结构和联合数据类型的分配和访问

南京大学
计算机科学与技术系
袁春风
email: cfyuan@nju.edu.cn
2015.6

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

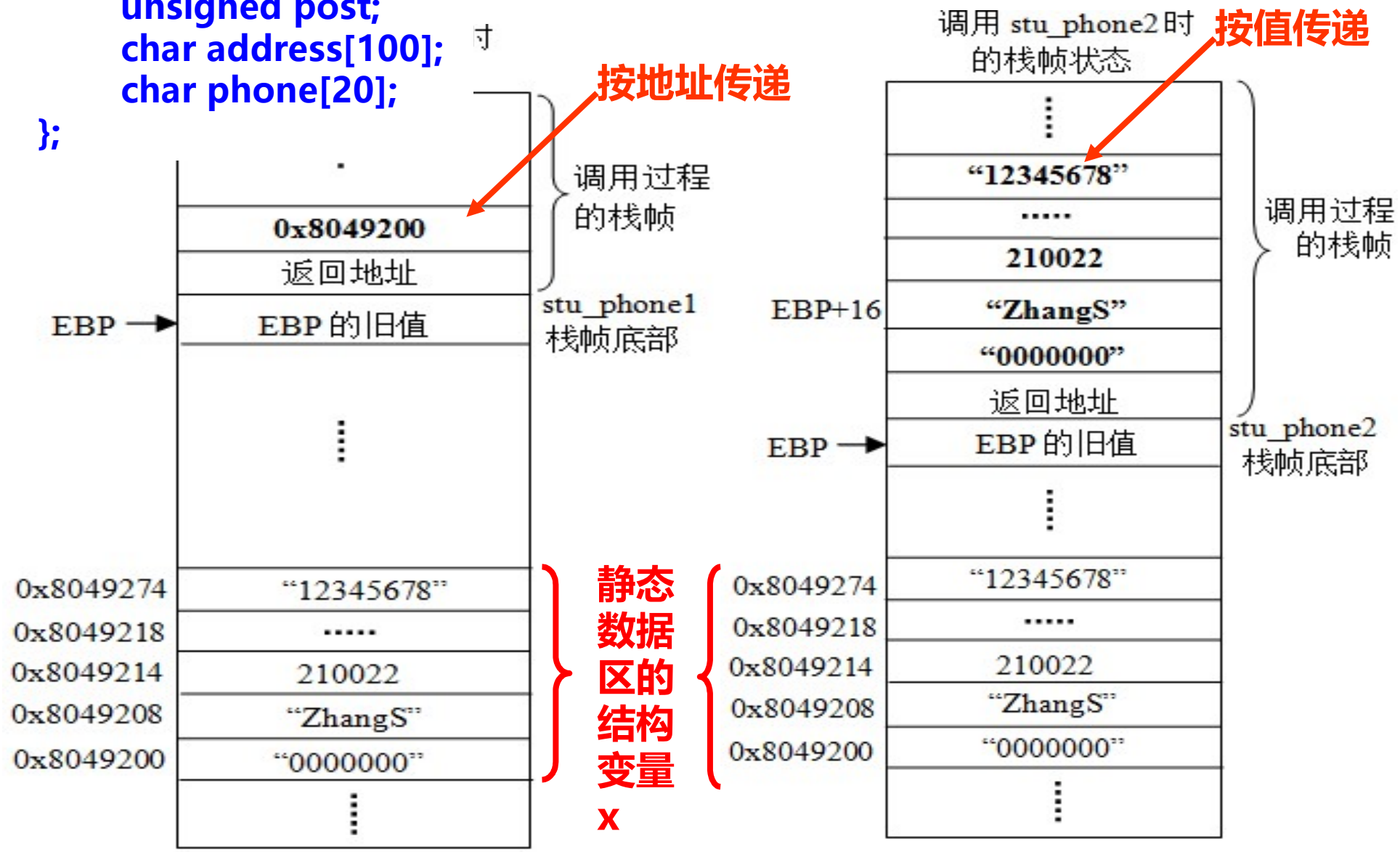
- **结构体数据作为入口参数**
  - **当结构体变量需要作为一个函数的形参时，形参和调用函数中的实参应具有相同结构**
  - **有按值传递和按地址传递两种方式**
    - **若采用按值传递，则结构成员都要复制到栈中参数区，这既增加时间开销又增加空间开销，且更新后的数据无法在调用过程使用**
    - **通常应按地址传递，即：在执行CALL指令前，仅需传递指向结构体的指针而不需复制每个成员到栈中**

```
void stu_phone1 ( struct cont_info *s_info_ptr)    按地址调用
{
   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
{
   printf ( "%s phone number: %s", s_info.name, s_info.phone);
}
```

结构体数据的分配和访问

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

- **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_info).name可写成

  stu_info->name，执行

  以下两条指令后:

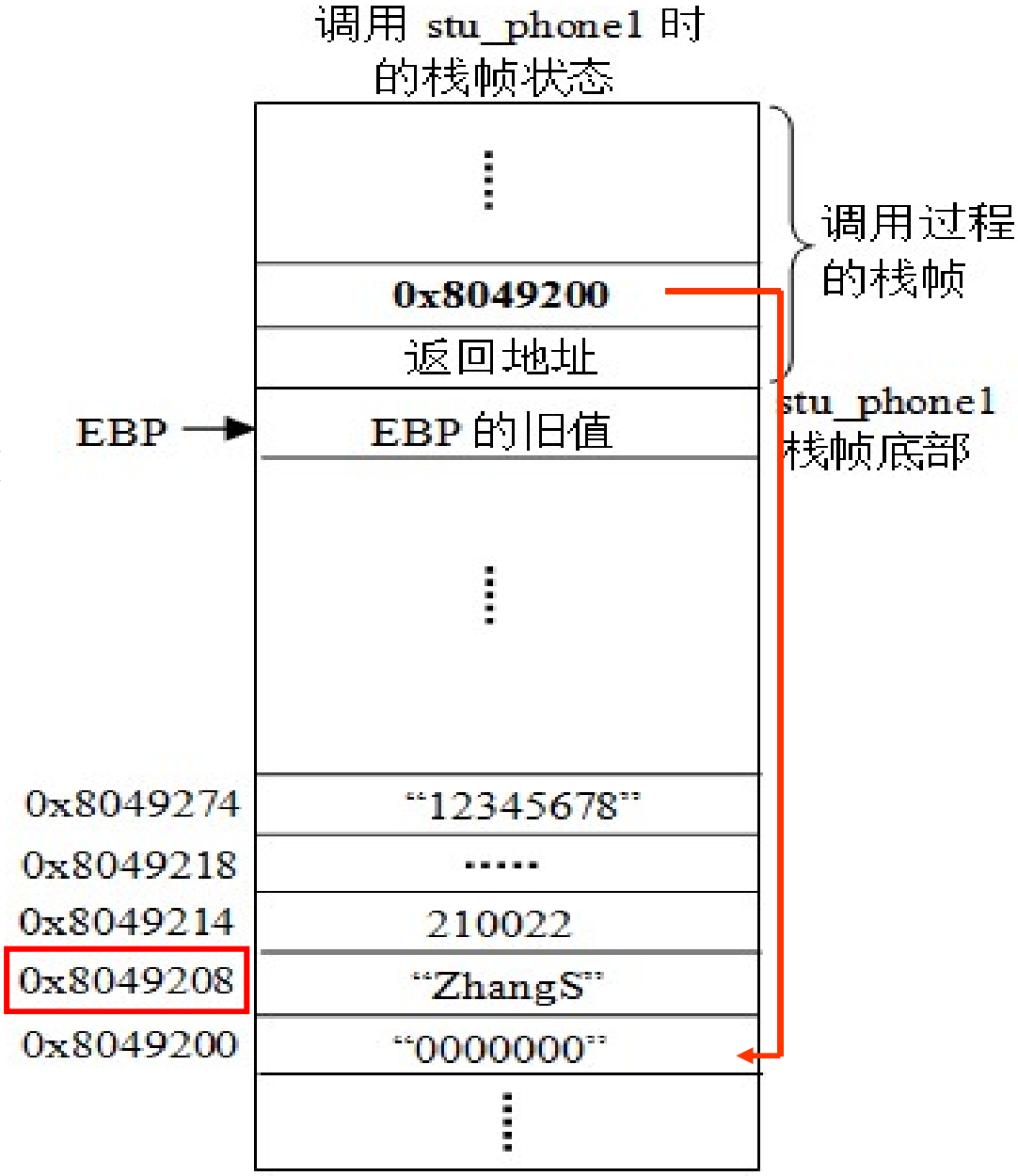
  movl 8(%ebp), %edx

  leal 8(%edx), %eax

  EAX中存放的是字符串

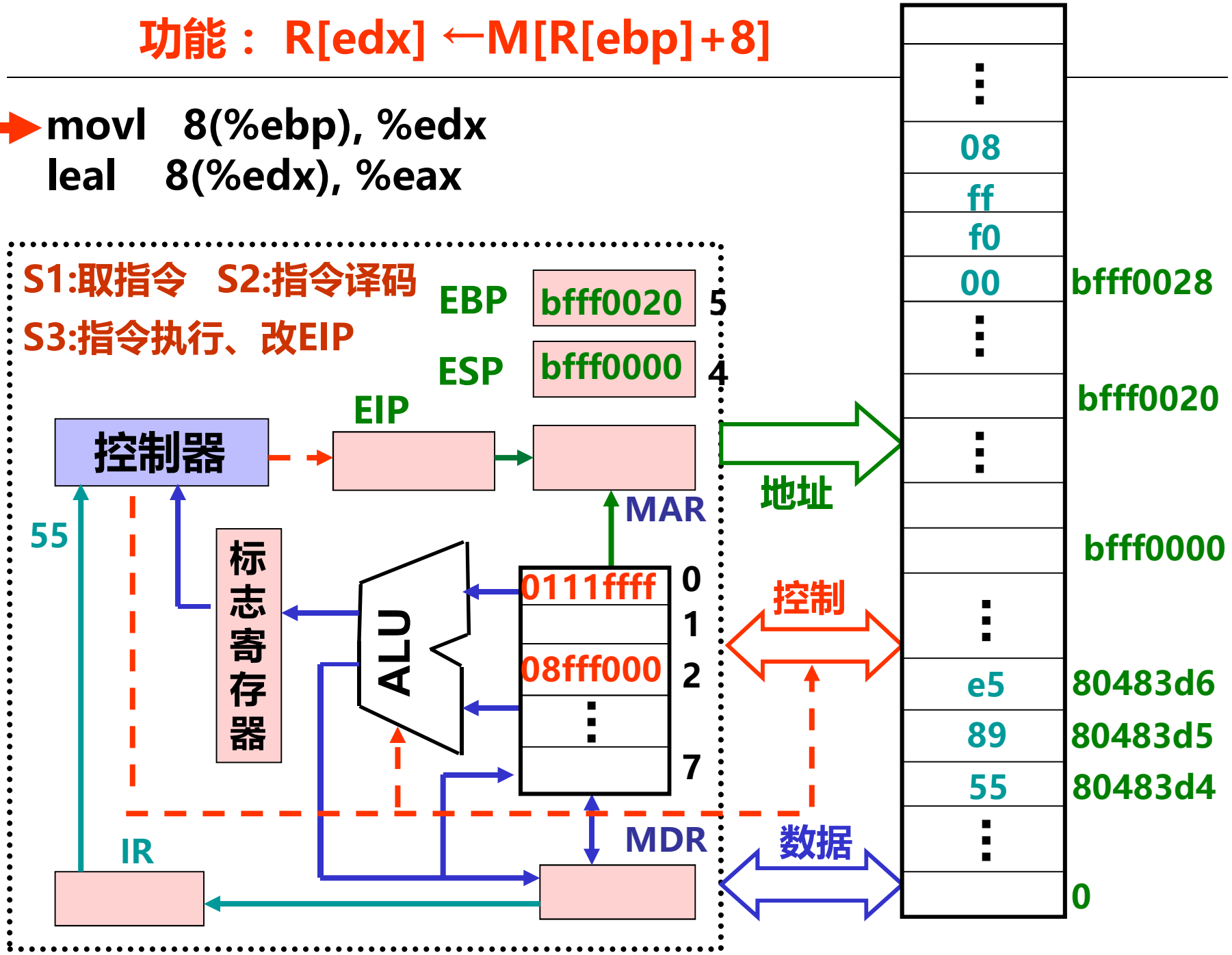
  “ZhangS”在静态存储区内的首地址

  0x8049208



功能： R[edx] ←M[R[ebp]+8]

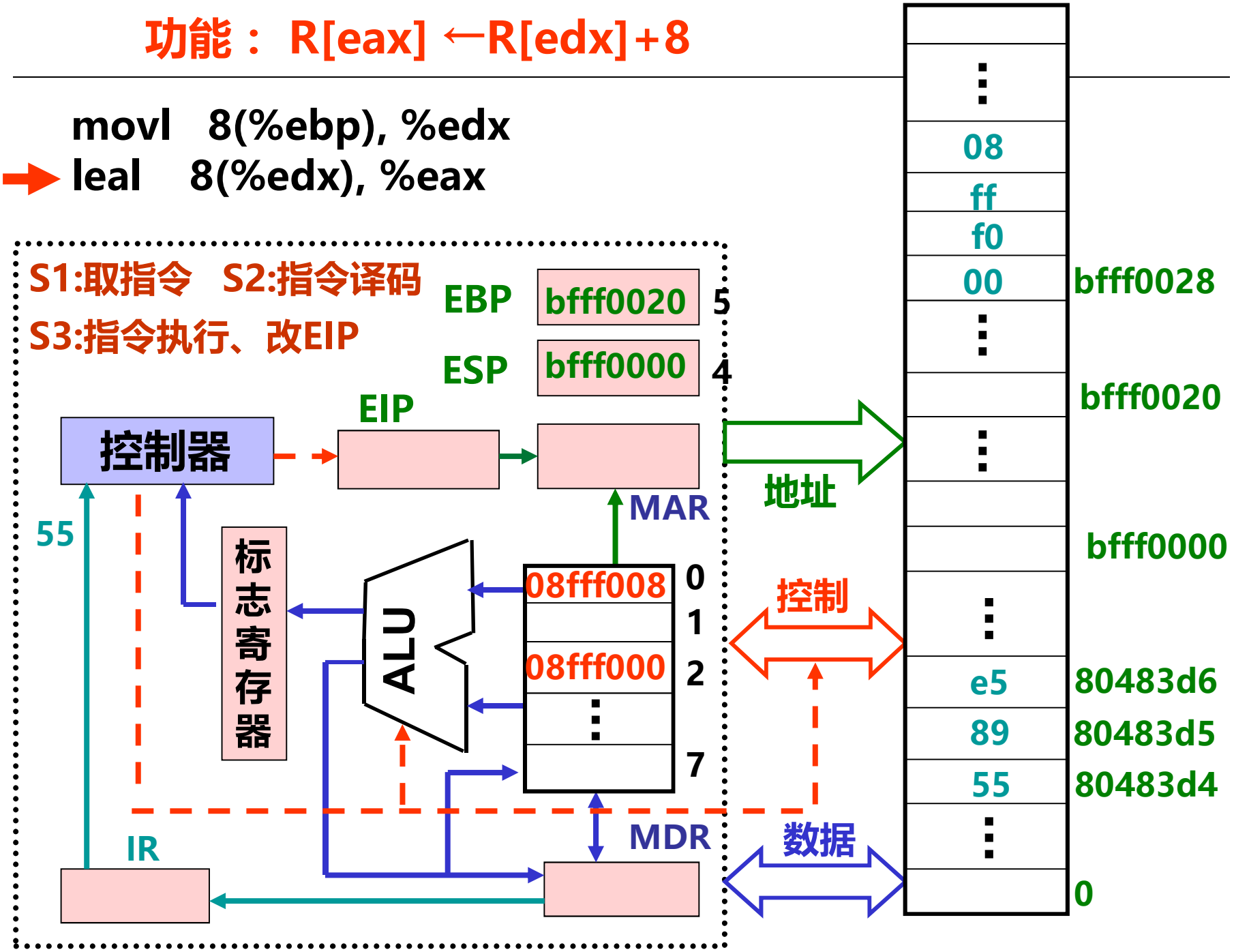
movl 8(%ebp), %edx
leal 8(%edx), %eax



功能： R[eax] ←R[edx]+8

movl 8(%ebp), %edx

leal 8(%edx), %eax



结构体数据的分配和访问

- **按值传递参数**

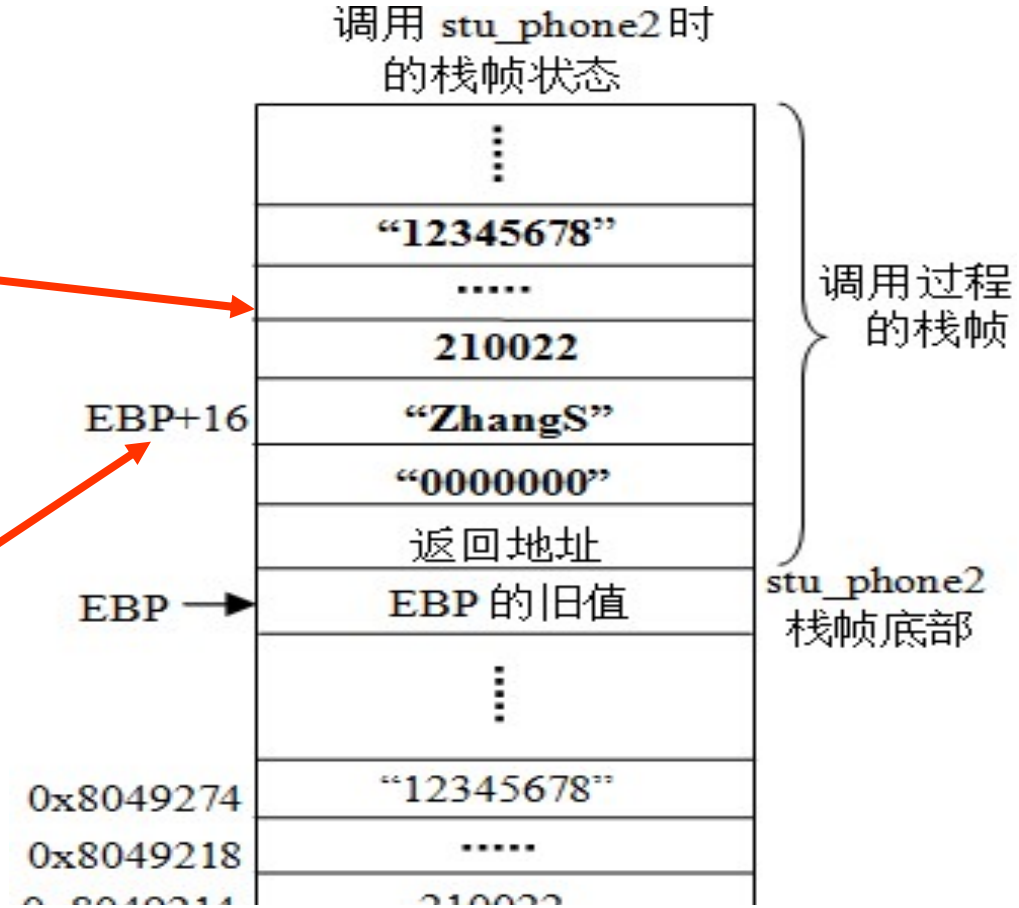
  x所有成员值作为实参存到参数区。

  stu_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，但两者的时、空开销都不一样。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**

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）

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寄存器组织的模拟

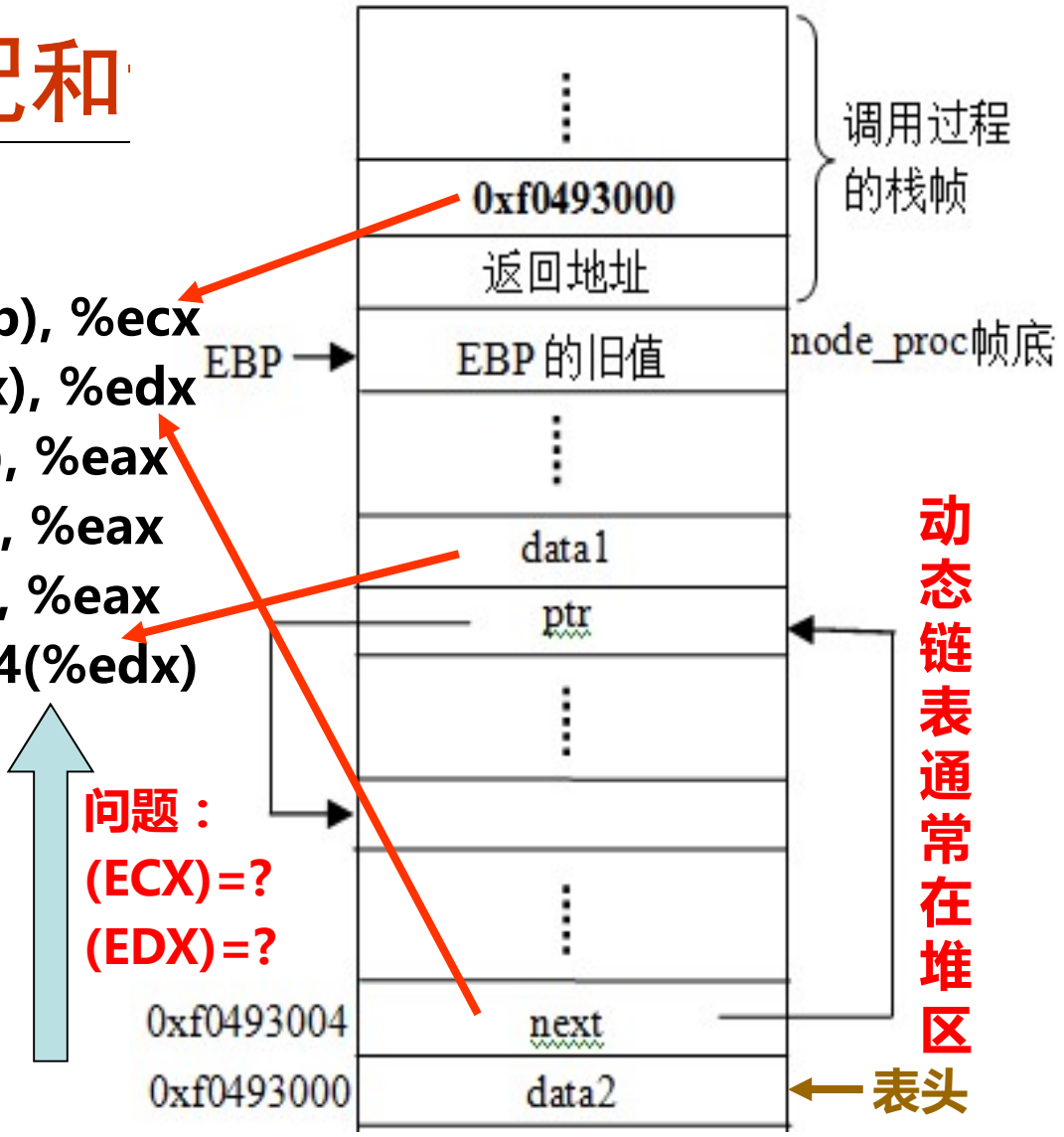
```
typedef struct{
union{
      struct {        uint32_t  eax;
                      uint32_t  ecx;
                      uint32_t  edx;
                      uint32_t  ebx;
                      uint32_t  esp;
                      uint32_t  ebp;
                      uint32_t  esi;
              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

#define reg_l(index)   (cpu.gpr[index]._32)
#define reg_w(index)  (cpu.gpr[index]._16)
#define reg_b(index)  (cpu.gpr[index & 0x3]._8[index >> 2])
```

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;
}
```