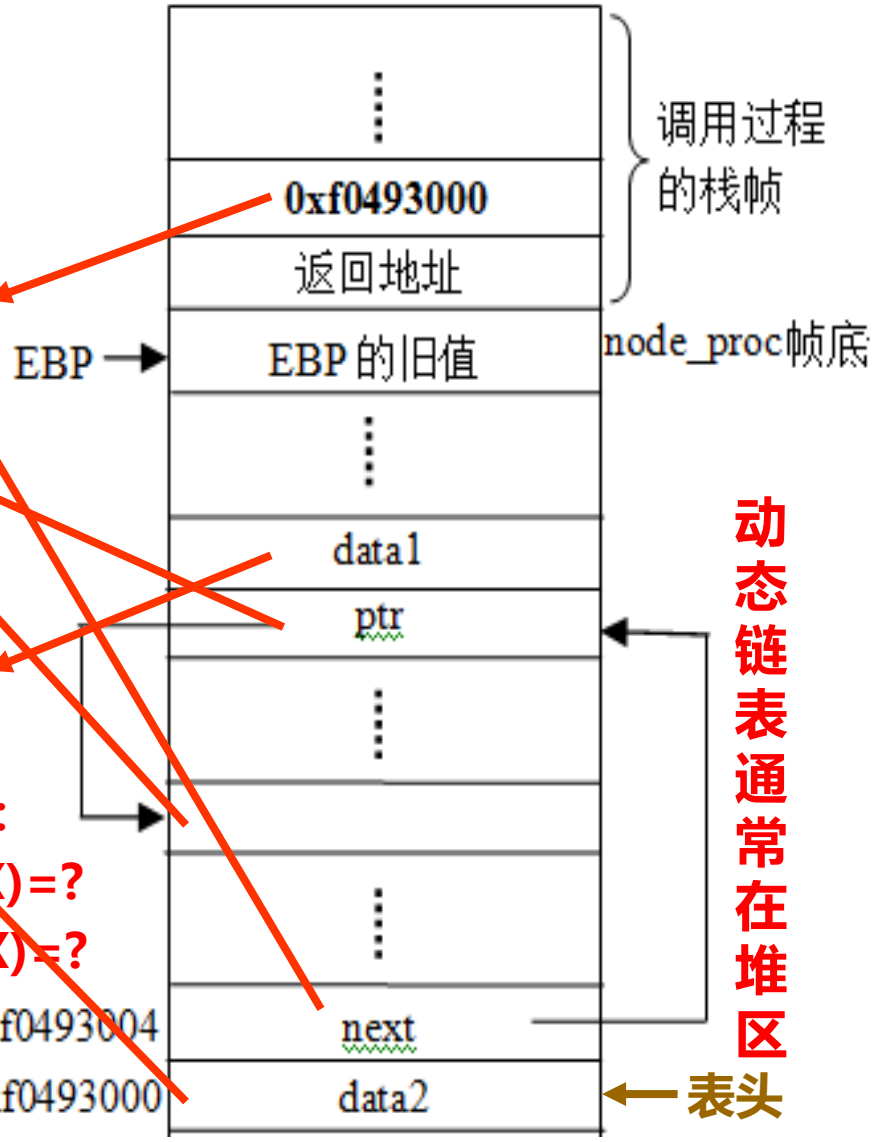
PA中模拟的IA-32的寄存器组织

联合体数据的分配和

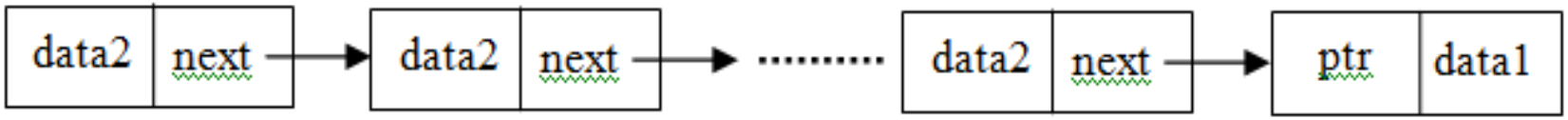
- **利用嵌套可定义链表结构**

```
union node {
    struct {
        int *ptr;
        int data1
    } node1;
    struct {
        int data2;
        union node *next;
    } node2;
};
```

```
movl  8(%ebp), %ecx
movl  4(%ecx), %edx
movl  (%edx), %eax
movl  (%eax), %eax
addl  (%ecx), %eax
movl  %eax, 4(%edx)
```




```
void node_proc ( union node *np) {
  np->node2.next->node1.data1=*(np->node2.next->node1.ptr)+np->node2.data2;
}
```

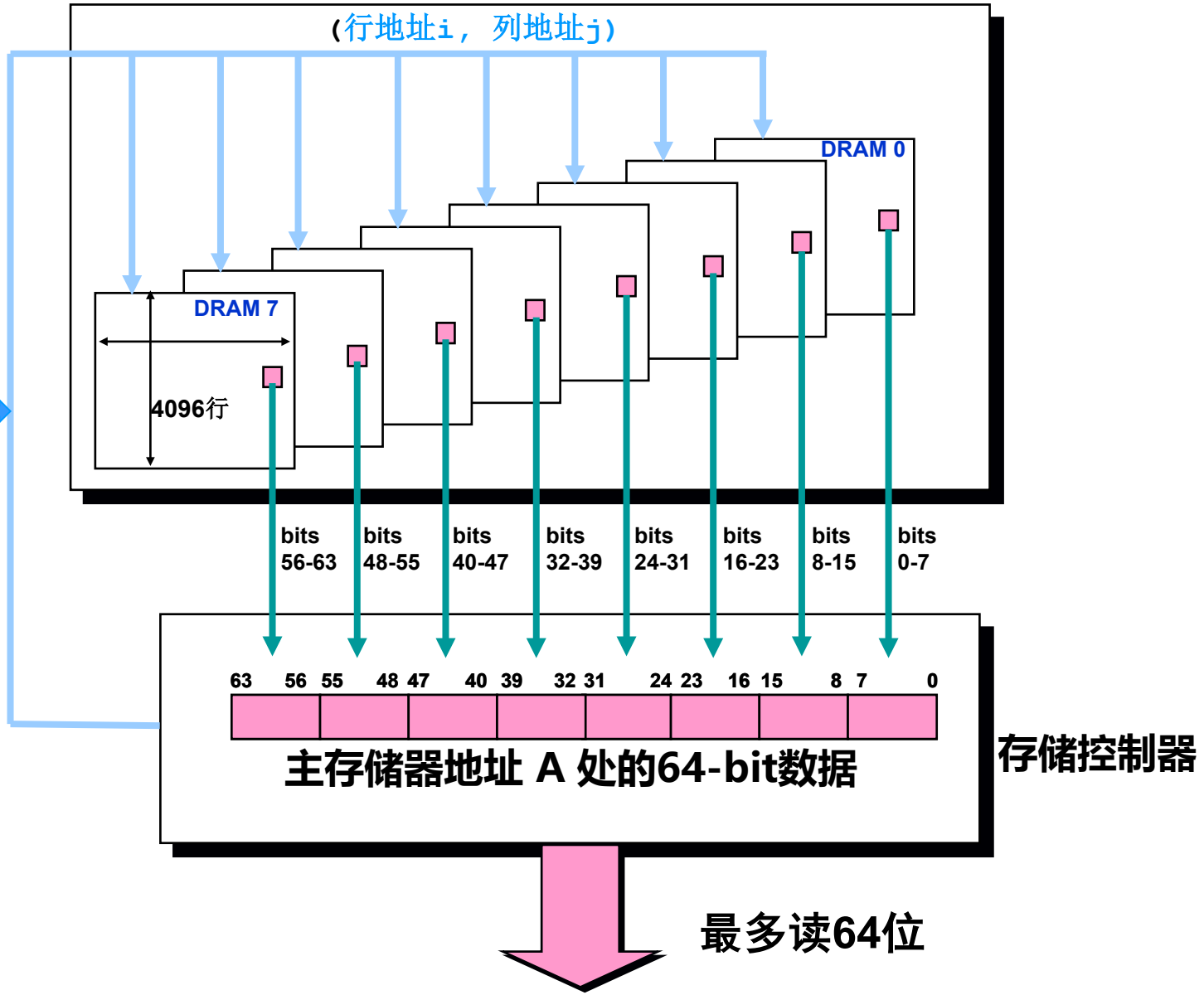


数据的对齐 SKIP

- CPU访问主存时只能一次读取或写入若干特定位
  - 例如，若每次最多读写64位，则第0-7字节可同时读写，第8-15字节可同时读写，……，以此类推
- 按边界对齐可使读写数据位于8i~8i+7(i=0,1,2,...) 单元内
- 最简单的对齐策略是，按其数据长度对齐。如，int型地址是4的倍数，short型地址是2的倍数，double和long long型则8的倍数，float型是4的倍数，char不对齐
- Windows遵循的ABI规范采用上述简单对齐策略
- I386 System V ABI策略更宽松：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）。虽然IA-32中扩展精度（long double）为80位=10字节，但是，为了使随后相同类型按4字节边界对齐，在内存分配了12字节空间

主存储器的结构

按边界对齐，
可使读写数据
位于8i~8i+7
(i=0,1,2,...)
单元内



存储宽度为64位=8B，交叉编址！第0-7字节可同时读写，第8-15字节可同时读写，……，以此类推

Alignment(对齐)

BACK

如：int i, short k, double x, char c, short j,……

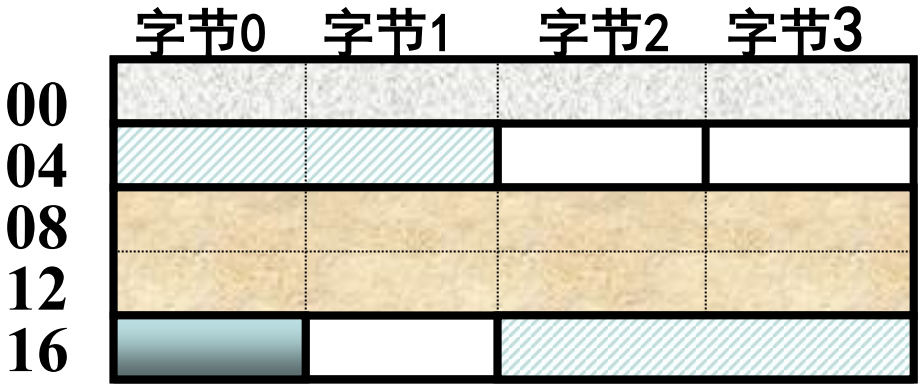
按字节编址

每次只能读写某个字地址开始的4个单元中连续的1个、2个、3个或4个字节

按边界对齐

x：2个周期

j：1个周期



则：&i=0; &k=4; &x=8; &c=16; &j=18;……

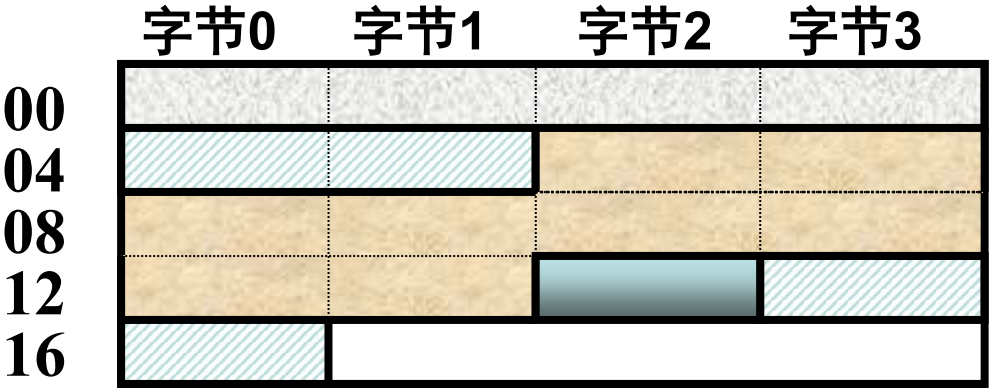
虽节省了空间，但增加了访存次数！

需要权衡，目前来看，浪费一点存储空间没有关系！

边界不对齐

x：3个周期

j：2个周期



则： &i=0; &k=4; &x=6; &c=14; &j=15;……

Windows中的对齐和分配顺序

```
#include  ……
int main()
{
    int a;
    char b;
    int c;
    printf(  “a: 0x%08x\n",&a);
    printf(  “b: 0x%08x\n",&b);
    printf(  “c: 0x%08x\n",&c);
    return 0;
}
```

a、b、c不一定按顺序分配，
但a和c的地址总是4的倍数，
b的地址则不一定是4的倍数。

用VC编译后的执行结果：

a: 0x0012ff7c

b: 0x0012ff7b

c: 0x0012ff80

顺序：b(1B)-a(4B)-c(4B)

用Dev-C++编译后的执行结果：

a: 0x0022ff7c

b: 0x0022ff7b

c: 0x0022ff74

顺序：c(4B)-隔3B-b(1B)-a(4B)

用lcc编译后的执行结果：

a: 0x0012ff6c

b: 0x0012ff6b

c: 0x0012ff64

顺序：同上（大地址->小地址）

数据的对齐

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。

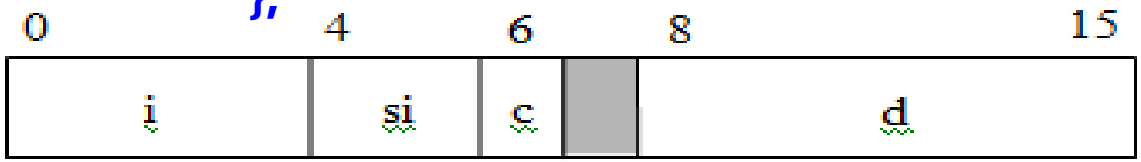
第③条规则是为了保证使结构体数组中的每个元素都能满足对齐要求

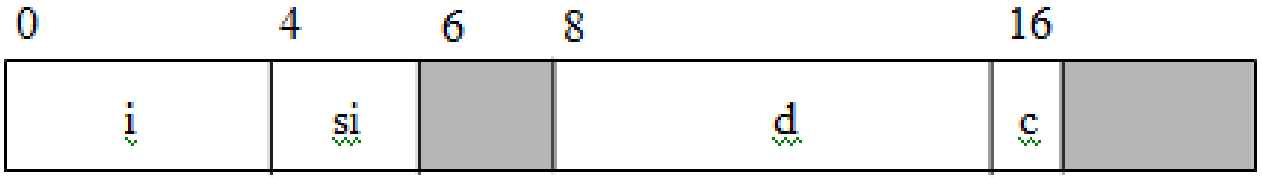
```
struct SDT {
    int     i;
    short   si;
    double  d;
    char    c;
} sa[10];
```

结构数组变量的最末可能需要插空，以使每个数组元素都按4字节边界对齐

```
struct SD {
    int     i;
    short   si;
    char    c;
    double  d;
};
```

结构变量首地址按4字节边界对齐





只要SD首址按4B边界对齐，所有字段都能按要求对齐

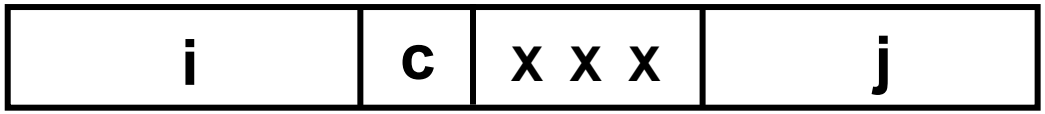
Alignment(对齐) 举例

例如，考虑下列两个结构声明：

```
struct  S1 {
        int     i;
        char    c;
        int     j;
};
```

```
struct  S2 {
        int     i;
        int     j;
        char    c;
};
```

S1：



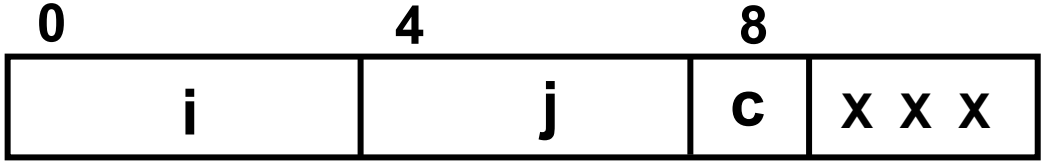
需要12个字节

S2：

至少需要9个字节

对于“struct S2 d[4]”，只分配9个字节能否满足对齐要求？不能!

S2：



需要12个字节

Alignment(对齐) 举例

Alignment(对齐)

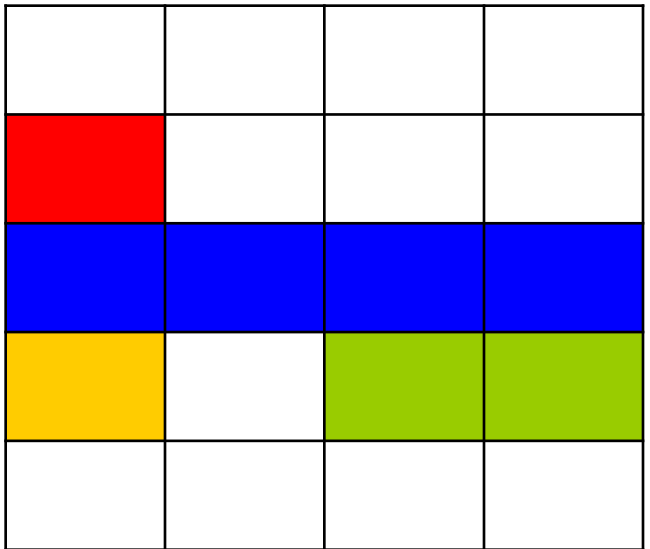
```
struct record {
        char   a;
        int    b;
        char   c;
        short  d;
 } ;
struct  record R[64];
```

在上述定义中，数组R占用多少字节？代码是否可以优化?

```
11:     R[0].a=1;
00401028 C6 45 E8 01          mov       byte ptr [ebp-18h],1
12:     R[0].b=2;
0040102C C7 45 EC 02 00 00 00 mov       dword ptr [ebp-14h],2
13:     R[0].c=3;
00401033 C6 45 F0 03          mov       byte ptr [ebp-10h],3
14:     R[0].d=4;
00401037 66 C7 45 F2 04 00    mov       word ptr [ebp-0Eh],offset main+2Bh (0040103b)
15:     R[1].a=5;
0040103D C6 45 F4 05          mov       byte ptr [ebp-0Ch],5
16:     R[1].b=6;
00401041 C7 45 F8 06 00 00 00 mov       dword ptr [ebp-8],6
17:     R[1].c=7;
00401048 C6 45 FC 07          mov       byte ptr [ebp-4],7
18:     R[1].d=8;
0040104C 66 C7 45 FE 08 00    mov       word ptr [ebp-2],offset main+40h (00401050)
```

地址: 0x0012FF48

```
0012FF20  CC CC CC CC CC CC CC CC
0012FF28  CC CC CC CC CC CC CC CC
0012FF30  01 CC CC CC 02 00 00 00
0012FF38  03 CC 04 00 05 CC CC CC
0012FF40  06 00 00 00 07 CC 08 00
0012FF48  88 FF 12 00 39 12 40 00
0012FF50  01 00 00 00 A0 0E 30 00
0012FF58  C0 0D 30 00 00 00 00 00
```



Alignment(对齐) 举例

Alignment(对齐)

```
struct record  {
  char        a;
  int         b;
  char        c;
  short       d;
 } ;
struct  record R[64];
```

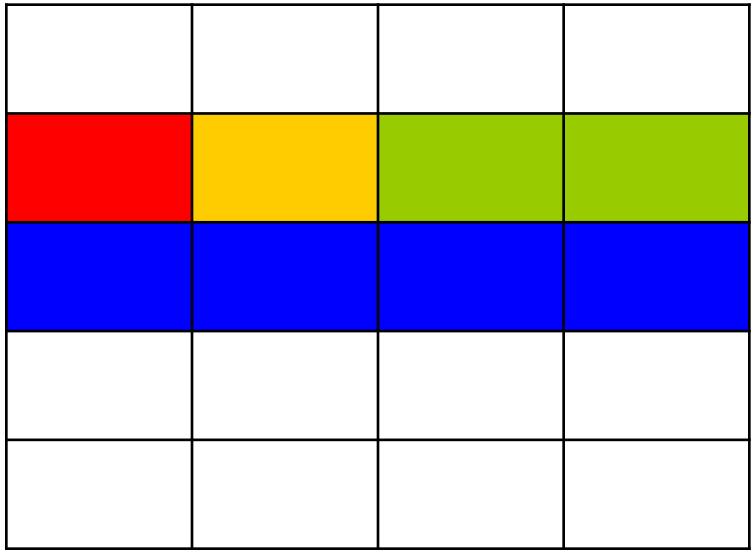
```
struct record  {
  char        a;
  char        c;
  short       d;
  int         b;
 } ;
struct  record R[64];
```

上述定义中，数组R占用多少字节？代码是否可以优化？

```
11:     R[0].a=1;
00401028 C6 45 F0 01          mov     byte ptr [ebp-10h],1
12:     R[0].b=2;
0040102C C7 45 F4 02 00 00 00 mov     dword ptr [ebp-0Ch],2
13:     R[0].c=3;
00401033 C6 45 F1 03          mov     byte ptr [ebp-0Fh],3
14:     R[0].d=4;
00401037 66 C7 45 F2 04 00    mov     word ptr [ebp-0Eh],offset main+2Bh (0040103b)
15:     R[1].a=5;
0040103D C6 45 F8 05          mov     byte ptr [ebp-8],5
16:     R[1].b=6;
00401041 C7 45 FC 06 00 00 00 mov     dword ptr [ebp-4],6
17:     R[1].c=7;
00401048 C6 45 F9 07          mov     byte ptr [ebp-7],7
18:     R[1].d=8;
0040104C 66 C7 45 FA 08 00    mov     word ptr [ebp-6],offset main+40h (00401050)
```

地址: 0x0012FF48

```
0012FF28   CC CC CC CC CC CC CC CC
0012FF30   CC CC CC CC CC CC CC CC
0012FF38   01 03 04 00 02 00 00 00
0012FF40   05 07 08 00 06 00 00 00
0012FF48   88 FF 12 00 39 12 40 00
0012FF50   01 00 00 00 A0 0E 2E 00
```



程序的机器级表示

- 分以下五个部分介绍
  - 第一讲：程序转换概述
    - 机器指令和汇编指令
    - 机器级程序员感觉到的属性和功能特性
    - 高级语言程序转换为机器代码的过程
  - 第二讲：IA-32 /x86-64指令系统
  - 第三讲： C语言程序的机器级表示
    - 过程调用的机器级表示
    - 选择语句的机器级表示
    - 循环结构的机器级表示
  - 第四讲：复杂数据类型的分配和访问
    - 数组的分配和访问
    - 结构体数据的分配和访问
    - 联合体数据的分配和访问
    - 数据的对齐
  - 第五讲：越界访问和缓冲区溢出 、x86-64架构

从高级语言程序出发，用其对应的机器级代码以及内存（栈）中信息的变化来说明底层实现

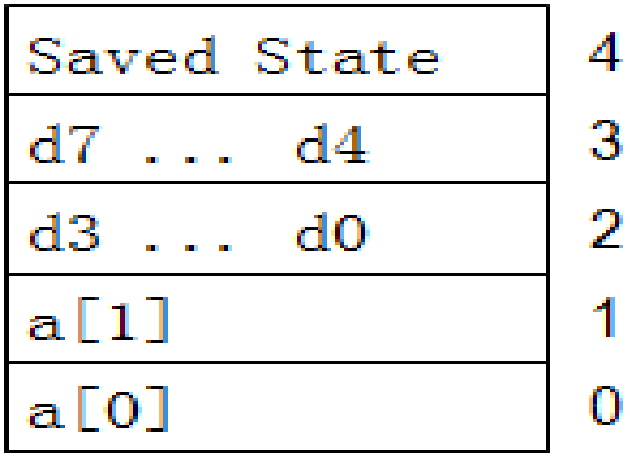
围绕C语言中的语句和复杂数据类型，解释其在底层机器级的实现方法

越界访问和缓冲区溢出

大家还记得以下的例子吗?

```
double fun(int i)
{
  volatile double d[1] = {3.14};
  volatile long int a[2];
  a[i] = 1073741824; /* Possibly out of bounds */
  return d[0];
}
```

fun(0) → 3.14
fun(1) → 3.14
fun(2) → 3.1399998664856
fun(3) → 2.0000061035156
fun(4) → 3.14, 然后存储保护错



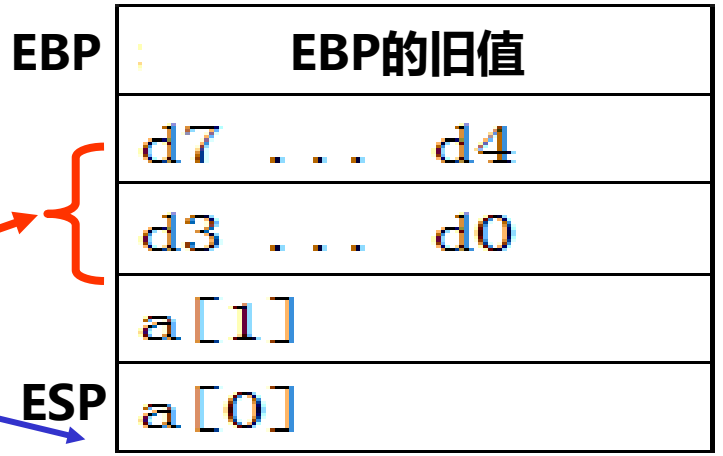
为什么当 i>1 就有问题?

因为数组访问越界!

```
double fun(int i)
{
  volatile double d[1] = {3.14};
  volatile long int a[2];
  a[i] = 1073741824;
  return d[0];
}
```

当i=0或1，OK
当i=2, d3~d0=0x40000000
低位部分（尾数）被改变
当i=3, d7~d3=0x40000000
高位部分被改变
当i=4, EBP被改变

```
<fun>:
    push   %ebp
    mov    %esp,%ebp
    sub    $0x10,%esp
    fldl   0x8048518
    fstpl  -0x8(%ebp)
    mov    0x8(%ebp),%eax
    movl   $0x40000000,-0x10(%ebp,%eax,4)
    fldl   -0x8(%ebp)
    leave
    ret
```



a[i]=1073741824;

0x40000000
$=2^{30}=1073741824$

return d[0];

fun(2) = 3.1399998664856
fun(3) = 2.00000061035156
fun(4) = 3.14, 然后存储保护错

越界访问和缓冲区溢出

- 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
- C标准规定，数组越界访问属于未定义行为，访问结果是不可预知的
- 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

main()函数的原型

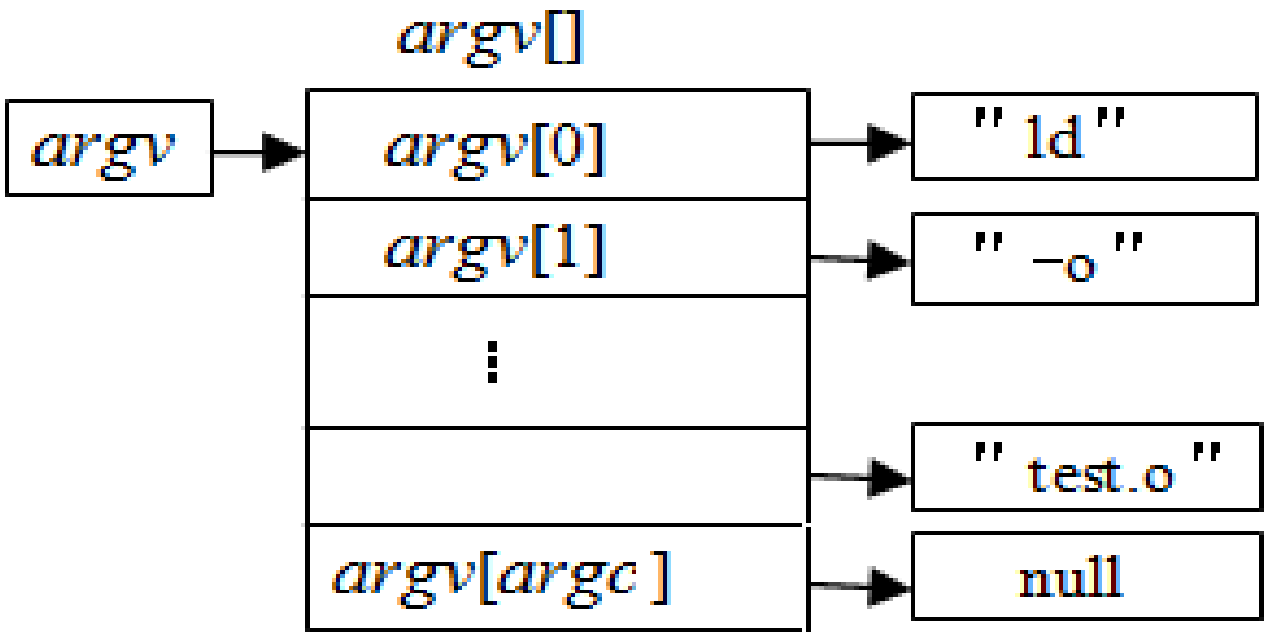
- **主函数main()的原型形式如下：**

  int main(int argc, char **argv, char **envp);　或

  int main(int argc, char *argv[], char *envp[]);

  argc：参数列表长度，参数列表中开始是命令名（可执行文件名），最后以NULL结尾。例: 当“.\hello” 时，argc=1

  例：当“ld -o test main.o test.o” 时，argc=5



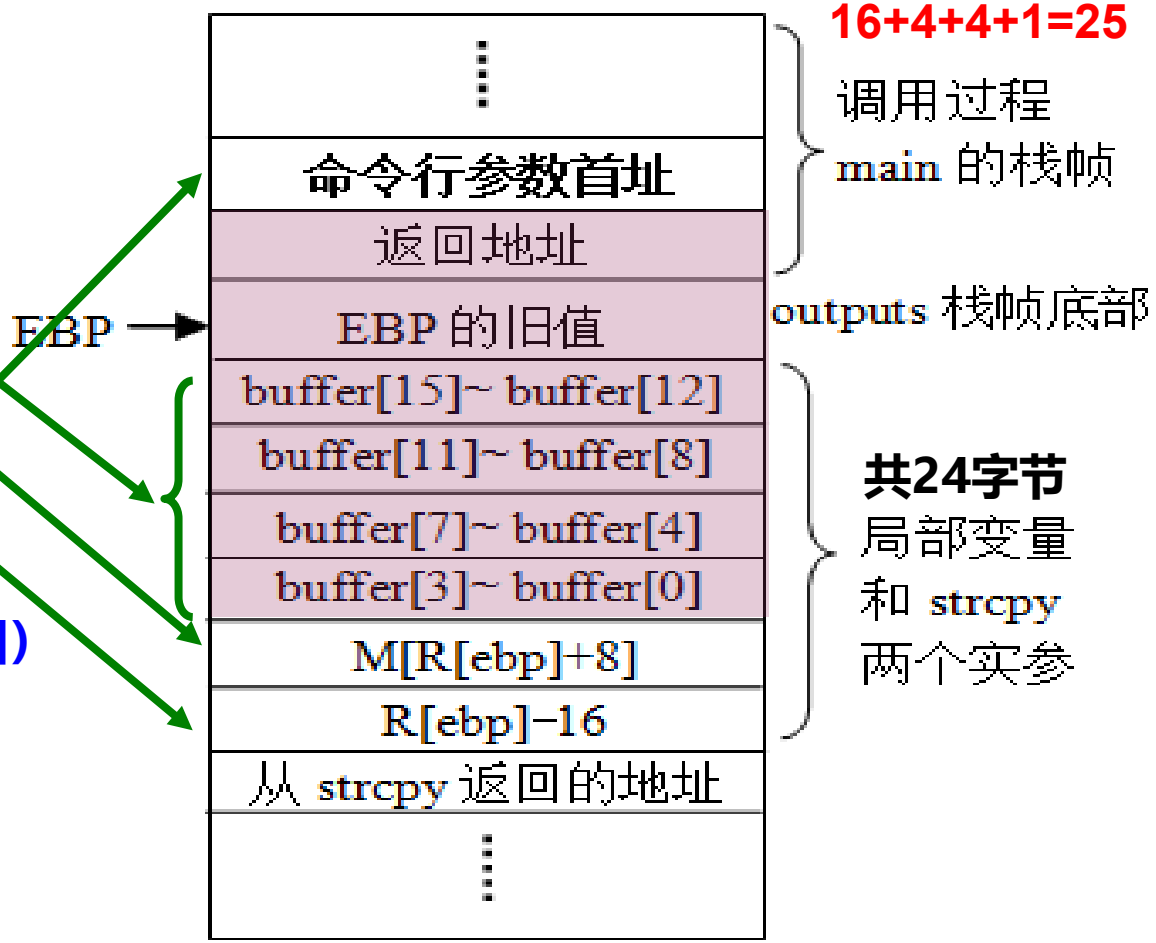
越界访问和缓冲区溢出

- 造成缓冲区溢出的原因是没有对栈中作为缓冲区的数组的访问进行越界检查。 举例：利用缓冲区溢出转到自设的程序hacker去执行

outputs漏洞：当命令行中字符串超25个字符时，使用strcpy函数就会使缓冲buffer造成写溢出并破坏返址

```
#include "stdio.h"
#include "string.h"
void outputs(char *str)
{
    char buffer[16];
    strcpy(buffer,str);
    printf("%s \n", buffer);
}
void hacker(void)
{
    printf("being hacked\n");
}
int main(int argc, char *argv[])
{
    outputs(argv[1]);
    return 0;
}
```

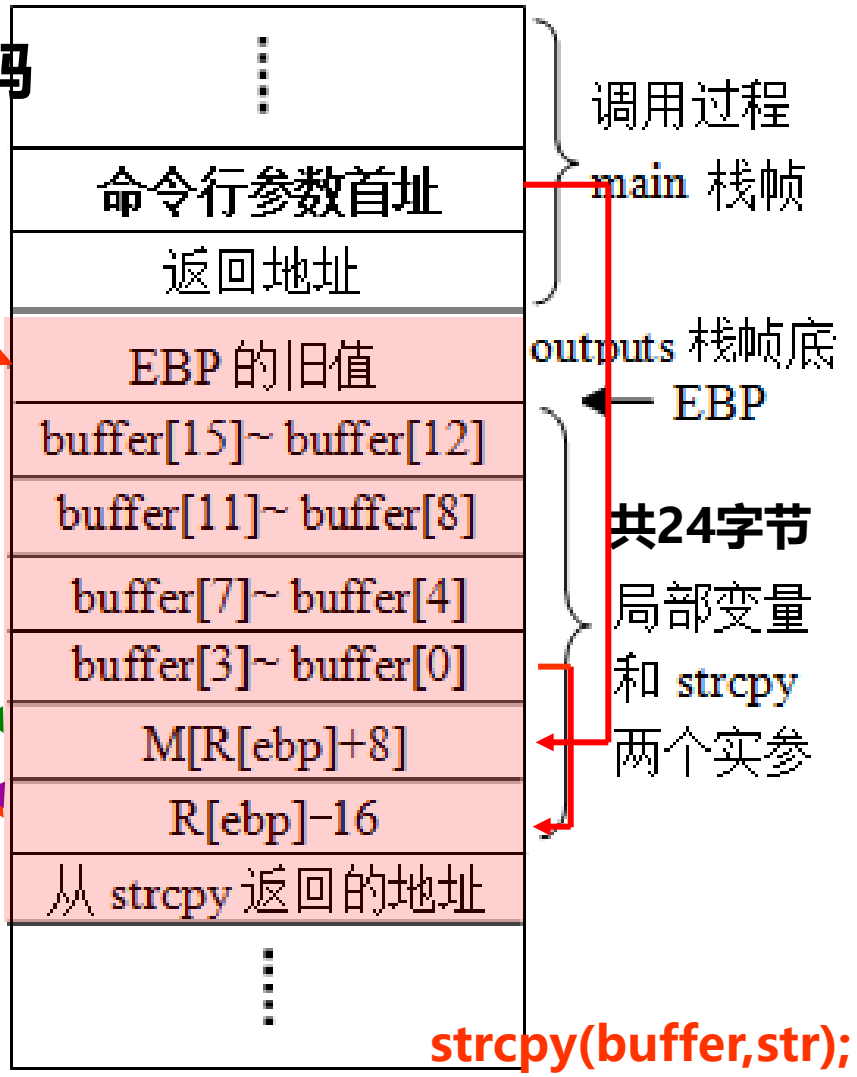
假定可执行文件名为test



越界访问和缓冲区溢出

test被反汇编得到的outputs汇编代码

```
080483e4 push   %ebp
080483e5 mov    %esp,%ebp
080483e7 sub    $0x18,%esp
080483ea mov    0x8(%ebp),%eax
080483ed mov    %eax,0x4(%esp)
080483f1 lea    0xfffffff0(%ebp),%eax
080483f4 mov    %eax,(%esp)
080483f7 call   0x8048330 <strcpy>
080483fc lea    0xfffffff0(%ebp),%eax
080483ff mov    %eax,0x4(%esp)
08048403 movl   $0x8048500,(%esp)
0804840a call   0x8048310
0804840f  leave
08048410 ret
```



若strcpy复制了25个字符到buffer中，并将hacker首址置于结束符‘\0’前4个字节，则在执行strcpy后，hacker代码首址被置于main栈帧返回地址处，当执行outputs代码的ret指令时，便会转到hacker函数实施攻击。

程序的加载和运行

- UNIX/Linux系统中，可通过调用execve()函数来加载并执行程序。
- execve()函数的用法如下：

  int execve(char *filename, char *argv[], *envp[]);

  filename是加载并运行的可执行文件名(如./hello)，可带参数列表argv和环境变量列表envp。若错误（如找不到指定文件filename），则返回-1，并将控制权交给调用程序； 若函数执行成功，则不返回，最终将控制权传递到可执行目标中的主函数main。

- 主函数main()的原型形式如下：

  int main(int argc, char **argv, char **envp); 或者：

  int main(int argc, char *argv[], char *envp[]);

前述例子： “.\test 0123456789ABCDEFXXXX▨ ▧▨▧” ,argc=2

argv[0] argv[1]

缓冲区溢出攻击

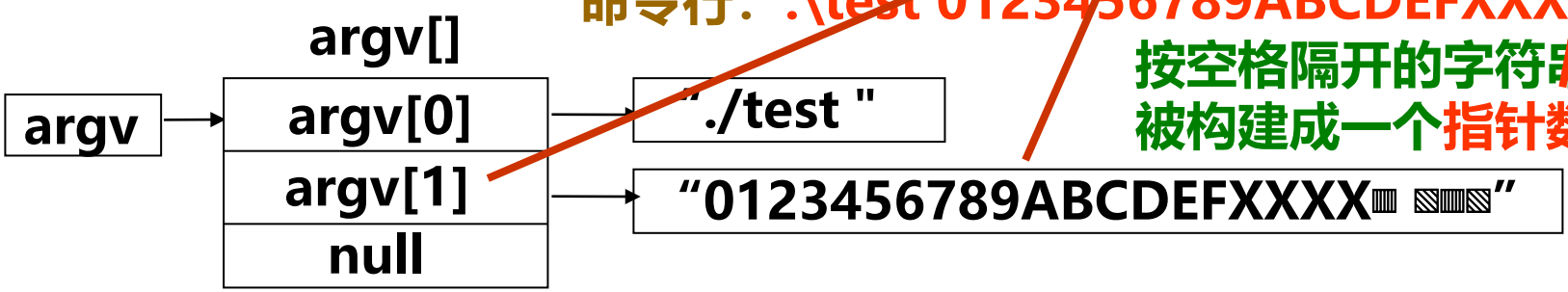
```
#include "stdio.h"
char code[]=
    "0123456789ABCDEFXXXX"
    "\x11\x84\x04\x08"
    "\x00";
int main(void)
{
    char *argv[3];
    argv[0]="./test";
    argv[1]=code;
    argv[2]=NULL;
    execve(argv[0],argv,NULL);
    return 0;
}
```

```
#include "stdio.h"
#include "string.h"
void outputs(char *str)
{
    char buffer[16];
    strcpy(buffer,str);
    printf("%s \n", buffer);
}
void hacker(void)
{
    printf("being hacked\n");
}
int main(int argc, char *argv[])
{
    outputs(argv[1]);
    return 0;
}
```

可执行文件名为test

命令行：.\test 0123456789ABCDEFXXXX▥ ▧▥▧

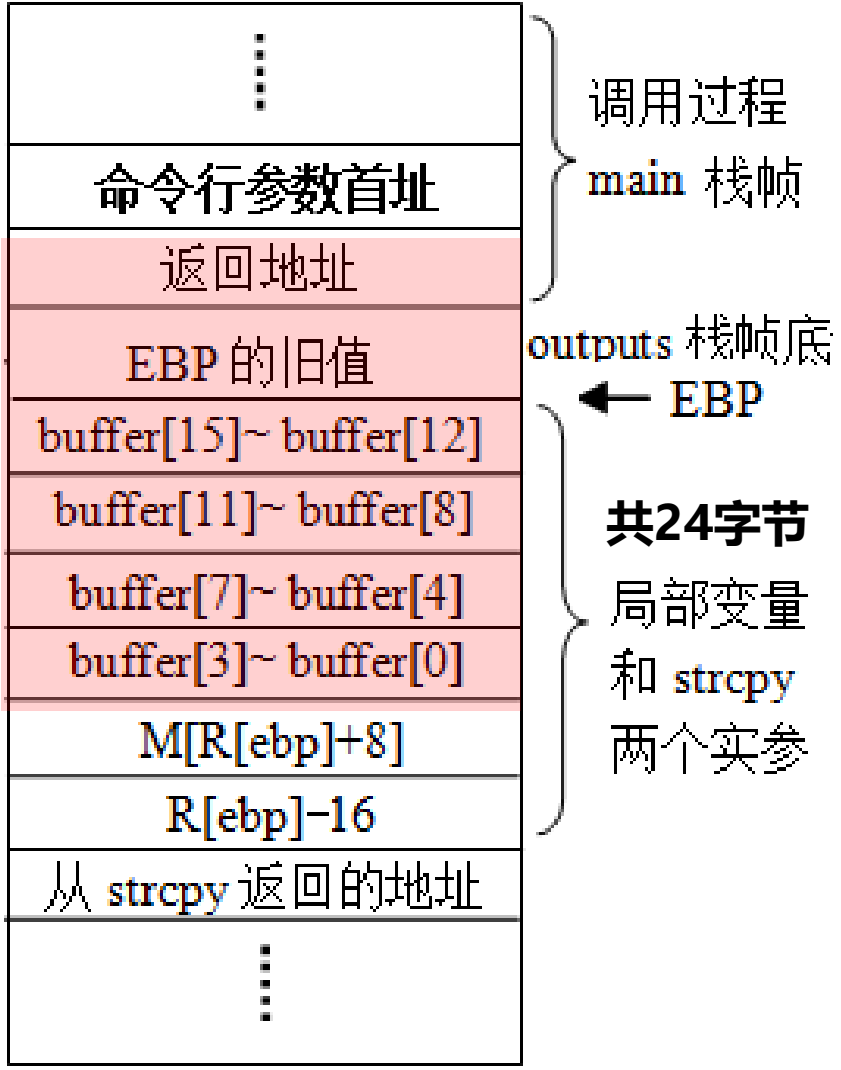
按空格隔开的字符串
被构建成一个指针数组



越界访问和缓冲区溢

假定hacker首址为0x08048411

```
void hacker(void) {
    printf("being hacked\n");
}
#include "stdio.h"
char code[]=
      "0123456789ABCDEFXXXX"
      "\x11\x84\x04\x08"
      "\x00";
int main(void) {
      char *argv[3];
      argv[0]="./test";
      argv[1]=code;
      argv[2]=NULL;
      execve(argv[0],argv,NULL);
      return 0;
}
```



执行上述攻击程序后的输出结果为：

"0123456789ABCDEFXXXX

being hacked

Segmentation fault

最后显示“Segmentation fault”，原因是执行到hacker过程的ret指令时取到的“返回地址”是一个不确定的值，因而可能跳转到数据区或系统区或其他非法访问的存储区执行，因而造成段错误。

缓冲区溢出攻击的防范（自学）

- 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

    （1）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

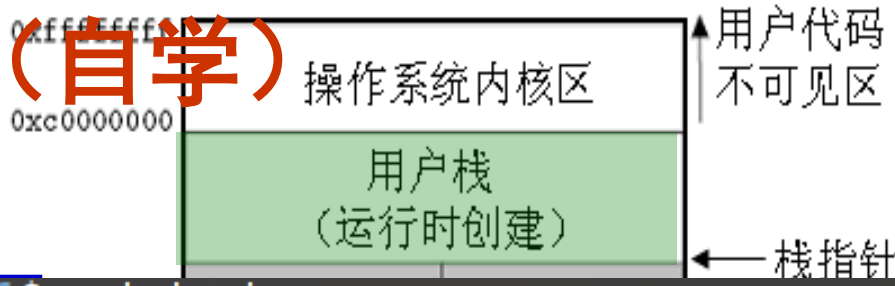
    目前在Linux、FreeBSD和Windows Vista等OS使用

    （2）栈破坏检测

    （3）可执行代码区域限制

    等等

缓冲溢出攻击防范（自学）



（1）地址空间随机化

- 只要操作系统相同，则
就一样，若攻击者知道
序使用的栈地址空间，
计一个针对性攻击，在
程序机器上实施攻击
- 地址空间随机化（栈随
的基本思路是，将加载
生成的代码段、静态数
堆区、动态库和栈区各
首地址进行随机化处理
次启动时，程序各段被
不同地址起始处
- 对于随机生成的栈起始
攻击者不太容易确定栈
位置

引用：唐瑞泽的PPT

```
环境1$ cat test.c
#include <stdio.h>
int main()
{
    int a=10;
    double *p=(double*)&a;
    printf("Scientific: %e\n", *p);
    printf("Machine: %08x %08x\n", *(&a+1), a);
    printf("Address: %p\n\n", &a);
    return 0;
}
环境1$ gcc test.c -o test
环境1$ for i in `seq 3`; do ./test; done
Scientific: -4.083169e-02
Machine: bfa4e7e4 0000000a
Address: 0xbfa4e7e4

Scientific: -1.102164e-02
Machine: bf869284 0000000a
Address: 0xbf869284

Scientific: -3.986657e-02
Machine: bfa46964 0000000a
Address: 0xbfa46964
```

缓冲区溢出攻击的防范（自学）

（2）栈破坏检测

- 若在程序跳转到攻击代码前能检测出程序栈已被破坏，就可避免受到严重攻击


```
int main()
{
00CF17A0  push        ebp
00CF17A1  mov         ebp,esp
00CF17A3  sub         esp,0DCh
00CF17A9  push        ebx
00CF17AA  push        esi
00CF17AB  push        edi
00CF17AC  lea         edi,[ebp-0DCh]
00CF17B2  mov         ecx,37h
00CF17B7  mov         eax,0CCCCCCCCh
00CF17BC  rep stos    dword ptr es:[edi]
00CF17BE  mov         eax,dword ptr [__security_cookie (0CF9004h)]
00CF17C3  xor         eax,ebp
00CF17C5  mov         dword ptr [ebp-4],eax
    int a = 10;
00CF17C8  mov         dword ptr [a],0Ah
    double *p = (double*)&a;
```

监视 2

名称	值
&a	0x009efd68 {10}

将栈空间全部初始化为0xCC

防止缓冲器溢出攻击的“金丝雀“

009EFD74H	//ebp
009EFD70H	security cookie ⊕R[ebp]
009EFD6CH	CCCCCCCCH
009EFD68H	a=10
.....	.....
009EFC98H	旧ebx
009EFC94H	旧esi
009EFC90H	旧edi

ESP->

引用：谢旻晖的PPT

缓冲区溢出攻击的防范（自学）

（3）可执行代码区域限制

- 通过将程序栈区和堆区设置为不可执行，从而使得攻击者不可能执行被植入在输入缓冲区的代码，这种技术也被称为非执行的缓冲区技术。
- 早期Unix系统只有代码段的访问属性是可执行，其他区域的访问属性是可读或可读可写。但是，近来Unix和Windows系统由于要实现更好的性能和功能，允许在栈段中动态地加入可执行代码，这是缓冲区溢出的根源。
- 为保持程序兼容性，虽然有些非代码段可设置成可执行区域。但是通过将动态的栈段设置为不可执行，既可保证程序的兼容性，又可以有效防止把代码植入栈（自动变量缓冲区）的溢出攻击。

X86-64架构（自学）

- 背景
  - Intel最早推出的64位架构是基于超长指令字VLIW技术的IA-64体系结构，Intel 称其为显式并行指令计算机EPIC （Explicitly Parallel Instruction Computer）。安腾和安腾2分别在2000年和2002年问世，它们是IA-64体系结构的最早的具体实现。
  - AMD公司利用Intel在IA-64架构上的失败，抢先在2003年推出兼容IA-32的64位版本指令集x86-64，AMD获得了以前属于Intel的一些高端市场。AMD后来将x86-64更名为AMD64。
  - Intel在2004年推出IA32-EM64T，它支持x86-64指令集。Intel为了表示EM64T的64位模式特点，又使其与IA-64有所区别，2006年开始把EM64T改名为Intel 64。

X86-64架构（自学）

- 与IA-32相比，x86-64架构的主要特点
  - 新增8个64位通用寄存器：R8、R9、R10、R11、R12、R13、R14和R15。可作为8位（R8B~R15B）、16位（R8W~R15W）或32位寄存器（R8D~R15D）使用
  - 所有GPRs都从32位扩充到64位。8个32位通用寄存器EAX、EBX、ECX、EDX、EBP、ESP、ESI和 EDI对应扩展寄存器分别为RAX、RBX、RCX、RDX、RBP、RSP、RSI和RDI；EBP、ESP、ESI和 EDI的低8位寄存器分别是BPL、SPL、SIL和DIL
  - 字长从32位变为64位，故逻辑地址从32位变为64位
  - long double型数据虽还采用80位扩展精度格式，但所分配存储空间从12B扩展为16B，即改为16B对齐，但不管是分配12B还是16B，都只用到低10B
  - 过程调用时，通常用通用寄存器而不是栈来传递参数，因此，很多过程不用访问栈，这使得大多数情况下执行时间比IA-32代码更短
  - 128位的MMX寄存器从原来的8个增加到16个，浮点操作采用基于SSE的面向XMM寄存器的指令集，而不采用基于浮点寄存器栈的指令集

X86-64架构（自学）

- **过程调用的参数传递（Linux/GCC）**
  - **通过寄存器传送参数**
  - **最多可有6个整型或指针型参数通过寄存器传递**
  - **超过6个入口参数时，后面的通过栈来传递**
  - **在栈中传递的参数若是基本类型，则都被分配8个字节**
  - **call（或callq）将64位返址保存在栈中之前，执行R[rsp]←R[rsp]-8**
  - **ret从栈中取出64位返回地址后，执行R[rsp]←R[rsp]+8**

操作数宽度（字节）	入口参数						返回参数
	1	2	3	4	5	6	
8	RDI	RSI	RDX	RCX	R8	R9	RAX
4	EDI	ESI	EDX	ECX	R8D	R9D	EAX
2	DI	SI	DX	CX	R8W	R9W	AX
1	DIL	SIL	DL	CL	R8B	R9B	AL

X86-64架构（自学）

- x86-64的基本指令和对齐
  - 数据传送指令（汇编指令中助记符“q”表示操作数长度为四字（即64位））
    - movabsq指令用于将一个64位立即数送到一个64位通用寄存器中；
    - movq指令用于传送一个64位的四字；
    - movsbq、movswq、movslq用于将源操作数进行符号扩展并传送到一个64位寄存器或存储单元中；
    - movzbq、movzwq用于将源操作数进行零扩展后传送到一个64位寄存器或存储单元中；
    - pushq和popq分别是四字压栈和四字出栈指令；
    - movl指令的功能相当于movzlq指令 。

X86-64架构（自学）

- **数据传送指令举例**

以下函数功能是将类型为source_type的参数转换为dest_type型数据并返回

```
dest_type  convert(source_type x) {
        dest_type y = (dest_type) x;
        return y;
}
```

根据参数传递约定知，x在RDI对应的适合宽度的寄存器（RDI、EDI、DI和DIL）中，y存放在RAX对应的寄存器（RAX、EAX、AX或AL）中，填写下表中的汇编指令以实现convert函数中的赋值语句

source_type	dest_type	汇 编 指 令
char	long	问题：每种情况对应的汇编指令各是什么?
int	long	
long	long	
long	int	
unsigned int	unsigned long	
unsigned long	unsigned int	
unsigned char	unsigned long	

X86-64架构（自学）

- **数据传送指令举例**

以下函数功能是将类型为source_type的参数转换为dest_type型数据并返回

```
dest_type  convert(source_type x) {
        dest_type y = (dest_type) x;
        return y;
}
```

根据参数传递约定知，x在RDI对应的适合宽度的寄存器（RDI、EDI、DI和DIL）中，y存放在RAX对应的寄存器（RAX、EAX、AX或AL）中，填写下表中的汇编指令以实现convert函数中的赋值语句

source_type	dest_type	汇 编 指 令
char	long	movsbq %dil, %rax
int	long	movslq %edi, %rax
long	long	movq %rdi, %rax
long	int	movslq %edi, %rax //符号扩展到 64 位 movl %edi, %eax // **只需x的低32位**
unsigned int	unsigned long	movl %edi, %eax //零扩展到 64 位
unsigned long	unsigned int	movl %edi, %eax //零扩展到 64 位
unsigned char	unsigned long	movzbq %dil, %rax //零扩展到 64 位

X86-64架构（自学）

- 算术逻辑运算指令
  - addq（四字相加）
  - subq（四字相减）
  - imulq（带符号整数四字相乘）
  - orq（64位相或）
  - leaq（有效地址加载到64位寄存器）

以下是C赋值语句“x=a*b+c*d;”对应的x86-64汇编代码，已知x、a、b、c和d分别在寄存器RAX(x)、RDI(a)、RSI(b)、RDX(c)和RCX(d)对应宽度的寄存器中。根据以下汇编代码，推测x、a、b、c和d的数据类型

```
movslq %ecx, %rcx
imulq  %rdx, %rcx
movsbl %sil, %esi
imull %edi, %esi
movslq %esi, %rsi
leaq (%rcx, %rsi), %rax
```

d从32位符号扩展为64位，故d为int型

在RDX中的c为64位long型

在SIL中的b为char型

在EDI中的a是int型

在RAX中的x是long型

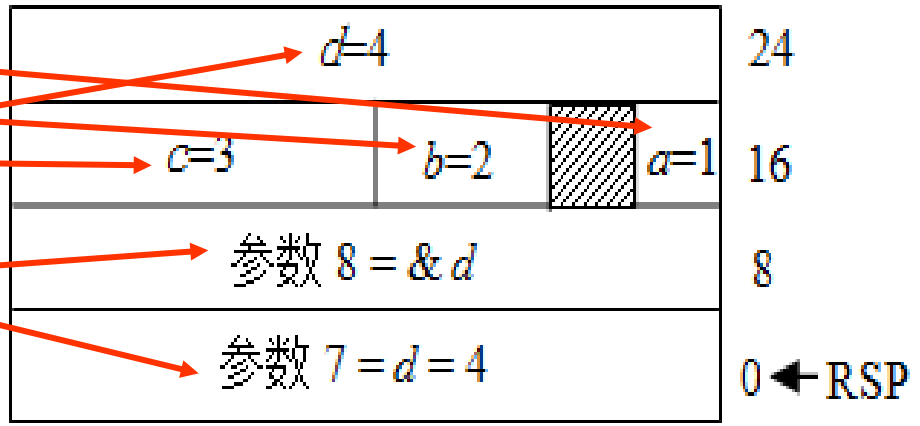
X86-64架构过程调用举例（自学）

```
long caller ( )
{
    char a=1;
    short b=2;
    int c=3;
    long d=4;
    test(a, &a, b, &b, c, &c, d, &d);
    return  a*b+c*d;
}
```

其他6个参数在哪里?

```
void test(char a, char *ap,
          short b, short *bp,
          int c, int *cp,
          long d, long *dp)
{
    *ap+=a;
    *bp+=b;
    *cp+=c;
    *dp+=d;
}
```

执行到 caller 的 call 指令时栈中情况



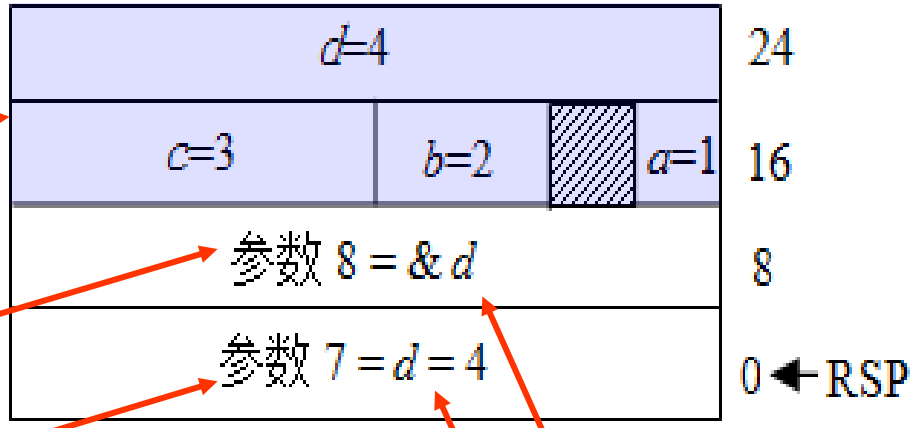
执行到caller的call指令前，栈中的状态如何?

操作数宽度（字节）	入口参数 1	2	3	4	5	6	返回参数
8	RDI	RSI	RDX	RCX	R8	R9	RAX
4	EDI	ESI	EDX	ECX	R8D	R9D	EAX
2	DI	SI	DX	CX	R8W	R9W	AX
1	DIL	SIL	DL	CL	R8B	R9B	AL

X86-64架构过程调用举例（自学）

```
subq $32, %rsp          //R[rsp]←R[rsp]-32
movb $1, 16(%rsp)       //M[R[rsp]+16]←1
movw $2, 18(%rsp)       //M[R[rsp]+18]←2
movl $3, 20(%rsp)       //M[R[rsp]+20]←3
movq $4, 24(%rsp)       //M[R[rsp]+24]←4
leaq 24(%rsp), %rax     //R[rax]←R[rsp]+24
movq %rax, 8(%rsp)      //M[R[rsp]+8]←R[rax]
movq $4, (%rsp)         //M[R[rsp]]←4
leaq 20(%rsp), %r9      //R[r9]←R[rsp]+20
movl $3, %r8d           //R[r8d]←3
leaq 18(%rsp), %rcx     //R[rcx]←R[rsp]+18
movw $2, %dx            //R[dx]←2
leaq 16(%rsp), %rsi     //R[rsi]←R[rsp]+16
movb $1, %dil           //R[dil]←1
call test   第15条指令
```

执行到 caller 的 call 指令时栈中情况




```
long caller ( )
{
    char a=1;
    short b=2;
    int c=3;
    long d=4;
    test(a, &a, b, &b, c, &c, d, &d);
    return  a*b+c*d;
}
```

X86-64架构过程调用举例（自学）

```
movq 16(%rsp), %r10 //R[r10]←M[R[rsp]+16]
addb %dil, (%rsi)   //M[R[rsi]]←M[R[rsi]]+R[dil]
addw %dx, (%rcx)    //M[R[rcx]]←M[R[rcx]]+R[dx]
addl %r8d, (%r9)    //M[R[r9]]←M[R[r9]]+R[r8d]
movq 8(%rsp), %rax  //R[rax]←M[R[rsp]+8]
addq %rax, (%r10)   //M[R[r10]]←M[R[r10]]+R[rax]
ret
```

R[r10]←&d

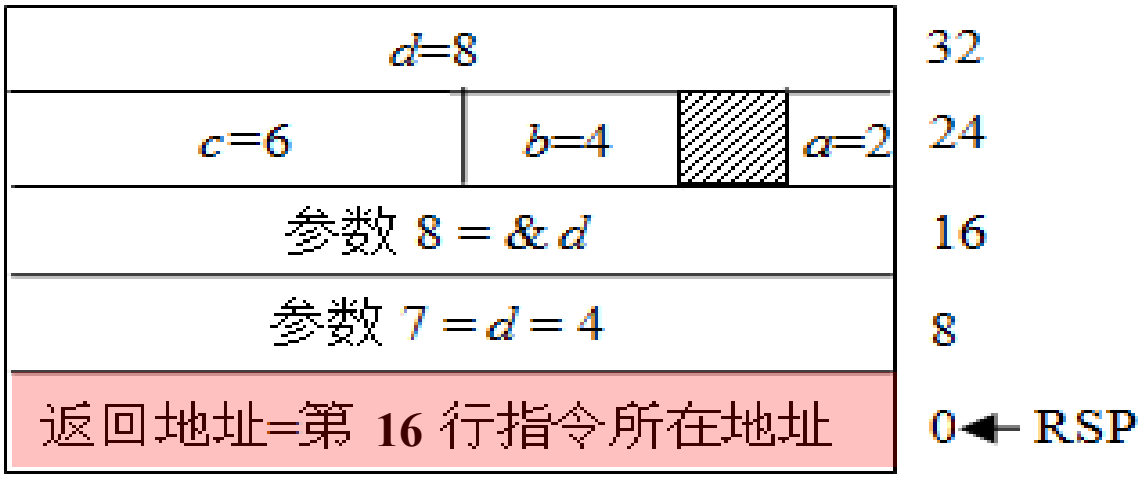
*ap+=a;

*bp+=b;

*cp+=c;

*dp+=d;

执行到test的ret指令前，栈中的状态如何？ret执行后怎样？



DIL、RSI、DX、RCX、R8D、R9

```
void test(char a, char *ap,
          short b, short *bp,
          int c, int *cp,
          long d, long *dp)
{
    *ap+=a;
    *bp+=b;
    *cp+=c;
    *dp+=d;
}
```

X86-64架构过程调用举例（自学）

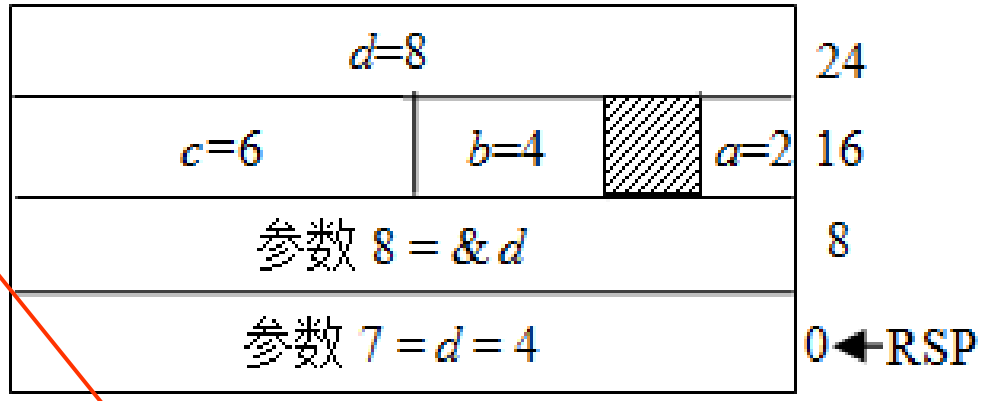
从第16条指令开始

```
movslq 20(%rsp), %rcx
movq   24(%rsp), %rdx
imulq  %rdx, %rcx
movsbw 16(%rsp), %ax
movw   18(%rsp), %dx
imulw  %dx, %ax
movswq %ax, %rax
leaq   (%rax, %rcx),%rax
addq   $32, %rsp
ret
```

释放caller的栈帧

执行到ret指令时，
RSP指向调用caller
函数时保存的返回值

执行test的ret指令后，栈中的状态如何?




```
long caller ( )
{
    char a=1;
    short b=2;
    int c=3;
    long d=4;
    test(a, &a, b, &b, c, &c, d, &d);
    return  a*b+c*d;
}
```

浮点操作与SIMD指令（自学）

- IA-32的浮点处理架构有两种

  (1) x86配套的浮点协处理器x87FPU架构，80位浮点寄存器栈

  (2) 由MMX发展而来的SSE指令集架构，采用的是单指令多数据（Single Instruction Multi Data，SIMD）技术

  对于IA-32架构， gcc默认生成x87 FPU 指令集代码

  如果想要生成SEE指令集代码，则需要设置适当的编译选项

- 在x86-64中，浮点运算采用SIMD指令

  浮点数存放在128位的XMM寄存器中

IA-32和x86-64的比较（自学）

例：以下是一段C语言代码：

```
#include <stdio.h>
int main()
{
    double a = 10;
    printf("a = %d\n", a);
}
```

在IA-32上运行时，打印结果为a=0

在x86-64上运行时，打印一个不确定值

为什么?

$10=1010B=1.01\times2^3$

阶码e=1023+3=10000000010B

10的double型表示为：

0 10000000010 0100...0B

即4024 0000 0000 0000H

先执行fldl，再执行fstpl

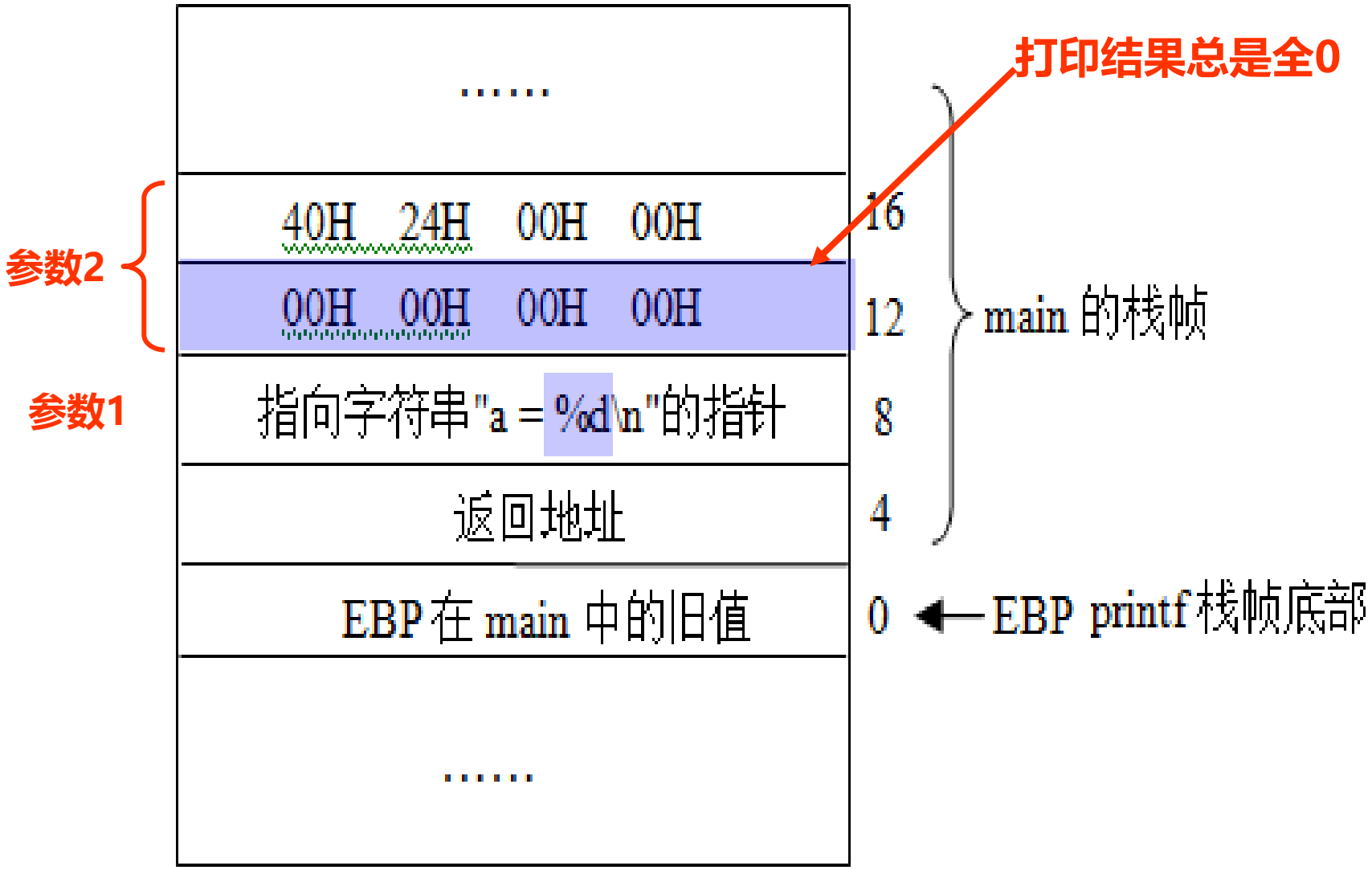
fldl：局部变量区→ST(0)

fstpl：ST(0) →参数区

在IA-32中a为float型又怎样呢？先执行flds，再执行fstpl

即：flds将32位单精度转换为80位格式入浮点寄存器栈，fstpl再将80位转换为64位送存储器栈中，故实际上与a是double效果一样！

IA-32过程调用参数传递（自学）



a的机器数对应十六进制为：40 24 00 00 00 00 00 00H

X64参数传递（Linux/GCC）

```
int main()
{
    double a = 10;
    printf("a = %d\n", a);
}
```

操作数宽度（字节）	入口参数						返回参数
	1	2	3	4	5	6	
8	RDI	RSI	RDX	RCX	R8	R9	RAX
4	EDI	ESI	EDX	ECX	R8D	R9D	EAX
2	DI	SI	DX	CX	R8W	R9W	AX
1	DIL	SIL	DL	CL	R8B	R9B	AL

```
.LC1:
        .string "a = %d\n "
……
movsd   .LC0(%rip), %xmm0 //a送xmm0
movl    $.LC1, %edi //RDI 高32位为0
movl    $1, %eax
call     printf
addq    $8, %rsp
ret
……
.LC0:
    .long   0
    .long   1076101120
```

00000000H

40240000H

小端方式！0存在低地址上

printf中为%d，故将从ESI中取打印参数进行处理；但a是double型数据，在x86-64中，a的值被送到XMM寄存器中而不会送到ESI中。故在printf执行时，从ESI中读取的并不是a的低32位，而是一个不确定的值。

X64参数传递（Win64/VC）

```
int main()
{
    double a = 10;
    printf("a = %d\n", a);
}
```

前4个整数、浮点参数分别通过

RCX、RDX、R8、R9传递

XMM0、XMM1、XMM2、XMM3

```
sub      $0x28, %rsp
callq    0x402269 <_main>
movsd    .LC0(%rip), %xmm0 //a送xmm0
movapd   %xmm0, %xmm1
movq     %xmm0, %rdx  //rdx低32位为0
Lea      0x2aef(%rip), %rcx //字符串首址
callq    0x402b18 <printf>
add      $0x28, %rsp
retq
......
.LC0:
    .long   0          00000000H
    .long   1076101120 40240000H
```

小端方式！0存在低地址上

printf中为%d，故将从RDX中低32位取打印参数进行处理；其中是double型数据10的低32位，因此为全0，故最后打印结果为0。

关于x86-64的调用约定的详细内容，可以参考AMD64 System V ABI手册

X86-64架构（自学）

- 数据的对齐
  - x86-64中各类型数据遵循一定的对齐规则，而且更严格
  - x86-64中存储器访问接口被设计成按8字节或16字节为单位进行存取，其对齐规则是，任何K字节宽的基本数据类型和指针类型数据的起始地址一定是K的倍数。
    - short型数据必须按2字节边界对齐
    - int、float等类型数据必须按4字节边界对齐
    - long型、double型、指针型变量必须按8字节边界对齐
    - long double型数据必须按16字节边界对齐

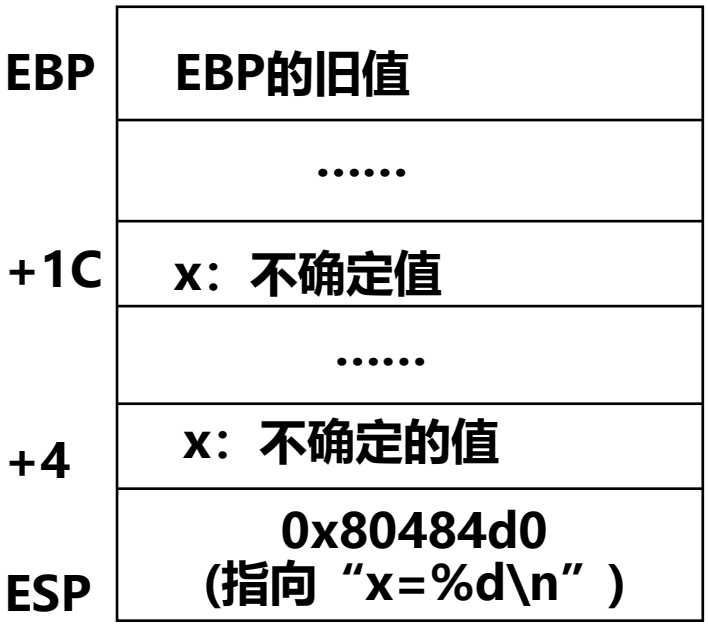
具体的对齐规则可以参考AMD64 System V ABI手册。

本章总结

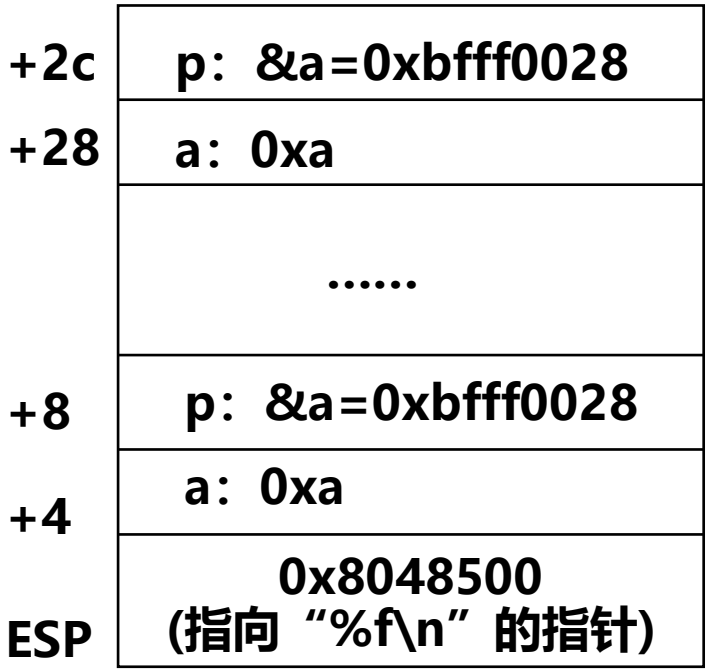
- 分以下五个部分介绍
  - 第一讲：程序转换概述
    - 机器指令和汇编指令
    - 机器级程序员感觉到的属性和功能特性
    - 高级语言程序转换为机器代码的过程
  - 第二讲：IA-32 /x86-64指令系统
  - 第三讲： C语言程序的机器级表示
    - 过程调用的机器级表示
    - 选择语句的机器级表示
    - 循环结构的机器级表示
  - 第四讲：复杂数据类型的分配和访问
    - 数组的分配和访问
    - 结构体数据的分配和访问
    - 联合体数据的分配和访问
    - 数据的对齐
  - 第五讲：越界访问和缓冲区溢出 、x86-64架构

从高级语言程序出发，用其对应的机器级代码以及内存（栈）中信息的变化来说明底层实现

围绕C语言中的语句和复杂数据类型，解释其在底层机器级的实现方法

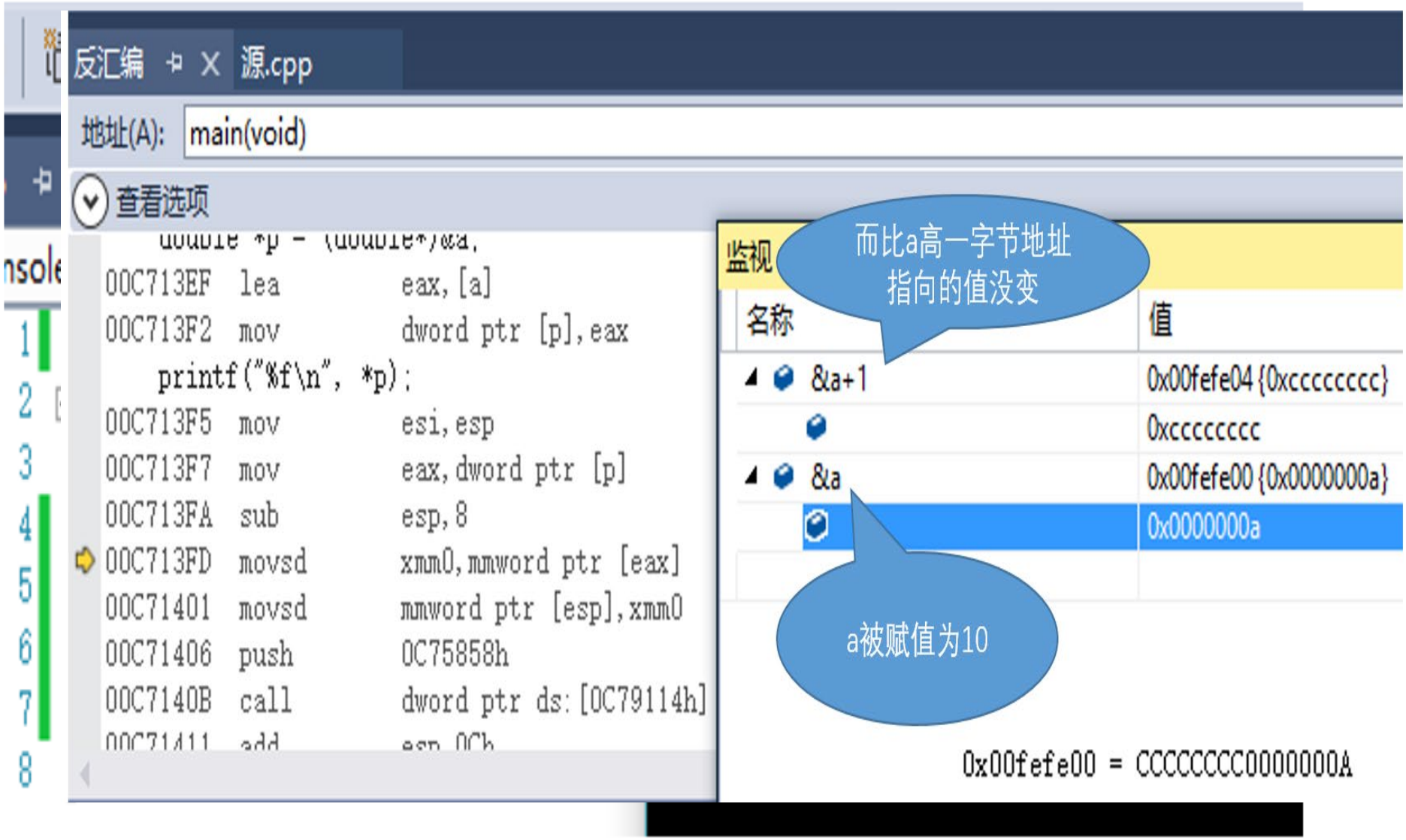


R[esp]：最低4位为0



假定R[esp]=0xbfff0000

Windows中的对齐和分配顺序



变量a的地址比变量p的地址更大，因而*p的高位为&a+1中的内容（Debug下为CCCCCCCCH，Release下为0）。

引用：唐瑞泽的PPT

有关“过程调用”的讨论

在32位Linux系统中反汇编结果：

```
int a = 10;
 8048425:  c7 44 24 28 0a 00 00 00     movl   $0xa,0x28(%esp)
```

+2c	p : &a=0xbfff0028
+28	a : 0xa
	.....
+8	p : &a=0xbfff0028
+4	a : 0xa
ESP	0x8048500 (指向 “%f\n” 的指针)

假定R[esp]=0xbfff0000

```
                                       lea    0x28(%esp),%eax
                                       mov    %eax,0x2c(%esp)
```
编层次并没有体现出来，都是直接 mov 过去
```
                                       mov    0x2c(%esp),%eax
                                       fldl   (%eax)
```
精度加载到浮点栈顶 ST(0)）
```
                                       fstpl  0x4(%esp)
```
*p 的类型是 double，故按 64 位压栈）
```
                                       movl   $0x8048500,(%esp)
                                       call   8048300 <printf@plt>
```

打印出来的是一个负数

```
                                       mov    0x28(%esp),%eax
                                       mov    %eax,0x1c(%esp)
```
由于没有优化，这里有一些冗余的 mov 操作，把变量 a 的值移来移去
```
 8048453:  db 44 24 1c                 fildl  0x1c(%esp)
```
把 10 转换成 double 型，注意这里用的是 fildl 指令，和上面用的 fldl 指令不一样!

对齐方式的设定

#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。

对齐方式的设定

```
#include<stdio.h>
#pragma pack(4)
typedef struct {
    uint32_t    f1;
    uint8_t     f2;
    uint8_t     f3;
    uint32_t    f4;
    uint64_t    f5;
}__attribute__((aligned(1024))) ts;
int main()
{
    printf("Struct size is: %d, aligned on 1024\n",sizeof(ts));
    printf("Allocate f1 on address: 0x%x\n",&(((ts*)0)->f1));
    printf("Allocate f2 on address: 0x%x\n",&(((ts*)0)->f2));
    printf("Allocate f3 on address: 0x%x\n",&(((ts*)0)->f3));
    printf("Allocate f4 on address: 0x%x\n",&(((ts*)0)->f4));
    printf("Allocate f5 on address: 0x%x\n",&(((ts*)0)->f5));
    return 0;
}
```

输出：

```
Struct size is: 1024, aligned on 1024
Allocate f1 on address: 0x0
Allocate f2 on address: 0x4
Allocate f3 on address: 0x5
Allocate f4 on address: 0x8
Allocate f5 on address: 0xc
```

```c
#include <stdio.h>
//#pragma pack(1)
struct test
{
    char x2;
    int x1;
    short x3;
    long long x4;
}__attribute__((packed));
struct test1
{
    char x2;
    int x1;
    short x3;
    long long x4;
};
struct test2
{
    char x2;
    int x1;
    short x3;
    long long  x4;
}__attribute__((aligned(8)));

void main()
{
    printf("size=%d\n",sizeof(struct test));
    printf("size=%d\n",sizeof(struct test1));
    printf("size=%d\n",sizeof(struct test2));
}
```

输出结果是什么?

```
size=15
size=20
size=24
```

```c
#include <stdio.h>
#pragma pack(1)
struct test
{
    char x2;
    int x1;
    short x3;
    long long x4;
}__attribute__((packed));
struct test1
{
    char x2;
    int x1;
    short x3;
    long long x4;
};
struct test2
{
    char x2;
    int x1;
    short x3;
    long long  x4;
}__attribute__((aligned(8)));

void main()
{
    printf("size=%d\n",sizeof(struct test));
    printf("size=%d\n",sizeof(struct test1));
    printf("size=%d\n",sizeof(struct test2));
}
```

如果设置了pragma pack(1)，结果又是什么?

```
size=15
size=15
size=16
```

```c
#include <stdio.h>
#pragma pack(2)
struct test
{
    char x2;
    int x1;
    short x3;
    long long x4;
}__attribute__((packed));
struct test1
{
    char x2;
    int x1;
    short x3;
    long long x4;
};
struct test2
{
    char x2;
    int x1;
    short x3;
    long long  x4;
}__attribute__((aligned(8)));

void main()
{
    printf("size=%d\n",sizeof(struct test));
    printf("size=%d\n",sizeof(struct test1));
    printf("size=%d\n",sizeof(struct test2));
}
```

如果设置了pragma pack(2)，结果又是什么?

```
size=15
size=16
size=16
```