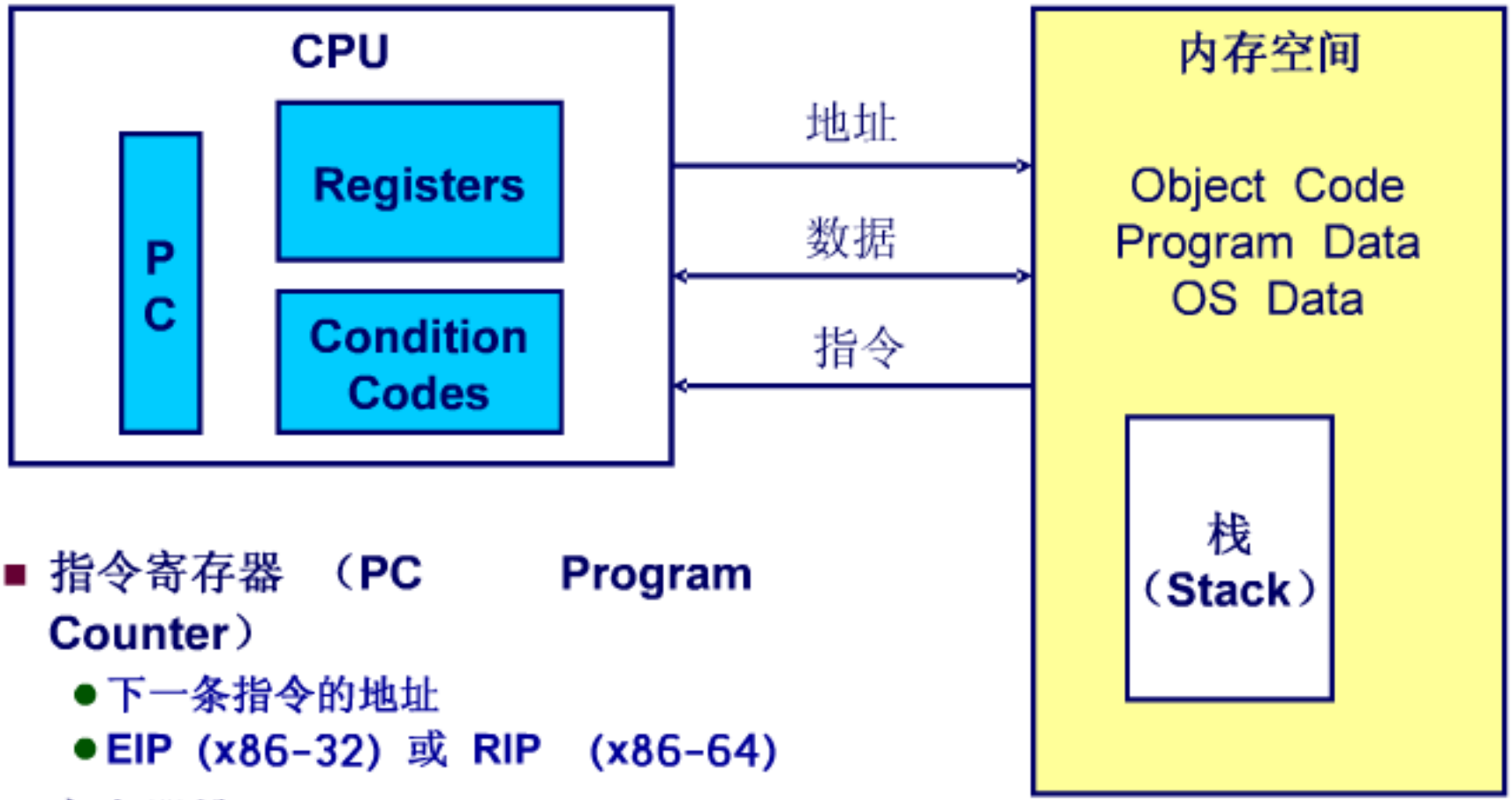
80X86汇编语言与C语言-1

数据传送指令
地址计算指令
64位模式

汇编程序员眼中的处理器系统结构



- 指令寄存器 （PC Program Counter）
  - 下一条指令的地址
  - EIP（x86-32）或 RIP （x86-64）
- 寄存器堆
- 条件码
  - 用于存储最近执行指令的结果状态信息
  - 用于条件跳转指令的判断

- 内存空间
  - 以字节编码的连续存储空间
  - 存放程序代码、数据、运行栈以及操作系统数据

如何从C代码生成汇编代码

C 代码

```
int sum(int x, int y)
{
  int t = x+y;
  return t;
}
```

命令行：gcc -O2 -S code.c

生成汇编文件code.s

对应的X86-32 汇编（AT&T汇编格式）

```
sum:
    pushl %ebp
    movl  %esp, %ebp
    movl  12(%ebp), %eax
    movl  8(%ebp), %edx
    addl  %edx, %eax
    leave
    ret
```

leave指令等价于：

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

汇编语言数据格式

C声明	Intel数据类型	汇编代码后缀	大小（字节）
char	字节	b	1
short	字	w	2
int	双字	l	4
long int	双字	l	4
long long int	—	—	4
char *	双字	l	4
float	单精度	s	4
double	双精度	l	8
long double	扩展精度	t	10/12

在X86-32中，使用“字（word）”来表示16位整数类型，“双字”表示32位。

汇编语言中没有数据类型，一般采用汇编指令的后缀来进行区分。

第一条汇编指令实例

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

C代码

- 两个整数（32位）相加

```
addl %edx, %eax
```

汇编代码

- 两个32位整数相加
  - “l” 后缀表示是双字运算
  - 无符号/带符号整数加法运算的指令是一样的
- 操作数:
  - x: Register eax
  - y: Register edx
  - t: Register eax
    - » 结果存于 eax

类似于表达式:

```
x += y
```

或者

```
int eax;
int edx
eax += edx
```

```
a:    01 d0    add %edx,%eax
```

机器码

- 2-字节指令

数据传送指令（mov）

数据传送（AT&T 语法）

`movl` *Source*, *Dest*:

- 将一个“双字”从Source移到Dest
- 常见指令

允许的操作数类型

- 立即数：常整数
  - 如：`$0x400`, `$-533`
  - 可以被1,2或4个字节来表示
- 寄存器：8个通用寄存器之一

- 存储器：四个连续字节
  - 支持多种访存寻址模式

数据传送指令支持的不同操作数类型组合

	源操作数	目的操作数		类似的C语言表示
movl	*Imm*	*Reg*	`movl $0x7,%eax`	`temp = 0x7;`
		Mem	`movl $189,(%eax)`	`*p = 189;`
	Reg	*Reg*	`movl %eax,%edx`	`temp2 = temp1;`
		Mem	`movl %eax,(%edx)`	`*p = temp;`
	Mem	*Reg*	`movl (%eax),%edx`	`temp = *p;`

但是不能两个操作数都为内存地址！

简单的寻址模式

间接寻址　　(R)　　Mem[Reg[R]]

- 寄存器R指定内存地址

```
movl (%ecx),%eax
```

基址+偏移量 寻址　　D(R)Mem[Reg[R]+D]

- 寄存器R指定内存起始地址
- 常数D给出偏移量

```
movl 8(%ebp),%edx
```

寻址模式使用实例

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    pushl %ebp
    movl  %esp,%ebp
    pushl %ebx

    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)

    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

Set Up

Body

Finish

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    pushl %ebp
    movl  %esp,%ebp
    pushl %ebx

    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)

    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

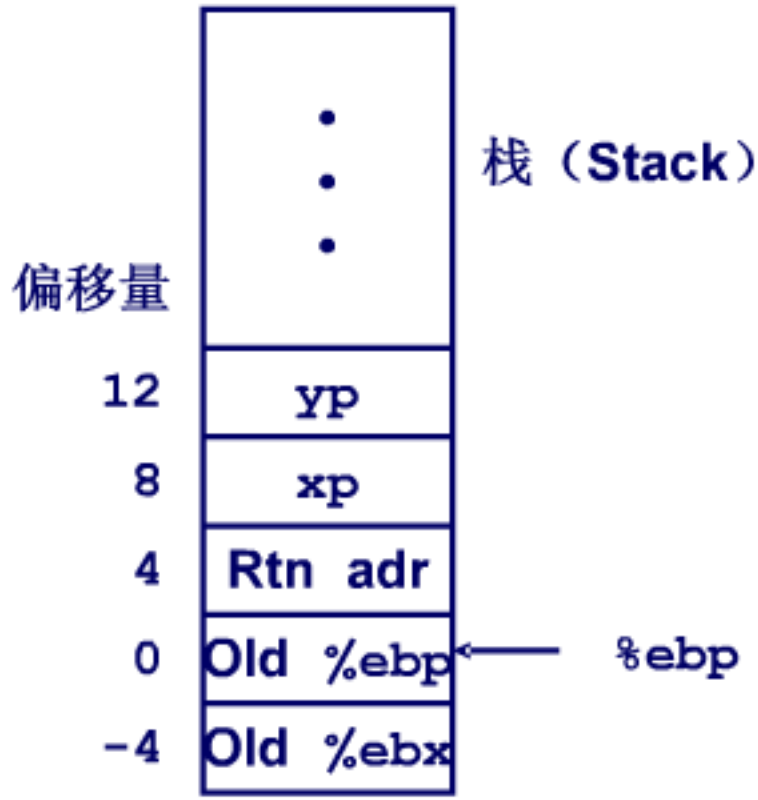
Set Up

Body

Finish

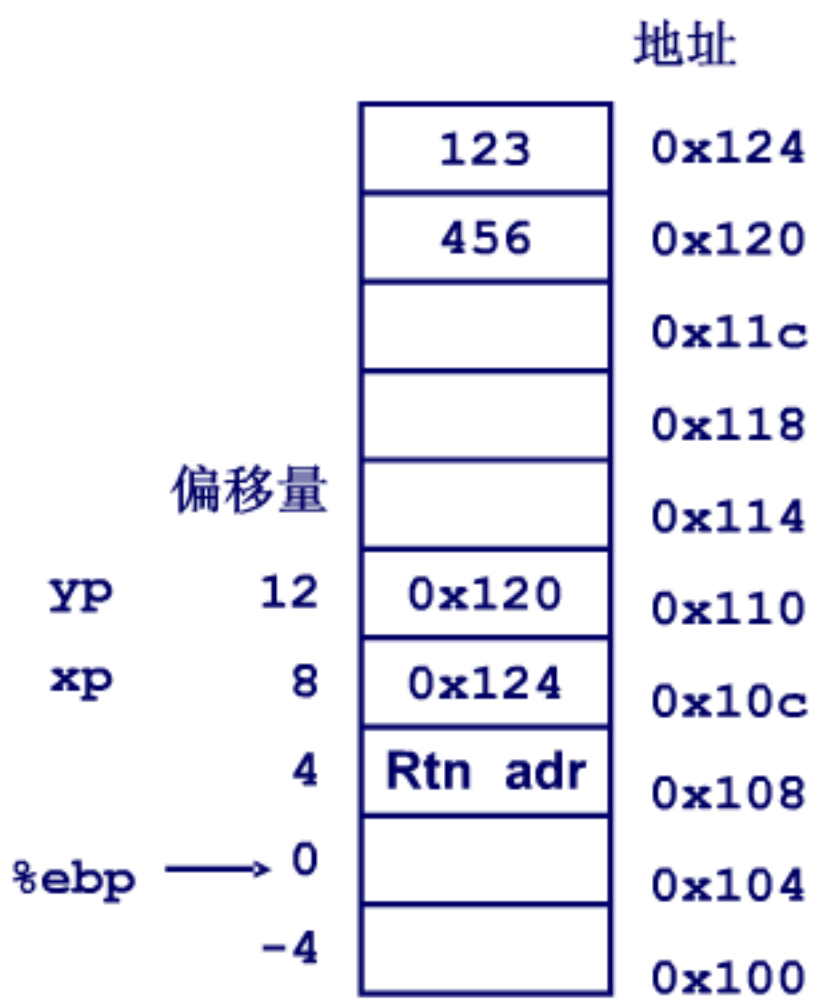
实例分析

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

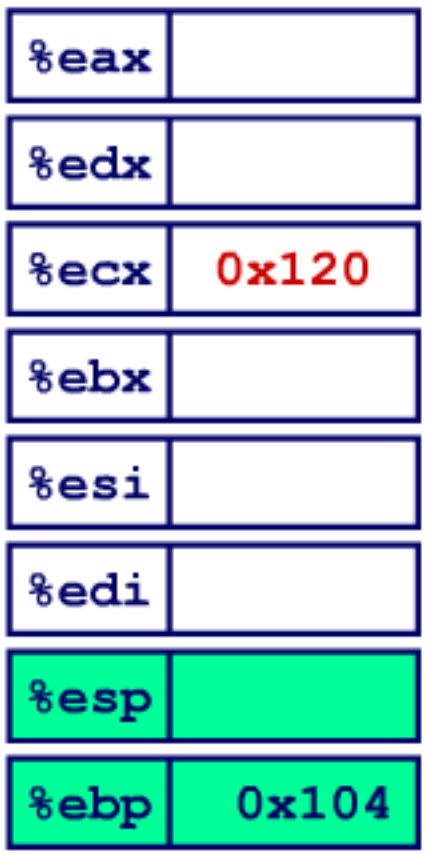


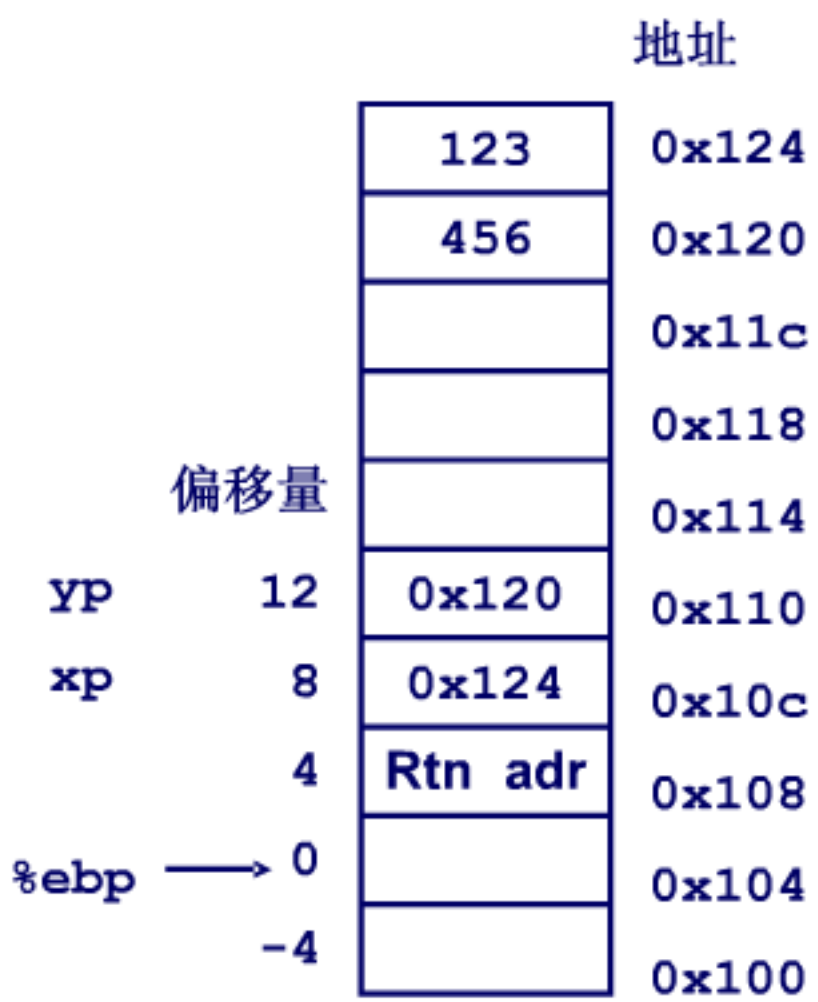
Register	Variable
%ecx	yp
%edx	xp
%eax	t1
%ebx	t0

```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```

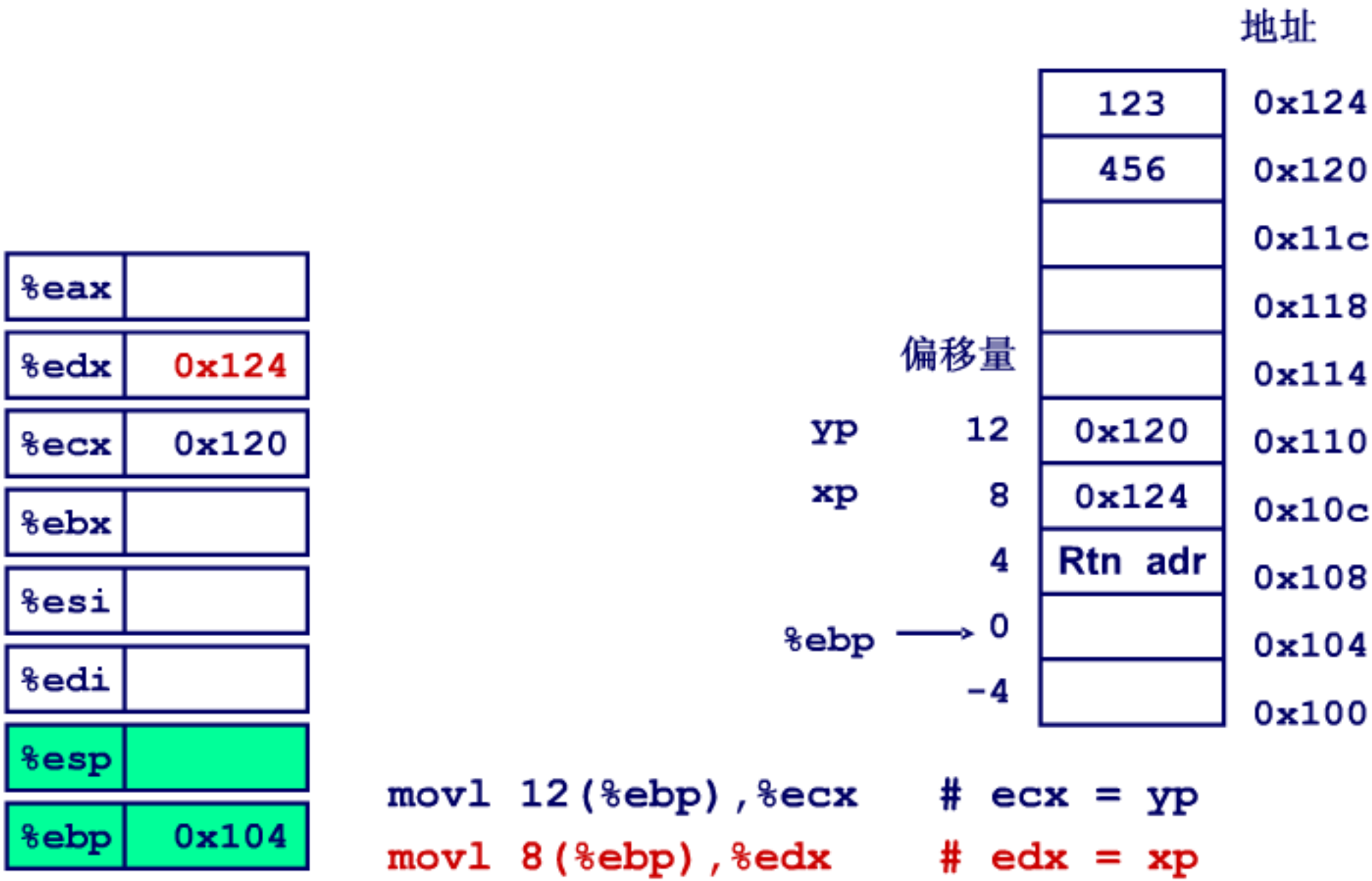



```
movl 12(%ebp),%ecx    # ecx = yp
movl 8(%ebp),%edx     # edx = xp
movl (%ecx),%eax      # eax = *yp (t1)
movl (%edx),%ebx      # ebx = *xp (t0)
movl %eax,(%edx)      # *xp = eax
movl %ebx,(%ecx)      # *yp = ebx
```

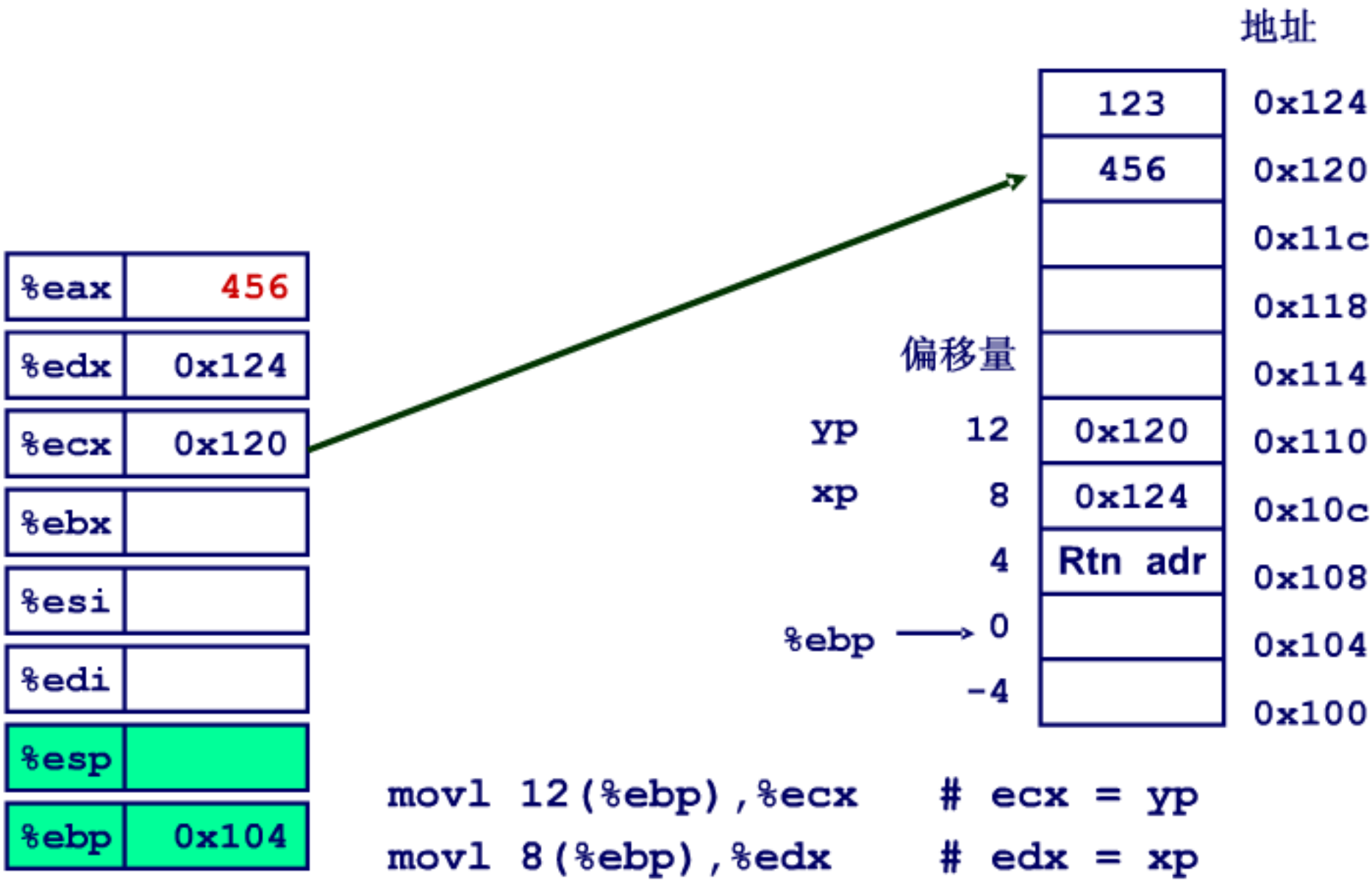




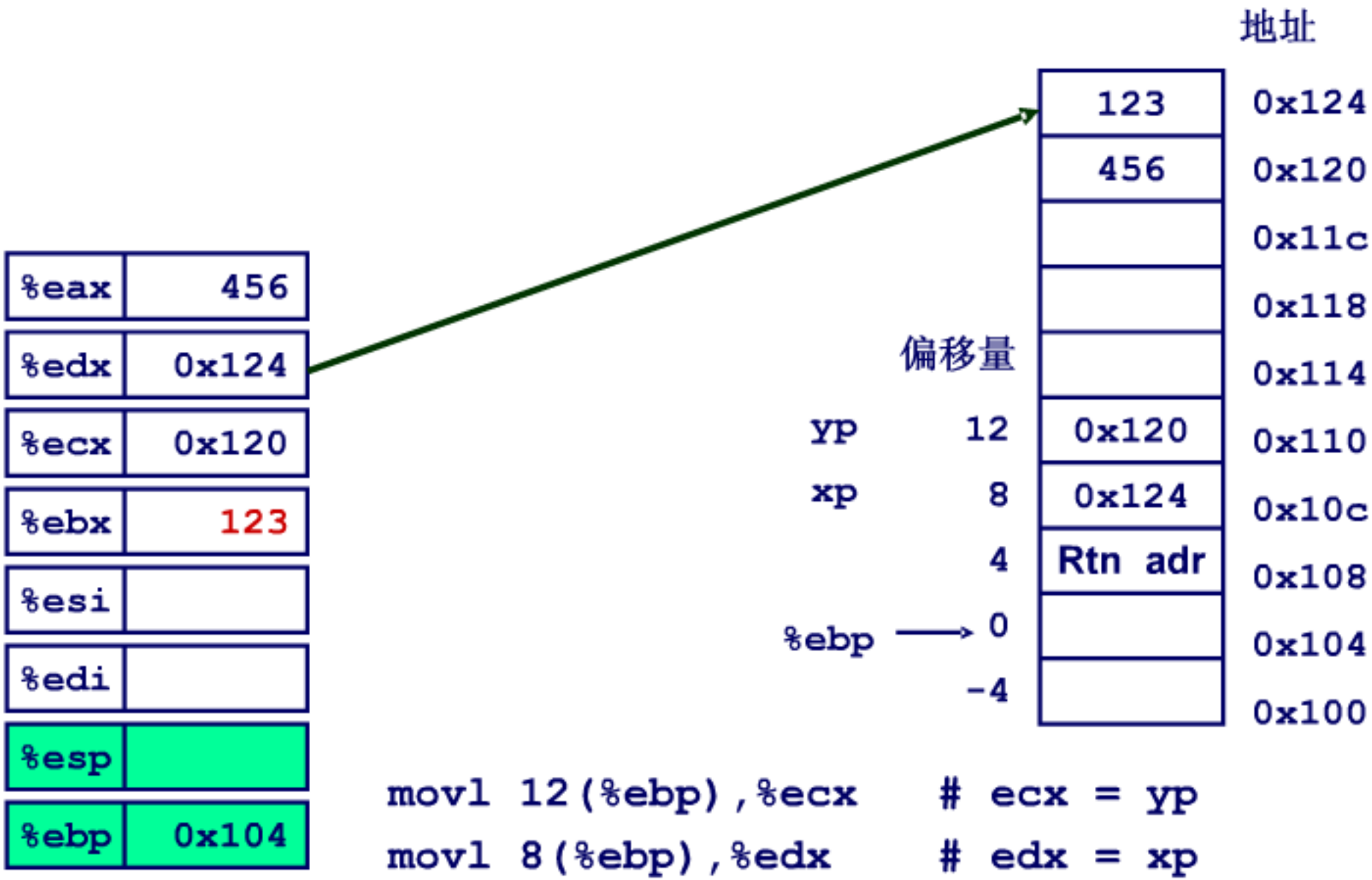

```
movl 12(%ebp),%ecx    # ecx = yp
movl 8(%ebp),%edx     # edx = xp
movl (%ecx),%eax      # eax = *yp (t1)
movl (%edx),%ebx      # ebx = *xp (t0)
movl %eax,(%edx)      # *xp = eax
movl %ebx,(%ecx)      # *yp = ebx
```



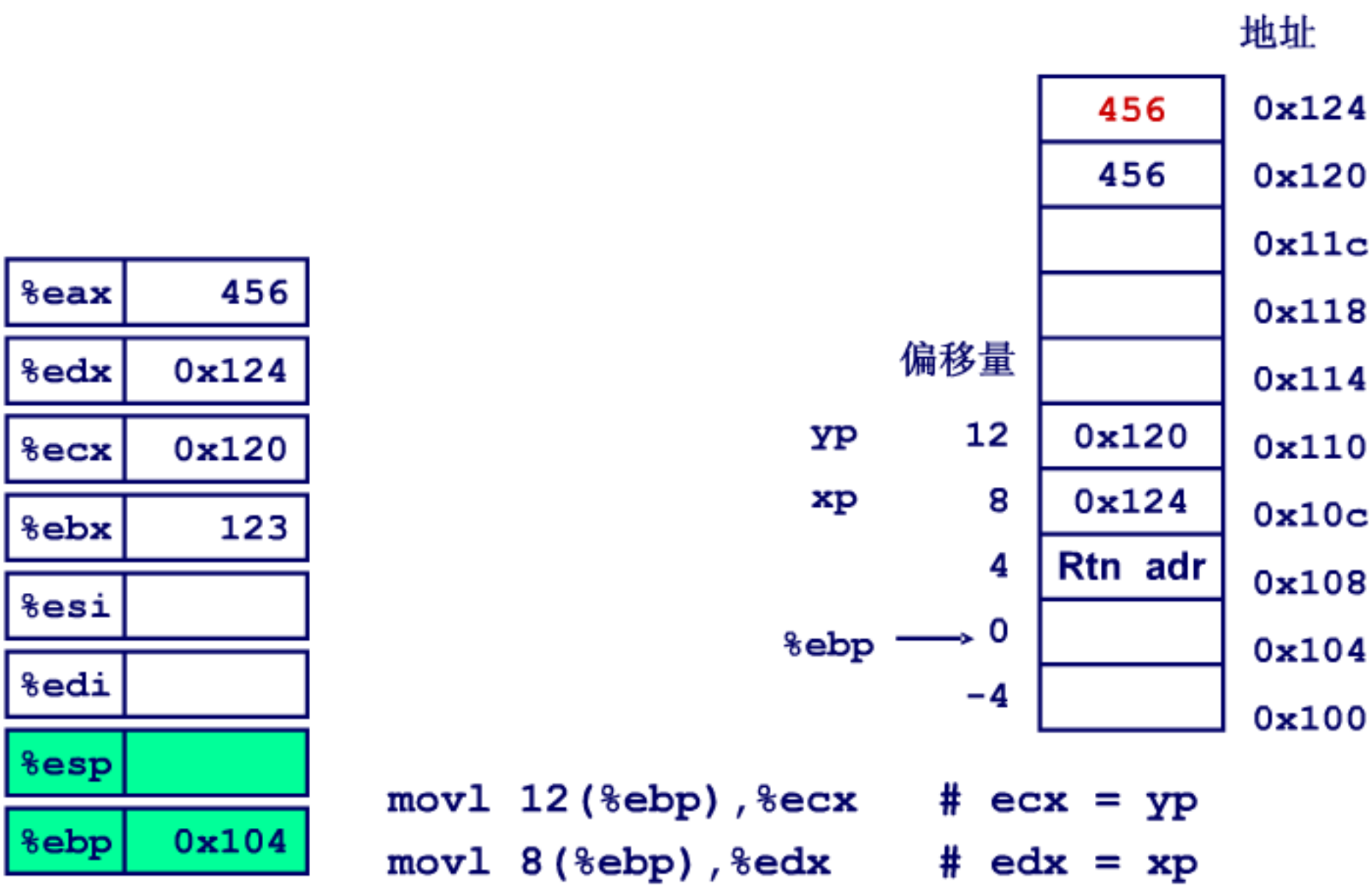

```
movl 12(%ebp),%ecx    # ecx = yp
movl 8(%ebp),%edx     # edx = xp
movl (%ecx),%eax      # eax = *yp (t1)
movl (%edx),%ebx      # ebx = *xp (t0)
movl %eax,(%edx)      # *xp = eax
movl %ebx,(%ecx)      # *yp = ebx
```



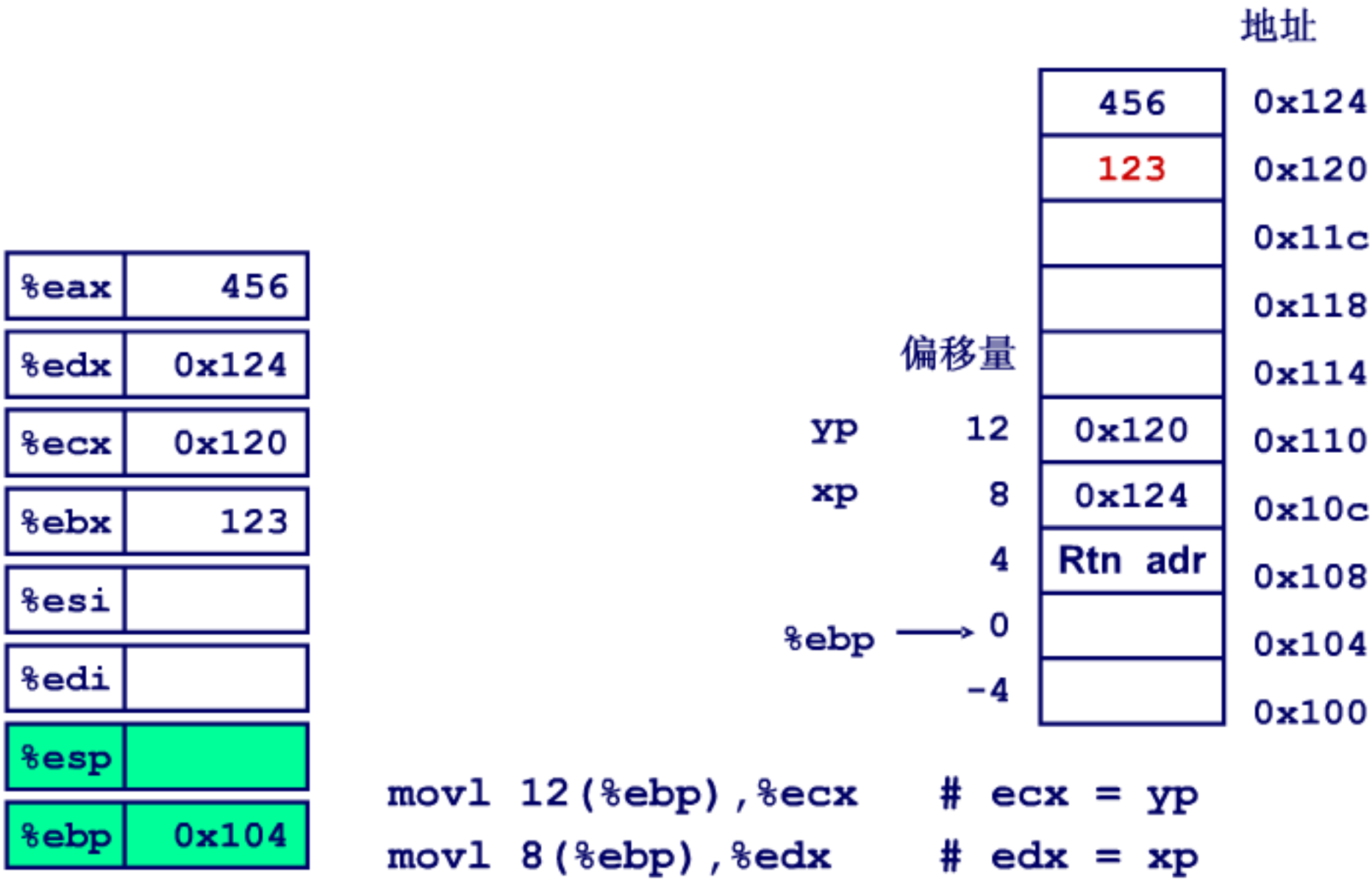

```
movl 12(%ebp),%ecx    # ecx = yp
movl 8(%ebp),%edx     # edx = xp
movl (%ecx),%eax      # eax = *yp (t1)
movl (%edx),%ebx      # ebx = *xp (t0)
movl %eax,(%edx)      # *xp = eax
movl %ebx,(%ecx)      # *yp = ebx
```




```
movl 12(%ebp),%ecx     # ecx = yp
movl 8(%ebp),%edx      # edx = xp
movl (%ecx),%eax       # eax = *yp (t1)
movl (%edx),%ebx       # ebx = *xp (t0)
movl %eax,(%edx)       # *xp = eax
movl %ebx,(%ecx)       # *yp = ebx
```




```
movl 12(%ebp),%ecx     # ecx = yp
movl 8(%ebp),%edx      # edx = xp
movl (%ecx),%eax       # eax = *yp (t1)
movl (%edx),%ebx       # ebx = *xp (t0)
movl %eax,(%edx)       # *xp = eax
movl %ebx,(%ecx)       # *yp = ebx
```




```
movl 12(%ebp),%ecx     # ecx = yp
movl 8(%ebp),%edx      # edx = xp
movl (%ecx),%eax       # eax = *yp (t1)
movl (%edx),%ebx       # ebx = *xp (t0)
movl %eax,(%edx)       # *xp = eax
movl %ebx,(%ecx)       # *yp = ebx
```

寻址模式

通用形式

D(Rb,Ri,S)　　Mem[Reg[Rb]+S*Reg[Ri]+ D]

- D:　常量（地址偏移量）
- Rb:　基址寄存器: 8个通用寄存器之一
- Ri:　索引寄存器: %esp不作为索引寄存器
  - » 一般 %ebp也不用做这个用途
- S:　比例因子 1, 2, 4, or 8

变形

(Rb,Ri)　　Mem[Reg[Rb]+Reg[Ri]]

D(Rb,Ri)　　Mem[Reg[Rb]+Reg[Ri]+D]

(Rb,Ri,S)　　Mem[Reg[Rb]+S*Reg[Ri]]

寻址模式实例

注意：如果是$0x8,则表示立即数；否则就是内存地址

%edx	0xf200
%ecx	0x100

地址表达式	地址计算	访存地址
0x8	0x8	0x8
0x8(%edx)	0xf200 + 0x8	0xf208
(%edx,%ecx)	0xf200 + 0x100	0xf300
(%edx,%ecx,4)	0xf200 + 4*0x100	0xf600
0x80(,%edx,2)	2*0xf200 + 0x80	0x1e480

指令		效果	描述
MOV	*S*, *D*	*D* ← *S*	传送
movb		传送字节	
movw		传送字	
movl		传送双字	
MOVS	*S*, *D*	*D* ← 符号扩展 (*S*)	传送符号扩展的字节
movsbw		将做了符号扩展的字节传送到字	
movsbl		将做了符号扩展的字节传送到双字	
movswl		将做了符号扩展的字传送到双字	
MOVZ	*S*, *D*	*D* ← 零扩展 (*S*)	传送零扩展的字节
movzbw		将做了零扩展的字节传送到字	
movzbl		将做了零扩展的字节传送到双字	
movzwl		将做了零扩展的字传送到双字	
pushl	*S*	R[%esp] ← R[%esp]-4； M[R[%esp]] ← *S*	将双字压栈
popl	*D*	*D* ← M[R[%esp]]; R[%esp] ← R[%esp]+4	将双字出栈

地址计算指令

`leal` ***Src,Dest***

- ***Src*** 是地址计算表达式
- 计算出来的地址赋给 ***Dest***

使用实例

- 地址计算（无需访存）
  - 比如计算元素的地址（`p = &x[i];`）
- 进行**x + k*y**这一类型的整数计算
  - **k = 1, 2, 4, or 8.**

整数计算指令

指令格式		计算	
双操作数指令			
`addl`	*Src,Dest*	*Dest = Dest + Src*	
`subl`	*Src,Dest*	*Dest = Dest - Src*	
`imull`	*Src,Dest*	*Dest = Dest * Src*	
`sall`	*Src,Dest*	*Dest = Dest << Src*	与`shll`等价
`sarl`	*Src,Dest*	*Dest = Dest >> Src*	算术右移
`shrl`	*Src,Dest*	*Dest = Dest >> Src*	逻辑右移
`xorl`	*Src,Dest*	*Dest = Dest ^ Src*	
`andl`	*Src,Dest*	*Dest = Dest & Src*	
`orl`	*Src,Dest*	*Dest = Dest | Src*	

指令格式	计算
单操作数指令	
`incl` *Dest*	*Dest* = *Dest* + 1
`decl` *Dest*	*Dest* = *Dest* – 1
`negl` *Dest*	*Dest* = – *Dest*
`notl` *Dest*	*Dest* = ~ *Dest*

将leal指令用于计算（实例1）

```
int arith
   (int x, int y, int z)
{
   int t1 = x+y;
   int t2 = z+t1;
   int t3 = x+4;
   int t4 = y * 48;
   int t5 = t3 + t4;
   int rval = t2 * t5;
   return rval;
}
```

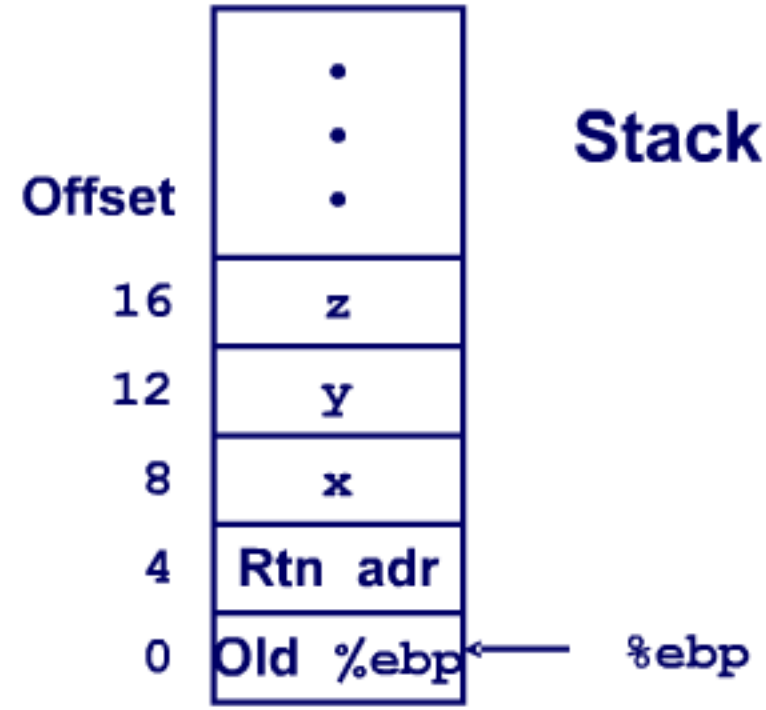
```
arith:
    pushl %ebp
    movl %esp,%ebp

    movl 8(%ebp),%eax
    movl 12(%ebp),%edx
    leal (%edx,%eax),%ecx
    leal (%edx,%edx,2),%edx
    sall $4,%edx
    addl 16(%ebp),%ecx
    leal 4(%edx,%eax),%eax
    imull %ecx,%eax

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

Set Up

Body

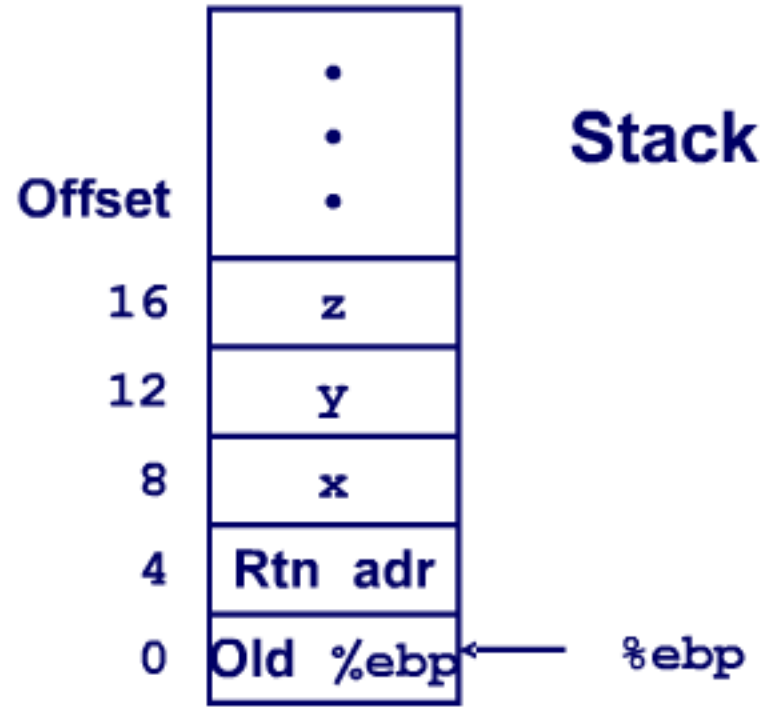
Finish

```
int arith
   (int x, int y, int z)
{
   int t1 = x+y;
   int t2 = z+t1;
   int t3 = x+4;
   int t4 = y * 48;
   int t5 = t3 + t4;
   int rval = t2 * t5;
   return rval;
}
```



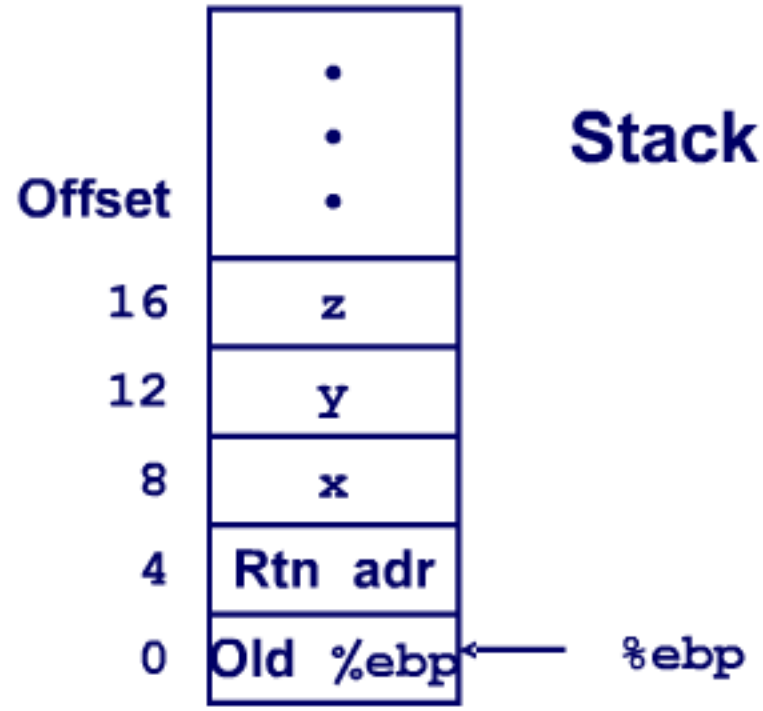

```
movl 8(%ebp),%eax          # eax = x
movl 12(%ebp),%edx         # edx = y
leal (%edx,%eax),%ecx      # ecx = x+y   (t1)
leal (%edx,%edx,2),%edx    # edx = 3*y
sall $4,%edx               # edx = 48*y (t4)
addl 16(%ebp),%ecx         # ecx = z+t1 (t2)
leal 4(%edx,%eax),%eax     # eax = 4+t4+x (t5)
imull %ecx,%eax            # eax = t5*t2 (rval)
```

```
int arith
   (int x, int y, int z)
{
   int t1 = x+y;
   int t2 = z+t1;
   int t3 = x+4;
   int t4 = y * 48;
   int t5 = t3 + t4;
   int rval = t2 * t5;
   return rval;
}
```



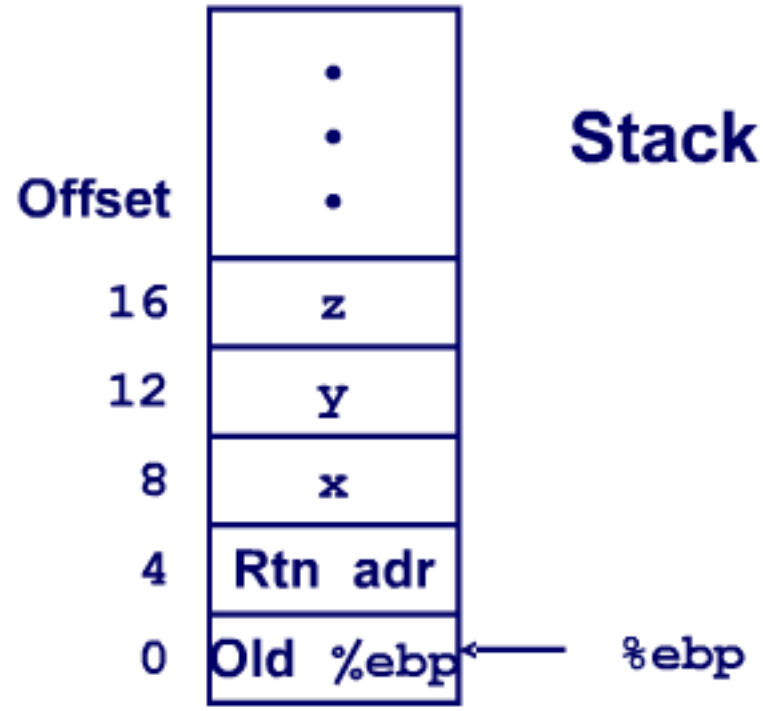

```
movl 8(%ebp),%eax          # eax = x
movl 12(%ebp),%edx         # edx = y
leal (%edx,%eax),%ecx      # ecx = x+y   (t1)
leal (%edx,%edx,2),%edx    # edx = 3*y
sall $4,%edx               # edx = 48*y (t4)
addl 16(%ebp),%ecx         # ecx = z+t1 (t2)
leal 4(%edx,%eax),%eax     # eax = 4+t4+x (t5)
imull %ecx,%eax            # eax = t5*t2 (rval)
```

```
int arith
   (int x, int y, int z)
{
   int t1 = x+y;
   int t2 = z+t1;
   int t3 = x+4;
   int t4 = y * 48;
   int t5 = t3 + t4;
   int rval = t2 * t5;
   return rval;
}
```



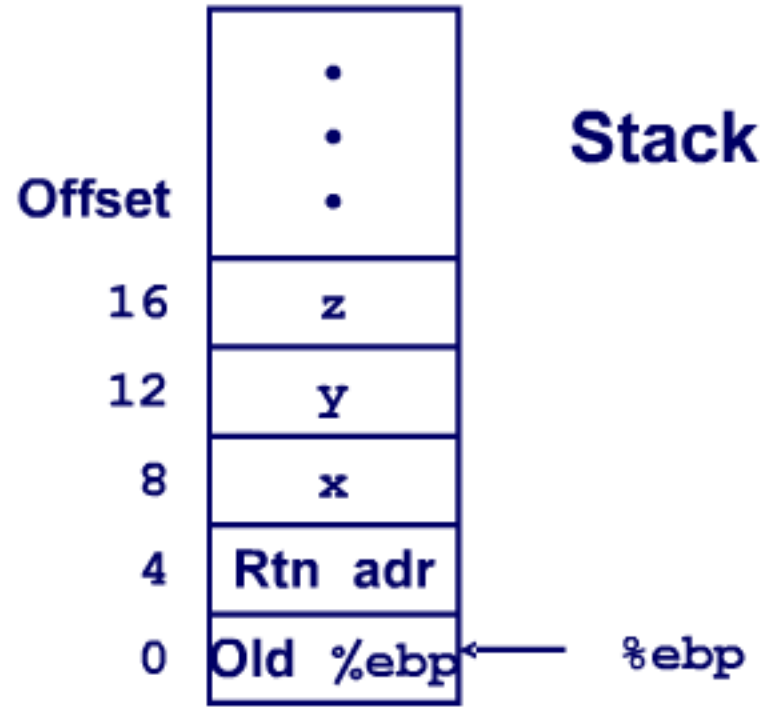

```
movl 8(%ebp),%eax          # eax = x
movl 12(%ebp),%edx         # edx = y
leal (%edx,%eax),%ecx      # ecx = x+y   (t1)
leal (%edx,%edx,2),%edx    # edx = 3*y
sall $4,%edx               # edx = 48*y (t4)
addl 16(%ebp),%ecx         # ecx = z+t1 (t2)
leal 4(%edx,%eax),%eax     # eax = 4+t4+x (t5)
imull %ecx,%eax            # eax = t5*t2 (rval)
```

```
int arith
   (int x, int y, int z)
{
   int t1 = x+y;
   int t2 = z+t1;
   int t3 = x+4;
   int t4 = y * 48;
   int t5 = t3 + t4;
   int rval = t2 * t5;
   return rval;
}
```



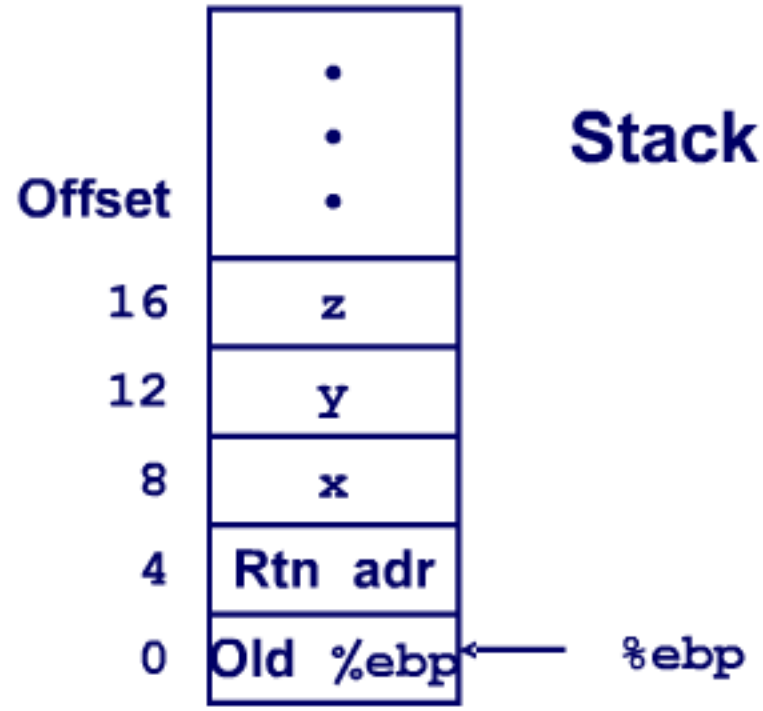

```
movl 8(%ebp),%eax          # eax = x
movl 12(%ebp),%edx         # edx = y
leal (%edx,%eax),%ecx      # ecx = x+y   (t1)
leal (%edx,%edx,2),%edx    # edx = 3*y
sall $4,%edx               # edx = 48*y (t4)
addl 16(%ebp),%ecx         # ecx = z+t1 (t2)
leal 4(%edx,%eax),%eax     # eax = 4+t4+x (t5)
imull %ecx,%eax            # eax = t5*t2 (rval)
```

```
int arith
   (int x, int y, int z)
{
   int t1 = x+y;
   int t2 = z+t1;
   int t3 = x+4;
   int t4 = y * 48;
   int t5 = t3 + t4;
   int rval = t2 * t5;
   return rval;
}
```




```
movl 8(%ebp),%eax          # eax = x
movl 12(%ebp),%edx         # edx = y
leal (%edx,%eax),%ecx      # ecx = x+y   (t1)
leal (%edx,%edx,2),%edx    # edx = 3*y
sall $4,%edx               # edx = 48*y (t4)
addl 16(%ebp),%ecx         # ecx = z+t1 (t2)
leal 4(%edx,%eax),%eax     # eax = 4+t4+x (t5)
imull %ecx,%eax            # eax = t5*t2 (rval)
```

```
int arith
   (int x, int y, int z)
{
   int t1 = x+y;
   int t2 = z+t1;
   int t3 = x+4;
   int t4 = y * 48;
   int t5 = t3 + t4;
   int rval = t2 * t5;
   return rval;
}
```




```
movl 8(%ebp),%eax          # eax = x
movl 12(%ebp),%edx         # edx = y
leal (%edx,%eax),%ecx      # ecx = x+y   (t1)
leal (%edx,%edx,2),%edx    # edx = 3*y
sall $4,%edx               # edx = 48*y (t4)
addl 16(%ebp),%ecx         # ecx = z+t1 (t2)
leal 4(%edx,%eax),%eax     # eax = 4+t4+x (t5)
imull %ecx,%eax            # eax = t5*t2 (rval)
```

实例2

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

```
logical:
    pushl %ebp
    movl %esp,%ebp

    movl 8(%ebp),%eax
    xorl 12(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax

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

Set Up

Body

Finish

```
movl 8(%ebp),%eax      eax = x
xorl 12(%ebp),%eax     eax = x^y
sarl $17,%eax          eax = t1>>17
andl $8185,%eax        eax = t2 & 8185
```

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

```
logical:
    pushl %ebp
    movl %esp,%ebp

    movl 8(%ebp),%eax
    xorl 12(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax

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

Set Up

Body

Finish

```
movl 8(%ebp),%eax      eax = x
xorl 12(%ebp),%eax     eax = x^y      (t1)
sarl $17,%eax          eax = t1>>17   (t2)
andl $8185,%eax        eax = t2 & 8185
```

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

```
logical:
    pushl %ebp
    movl %esp,%ebp

    movl 8(%ebp),%eax
    xorl 12(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax

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

Set Up

Body

Finish

```
movl 8(%ebp),%eax      eax = x
xorl 12(%ebp),%eax     eax = x^y      (t1)
sarl $17,%eax          eax = t1>>17   (t2)
andl $8185,%eax        eax = t2 & 8185
```

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

$2^{13} = 8192,\ 2^{13} - 7 = 8185$

```
logical:
    pushl %ebp
    movl %esp,%ebp

    movl 8(%ebp),%eax
    xorl 12(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax

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

Set Up

Body

Finish

```
movl 8(%ebp),%eax      eax = x
xorl 12(%ebp),%eax     eax = x^y      (t1)
sarl $17,%eax          eax = t1>>17   (t2)
andl $8185,%eax        eax = t2 & 8185   (rval)
```

x86-32 与 x86-64的数据类型宽度

Sizes of C Objects (in Bytes)

C Data Type	Typical 32-bit	Intel IA32	x86-64
unsigned	4	4	4
int	4	4	4
long int	4	4	8
char	1	1	1
short	2	2	2
float	4	4	4
double	8	8	8
long double	8	10/12	16
char *	4	4	8

» Or any other pointer

x86-64的通用寄存器

- 扩展现有的，并增加了8个新的
- %ebp/%rbp 不再是专用寄存器

X86-32下的swap

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    pushl %ebp
    movl  %esp,%ebp
    pushl %ebx

    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)

    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

Set Up

Body

Finish

X86-64下的...

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    movl    (%rdi), %edx
    movl    (%rsi), %eax
    movl    %eax, (%rdi)
    movl    %edx, (%rsi)
    retq
```

- 不同点
  - 参数通过寄存器来传递
    - » 第一个参数(xp) 由%rdi传递，第二个(yp) 位于 %rsi内
    - » 64位指针
  - 无栈操作
- 被操作的数据仍是32位
  - 所以使用寄存器 %eax 、%edx
  - 以及movl 指令

当参数少于7个时， 参数从左到右放入寄存器: rdi, rsi, rdx, rcx, r8, r9。当参数为 7 个以上时， 前 6 个传送方式不变， 但后面的依次从 "右向左" 放入栈中。

X86-64下long int类型的swap过程

```
void swap_l
 (long int *xp, long int *yp)
{
  long int t0 = *xp;
  long int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap_l:
    movq    (%rdi), %rdx
    movq    (%rsi), %rax
    movq    %rax, (%rdi)
    movq    %rdx, (%rsi)
    retq
```

- 被操作的数据是64位
  - 所以使用寄存器%rax 、%rdx
  - 以及movq 指令
    - » “q” 表示“4字”

小结

X86指令的特点

支持多种类型的指令操作数

- 立即数，寄存器，内存数据

算逻指令可以以内存数据为操作数

支持多种内存地址计算模式

- Rb + S*Ri + D
- 也可用于整数计算(如leal指令)

变长指令

- from 1 to 15 bytes

练习题

一个函数的原型为

```
int decode2(int x, int y, int z);
```

```
    x at %ebp+8, y at %ebp+12, z at %ebp+16
1       movl    16(%ebp), %edx
2       subl    12(%ebp), %edx
3       movl    %edx, %eax
4       sall    $15, %eax
5       sarl    $15, %eax
6       xorl    8(%ebp), %edx
7       imull   %edx, %eax
```

参数 x、y 和 z 存放在存储器中相对于寄存器 %ebp 中地址偏移量为 8、12 和 16 的地方。代码将返回值存放在寄存器 %eax 中。

写出等价于我们汇编代码的 decode2 的 C 代码。

```
int decode2(int x, int y, int z)
{
    int t1 = z - y;
    int t2 = (t1 << 15) >> 15;
    int t3 = x ^ t1;
    int t4 = t2 * t1;
    return t4;
}
```

80X86汇编语言与C语言-2

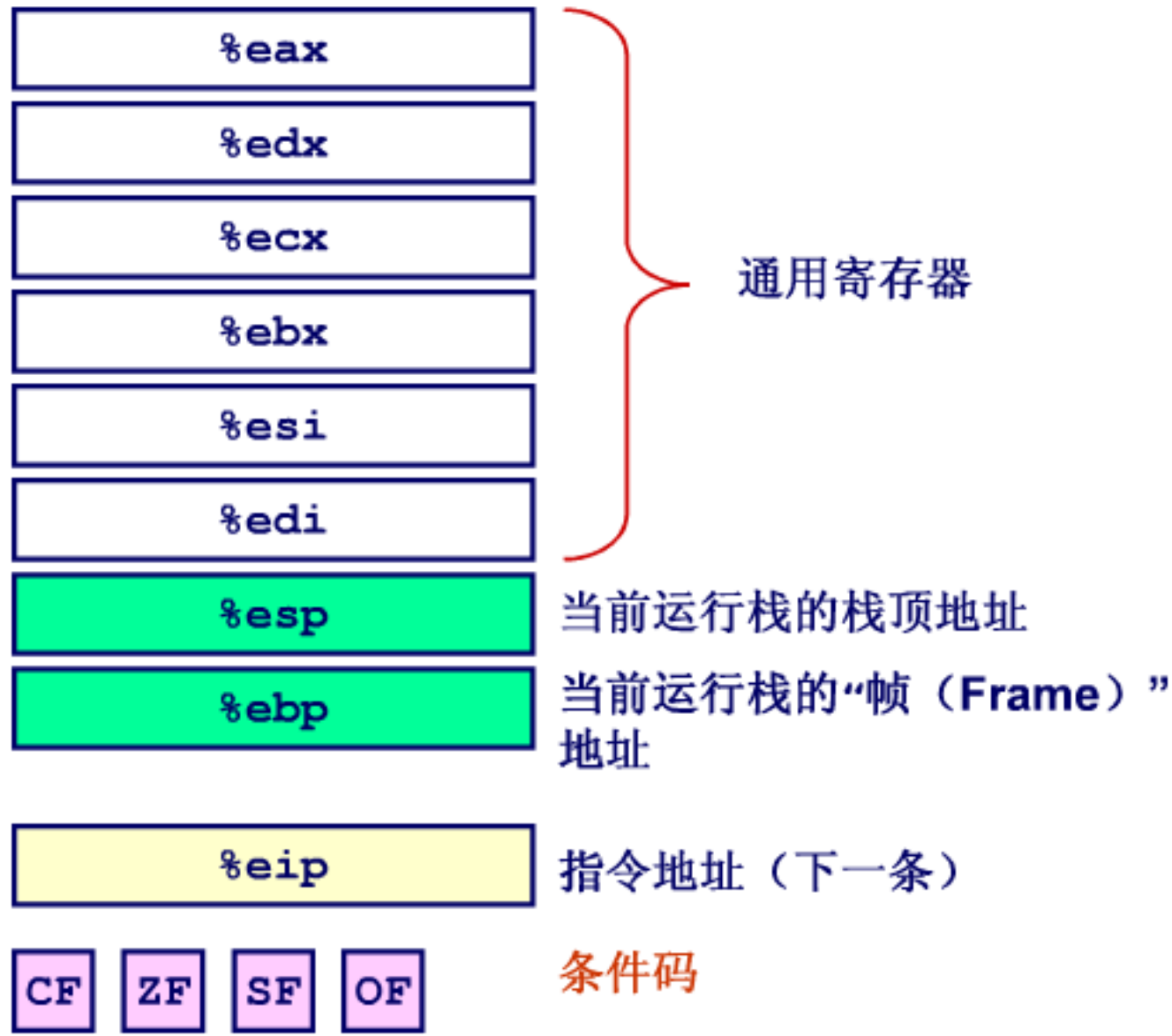
控制流

- 条件码
  - 设置
  - 读取
- 程序控制流
  - If-then-else
  - 循环结构
  - switch语句
- x86-64 模式
  - 条件传送指令
  - 循环结构的不同实现

汇编程序员眼中的系统结构（部分）

当前执行程序的信息

- 数据
- 指令地址
- 运行栈地址
- 条件码



条件码

CF 进位标志（Carry）　　SF　符号位（Sign）

ZF Zero Flag　　OF　溢出标志（Overflow ）

这些条件码由算术指令隐含设置

addl *Src,Dest*　　addq *Src,Dest*

类似的C语言表达式 ： t = a + b　　(a = Src, b = Dest)

- CF 进位标志
  - 可用于检测无符号整数运算的溢出
- 如果 t == 0，那么ZF=1;否则ZF=0
- 如果t < 0，那么SF=1; 否则SF=0
- 如果补码运算溢出，那么OF=1（即带符号整数运算）

```
(a>0 && b>0 && t<0)
|| (a<0 && b<0 && t>=0)
```

比较（Compare）指令

`cmpl` *Src2,Src1*　　`cmpq` *Src2,Src1*

- `cmpl b,a` 类似于计算a-b（但是不改变目的操作数）
- 如果向最高位有借位，那么CF=1;否则CF=0
  - 可用于无符号数的比较
- 如果a == b，那么ZF=1；否则ZF=0
- 如果(a - b)<0，那么SF=1；否则SF=0
  - 即运算后若结果最高位为1，那么SF=1；否则为0
- 如果补码运算溢出，那么OF=1
  - `(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)`

测试（Test）指令

`testl` *Src2,Src1*

`testq` *Src2,Src1*

- 计算*Src1* & *Src2*并设置相应的条件码，但是不改变目的操作数

- 如果`a&b == 0`，那么`ZF=1`；否则为`0`
- 如果`a&b < 0` ，那么`SF=1`；否则为`0`
  - 即运算后结果最高位为`1`，那么`SF=1`；否则为`0`

读取条件码

SetX 指令

- 读取当前的条件码（或者某些条件码的组合），并存入目的字节寄存器

SetX	Condition	Description
sete	ZF	Equal / Zero
setne	~ZF	Not Equal / Not Zero
sets	SF	Negative
setns	~SF	Nonnegative
setg	~(SF^OF)&~ZF	Greater (Signed)
setge	~(SF^OF)	Greater or Equal (Signed)
setl	(SF^OF)	Less (Signed)
setle	(SF^OF)|ZF	Less or Equal (Signed)
seta	~CF&~ZF	Above (unsigned)
setb	CF	Below (unsigned)

SetX 指令

- 读取当前的条件码（或者某些条件码的组合），并存入目的“字节”寄存器
  - 余下的三个字节不会被修改
  - 通常使用“ movzbl”指令对目的寄存器进行“0”扩展


```
int gt (int x, int y)
{
  return x > y;
}
```

Body

```
movl 12(%ebp),%eax    # eax = y
cmpl %eax,8(%ebp)     # Compare x : y
setg %al              # al = x > y
movzbl %al,%eax       # Zero rest of %eax
(movsbl)
```

x86-64下读取条件码

SetX 指令

- 读取当前的条件码（或者某些条件码的组合），并存入目的“字节”寄存器
  - 余下的七个字节不会被修改

```
int gt (long x, long y)
{
  return x > y;
}
```

```
long lgt (long x, long y)
{
  return x > y;
}
```

- x86-64 下的函数参数
  - x in %rdi
  - y in %rsi

“64-bit operands generate a 64-bit result in the destination general-purpose register.
32-bit operands generate a 32-bit result, zero-extended to a 64-bit result in the destination general-purpose register.“

摘自“Intel® 64 and IA-32 Architectures Software Developer’s Manual Volume 1: Basic Architecture“

这两个过程的汇编代码主体是一样的！

```
xorl %eax, %eax      # eax = 0
cmpq %rsi, %rdi      # Compare x : y
setg %al             # al = x > y
```

Δ微体系结构背景*

“ 32-bit operands generate a 32-bit result, zero-extended to a 64-bit result in the destination general-purpose register.”

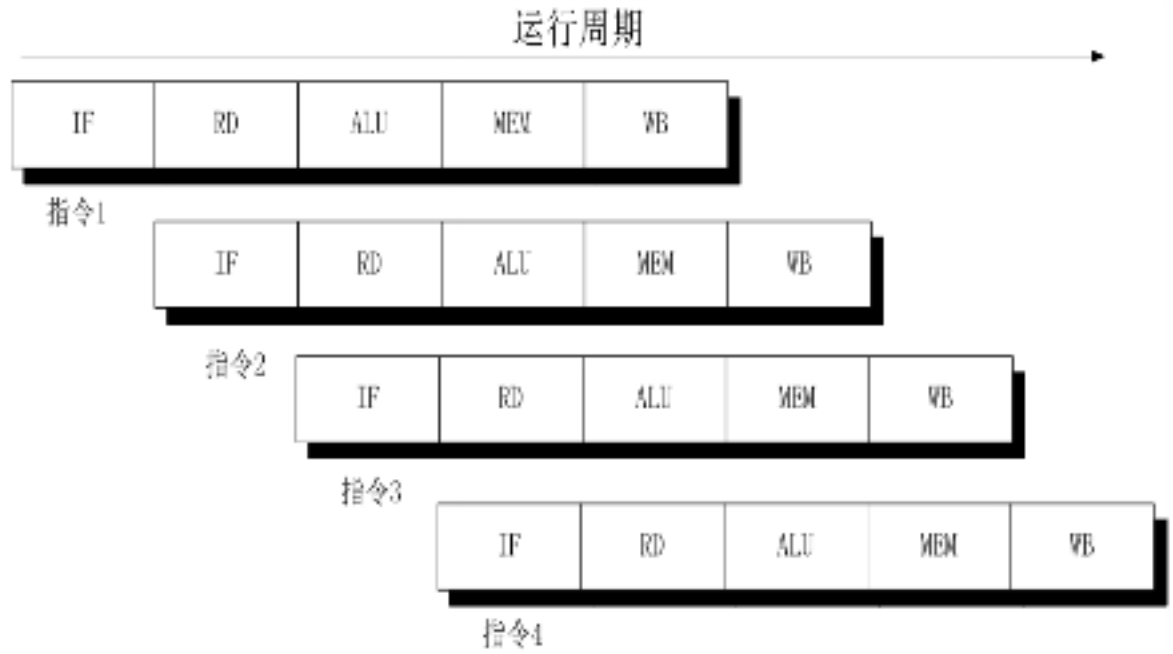
为什么？

- 部分原因来自于微体系结构（即处理器）内部实现的效率方面的考虑，目的是为了消除“部分数据依赖”

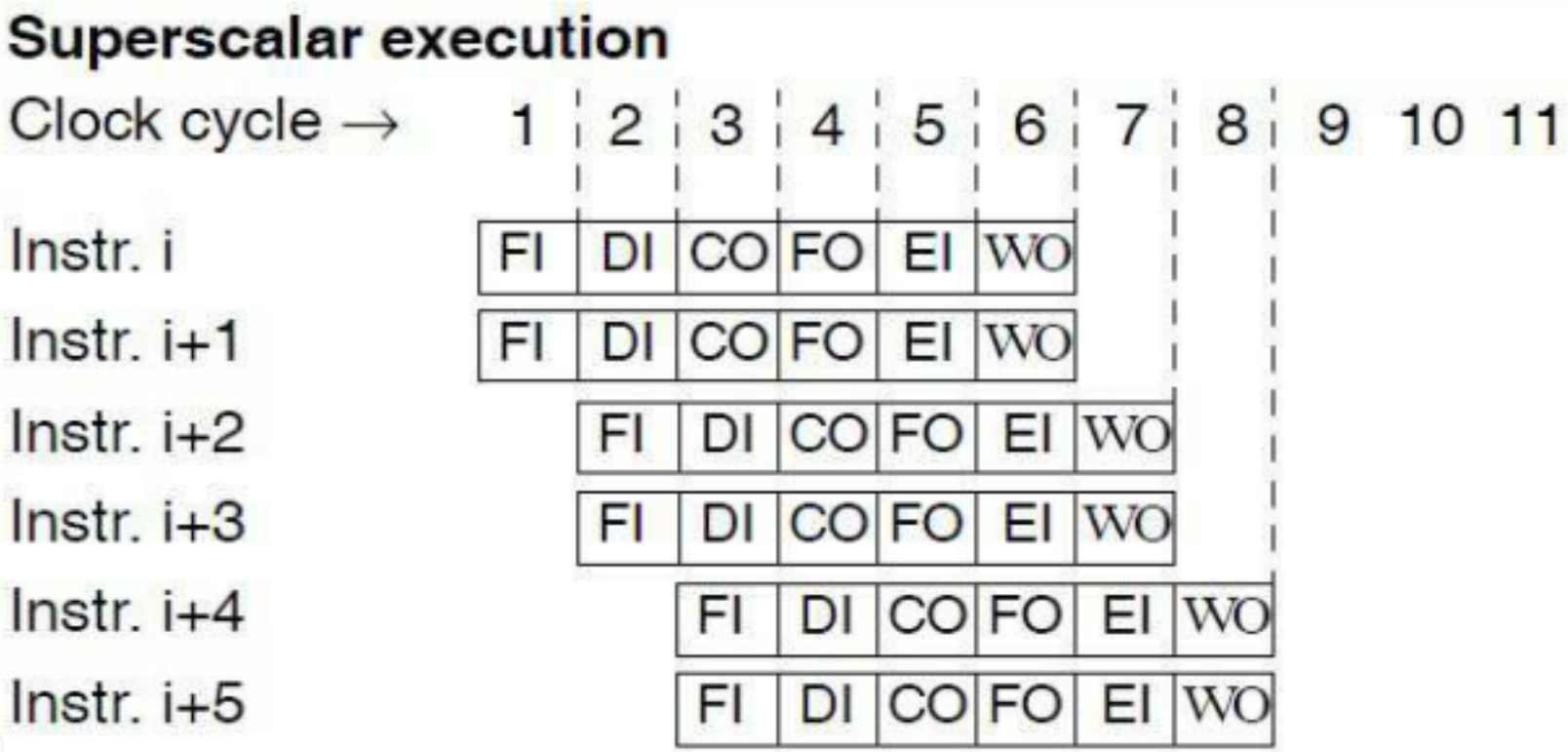
处理器流水线的概念

五级流水线（仅为示例，非x86处理器流水线）
- Instruction Fetch (IF)
- Read Registers (RD)
- Arithmetic Operation (ALU)
- Memory Access (MEM)
- Write Back (WB)

当前面指令产生的结果作为后续指令的操作数时，会引起“数据相关”(Data dependency)。



现代的通用处理器 支持深度流水线以及多发射结构，如 Pentium 4： >= 20 stages, up to 126 instructions on-fly



数据相关会导致指令执行效率降低（因为必须等待前面的结果出来），且流水线越深，影响越大。

•例子(**Partial Register Stall** is a problem that occurs when you write to part of a 32 bit register and later read from the whole register or a bigger part of it.)

```
addw %bx, %ax
movl %eax, %ecx
```

跳转指令

jX 指令

- 依赖当前的条件码选择下一条执行语句（是否顺序执行）

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Negative
jns	~SF	Nonnegative
jg	~(SF^OF)&~ZF	Greater (Signed)
jge	~(SF^OF)	Greater or Equal (Signed)
jl	(SF^OF)	Less (Signed)
jle	(SF^OF)|ZF	Less or Equal (Signed)
ja	~CF&~ZF	Above (unsigned)
jb	CF	Below (unsigned)

条件跳转指令实例

```
int absdiff(
    int x, int y)
{
    int result;
    if (x > y) {
        result = x-y;
    } else {
        result = y-x;
    }
    return result;
}
```

```
absdiff:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L7
    subl    %eax, %edx
    movl    %edx, %eax
.L8:
    leave
    ret
.L7:
    subl    %edx, %eax
    jmp     .L8
```

Set Up

Body1

Finish

Body2

```
int goto_ad(int x, int y)
{
   int result;
   if (x<=y) goto Else;
   result = x-y;
Exit:
   return result;
Else:
   result = y-x;
   goto Exit;
}
```

- 将原始的C代码变形为“goto”模式，使之接近编译出来的机器语言风格

Body1

```
   # x in %edx, y in %eax
   cmpl    %eax, %edx      # Compare x:y
   jle     .L7             # <= Goto Else
   subl    %eax, %edx      # x-= y
   movl    %edx, %eax      # result = x
.L8:  # Exit:
```

Body2

```
.L7:  # Else:
   subl    %edx, %eax      # result = y-x
   jmp     .L8             # Goto Exit
```

C语言:条件表达式

```
val = Test ? Then-Expr : Else-Expr;
```

```
val = x>y ? x-y : y-x;
```

Goto Version

```
   nt = !Test;
   if (nt) goto Else;
   val = Then-Expr;
Done:
   . . .
Else:
   val = Else-Expr;
   goto Done;
```

条件表达式的执行顺序：

- 先求解表达式Test，若为非0（真）则求解表达式*Then-Expr*，此时表达式2的值就作为整个表达式的值。
- 若Test的值为0（假），则求解*Else-Expr*，其值就是整个条件表达式的值。

x86-64下...

```
int absdiff(
    int x, int y)
{
    int result;
    if (x > y) {
        result = x-y;
    } else {
        result = y-x;
    }
    return result;
}
```

```
absdiff: # x in %edi, y in %esi
  movl   %edi, %eax  # v   = x
  movl   %esi, %edx  # ve  = y
  subl   %esi, %eax  # v  -= y
  subl   %edi, %edx  # ve -= x
  cmpl   %esi, %edi  # x:y
  cmovle %edx, %eax  # v=ve if <=
  ret
```

较新的32位Gcc也可以编译出类似代码 (-march=i686)

- 条件传送指令
  - cmov*C*　src, dest
  - 如果条件*C*成立，将数据从src 传送至dest
  - 从执行角度来看，比一般的条件跳转指令的效率高
    - » 因为其控制流可预测（即条件*C是已知的*）

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

movl 8(%ebp), %ecx
movl 12(%ebp), %edx
movl %ecx, %ebx
subl %edx, %ebx
movl %edx, %eax
subl %ecx, %eax
cmpl %edx, %ecx
cmovg %ebx, %eax

popl %ebx
popl %ebp
ret
```

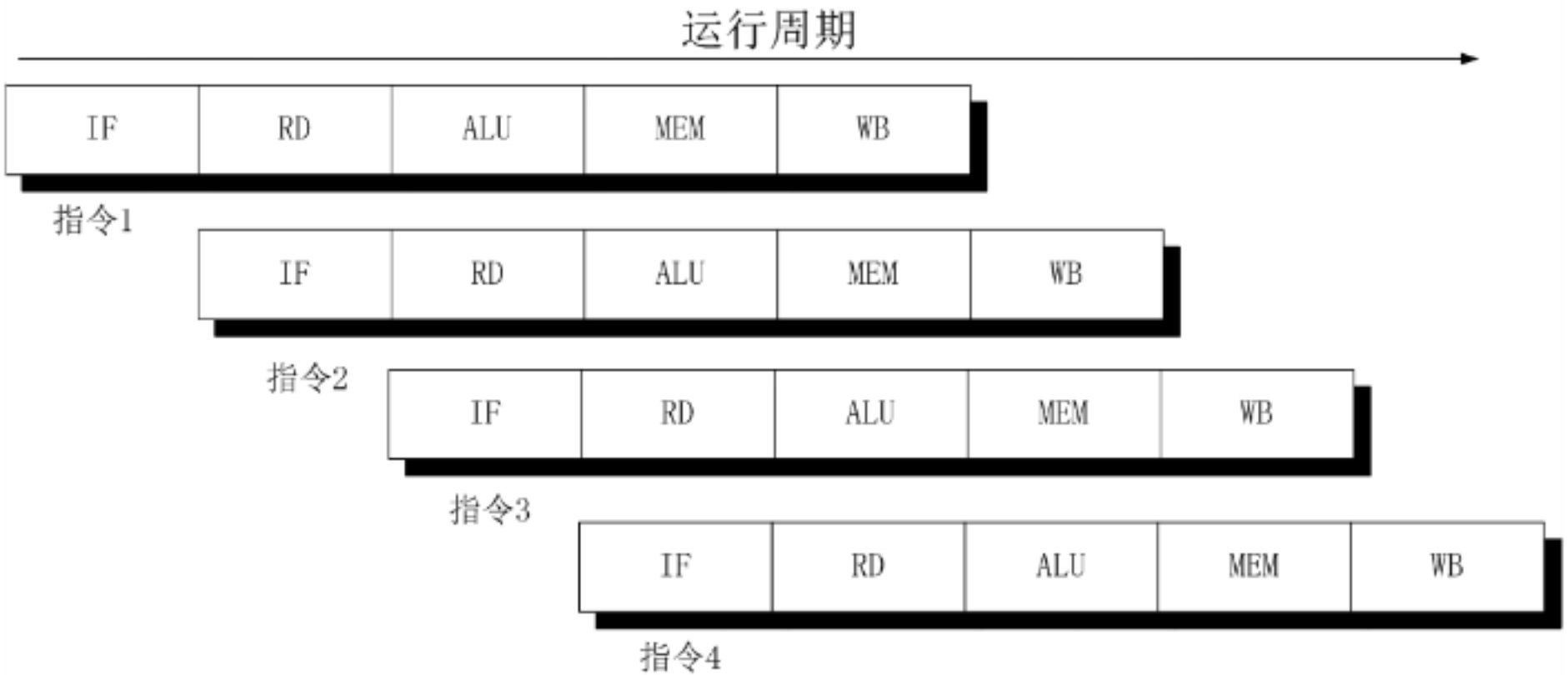
```
int absdiff(
    int x, int y)
{
    int result;
    if (x > y) {
        result = x-y;
    } else {
        result = y-x;
    }
    return result;
}
```

x86-32下条件传送指令的实例

Δ微体系结构背景

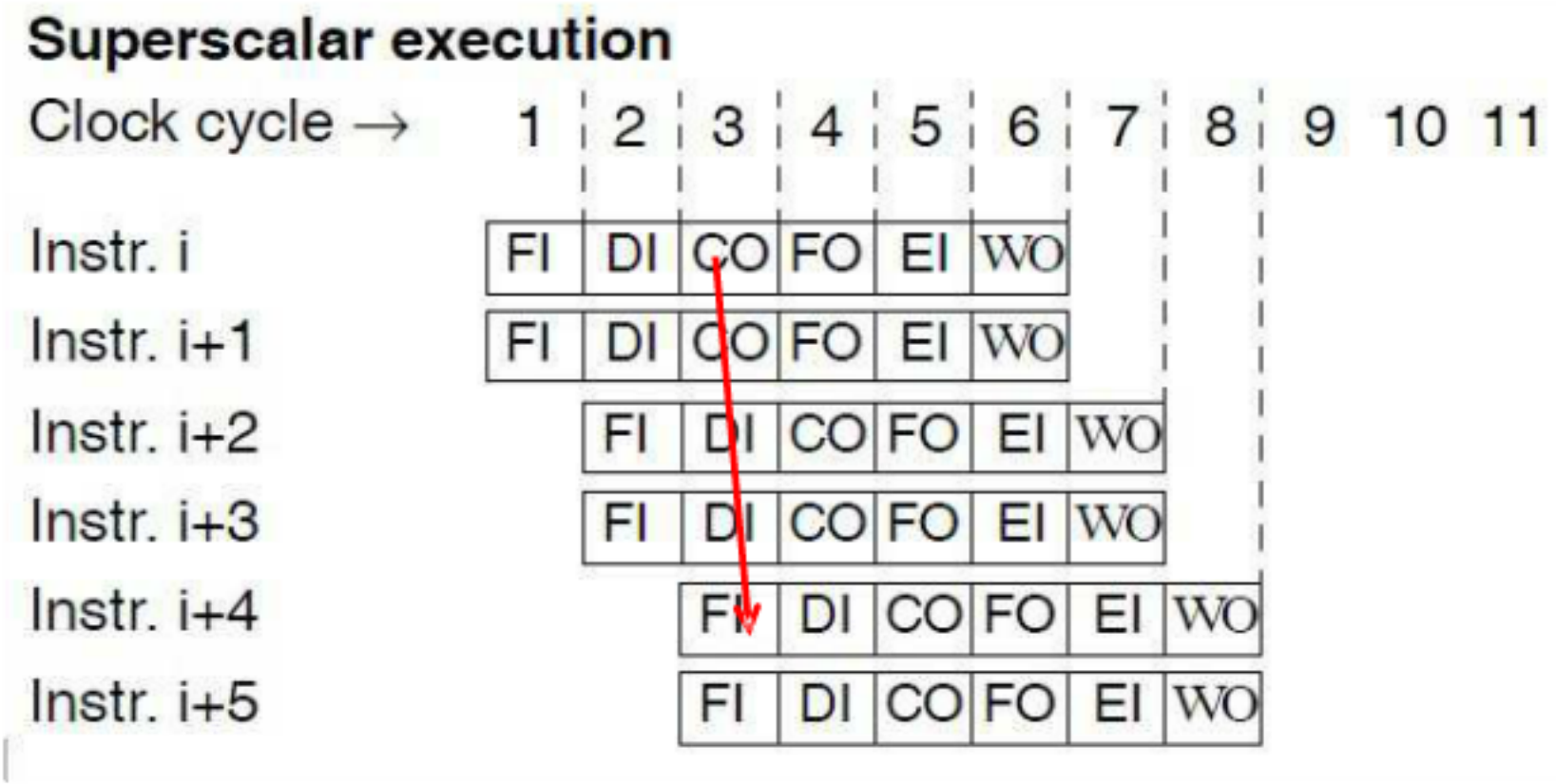
处理器流水线（五级流水示例）

- Instruction Fetch (IF)
- Read Registers (RD)
- Arithmetic Operation (ALU)
- Memory Access (MEM)
- Write Back (WB)



Δ微体系结构背景*

现代的通用处理器 支持深度流水线以及多发射结构，如
Pentium 4 : >= 20 stages, up to 126 instructions on-fly



条件跳转指令往往会引起一定的性能损失，因此需要尽量消除

条件转移指令的局限性

```
val   = Then-Expr;
vale  = Else-Expr;
val   = vale if !Test;
```

```
int xgty  = 0, xltey = 0;

int absdiff_se(
    int x, int y)
{
    int result;
    if (x > y) {
        xgty++;  result = x-y;
    } else {
        xltey++; result = y-x;
    }
    return result;
}
```

不宜使用的场合:

- Then-Expr 或Else-Expr 表达式有“副作用”
- Then-Expr 或 Else-Expr 表达式的计算量较大

练习题

使用条件移动指令来完成以下功能。

```
int cread(int *xp) {
    return (xp ? *xp : 0);
}
```

是否可以用如下汇编代码段来完成？

```
    Invalid implementation of function cread
    xp in register %edx
1       movl    $0, %eax        Set 0 as return value
2       testl   %edx, %edx      Test xp
3       cmovne  (%edx), %eax    if !0, dereference xp to get return value
```

```
int cread_alt(int *xp) {
    int t = 0;
    return *(xp ? xp : &t);
}
```

```
_cread_alt:
    ...
    movl  $0, -4(%ebp)      # t = 0
    movl  8(%ebp), %eax     #%eax = xp
    leal  -4(%ebp), %edx
    testl %eax, %eax
    cmove %edx, %eax
    movl (%eax), %eax
```

如何实现循环（Loops）

- 所有的循环模式（while，do-while，for）都转换为“do-while”形式
  - 再转换为汇编形式
- 历史上gcc采用过多种转换模式，经历了“否定之否定”的过程

“Do-While” 循环实例

原始的C代码

```
int fact_do(int x)
{
  int result = 1;
  do {
    result *= x;
    x = x-1;
  } while (x > 1);

  return result;
}
```

Goto Version

```
int fact_goto(int x)
{
  int result = 1;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
  return result;
}
```

- 编译器先转换为goto模式

Goto Version

```
int
fact_goto(int x)
{
  int result = 1;


loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;

  return result;
}
```

汇编

Registers

%edx	x
%eax	result

```
fact_goto:
  pushl %ebp          # Setup
  movl %esp,%ebp      # Setup
  movl $1,%eax        # eax = 1
  movl 8(%ebp),%edx   # edx = x

L11:
  imull %edx,%eax     # result *= x
  decl %edx           # x--
  cmpl $1,%edx        # Compare x : 1
  jg L11              # if > goto loop

  movl %ebp,%esp      # Finish
  popl %ebp           # Finish
  ret                 # Finish
```

“While” 循环-版本1

原始的C代码

```
int fact_while(int x)
{
  int result = 1;
  while (x > 1) {


    result *= x;
    x = x-1;
  };

  return result;
}
```

Goto Version-1

```
int fact_while_goto(int x)
{
  int result = 1;
loop:
  if (!(x > 1))
    goto done;
  result *= x;
  x = x-1;
  goto loop;
done:
  return result;
}
```

- 这个实例与上一个等价吗？

“While” 循环-版本2（do-while模式）

原始的C代码

```
int fact_while(int x)
{
  int result = 1;
  while (x > 1) {
    result *= x;
    x = x-1;
  };
  return result;
}
```

- 目前的GCC（4.0以后）使用的模式
- 内部循环与do-while 相同
- 在循环入口做额外的条件测试

Goto Version-2

```
int fact_while_goto2(int x)
{
  int result = 1;
  if (!(x > 1))
    goto done;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
done:
  return result;
}
```

“For” 循环

```
int result;
for (result = 1;
     p != 0;
     p = p>>1)
{
  if (p & 0x1)
    result *= x;
  x = x*x;
}
```

General Form

```
for (Init; Test; Update )
    Body
```

Init

```
result = 1
```

Test

```
p != 0
```

Update

```
p = p >> 1
```

Body

```
{
  if (p & 0x1)
    result *= x;
  x = x*x;
}
```

"For" → "While" → "Do-While"

For Version

```
for (Init; Test; Update )
    Body
```

While Version

```
Init;
while (Test ) {
    Body
    Update ;
}
```

Do-While Version

```
  Init;
  if (!Test)
    goto done;
  do {
    Body
    Update ;
  } while (Test)
done:
```

Goto Version

```
  Init;
  if (!Test)
    goto done;
loop:
  Body
  Update ;
  if (Test)
    goto loop;
done:
```

补充：

历史上gcc采用过多种转换模式，经历了“否定之否定”的过程

“While”循环-版本3 （jump-to-middle）

原始的C代码

```
int fact_while(int x)
{
  int result = 1;
  while (x > 1) {
    result *= x;
    x = x-1;
  };
  return result;
}
```

- 由jump-to- middle开始第一轮循环

Goto Version

```
int fact_while_goto3(int x)
{
  int result = 1;
  goto middle;
loop:
  result *= x;
  x = x-1;
middle:
  if (x > 1)
    goto loop;
  return result;
}
```

Jump-to-Middle 实例

```
int fact_while(int x)
{
  int result = 1;
  while (x > 1) {
    result *= x;
    x--;
  };
  return result;
}
```

- gcc 3.4.4
  - -O2

```
# x in %edx, result in %eax
  jmp   L34          #   goto Middle
L35:                 # Loop:
  imull %edx, %eax   #   result *= x
  decl  %edx         #   x--
L34:                 # Middle:
  cmpl  $1, %edx     #   x:1
  jg    L35          #   if >, goto Loop
```

"For" → "While" (Jump-to-Middle)

For Version

```
for (Init; Test; Update )
    Body
```

While Version

```
Init;
while (Test ) {
    Body
    Update ;
}
```

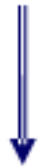
Goto Version

```
  Init;
  goto middle;
loop:
  Body
  Update ;
middle:
  if (Test)
    goto loop;
done:
```

Jump-to-Middle模式

C Code

```
while (Test)
  Body
```



Goto Version

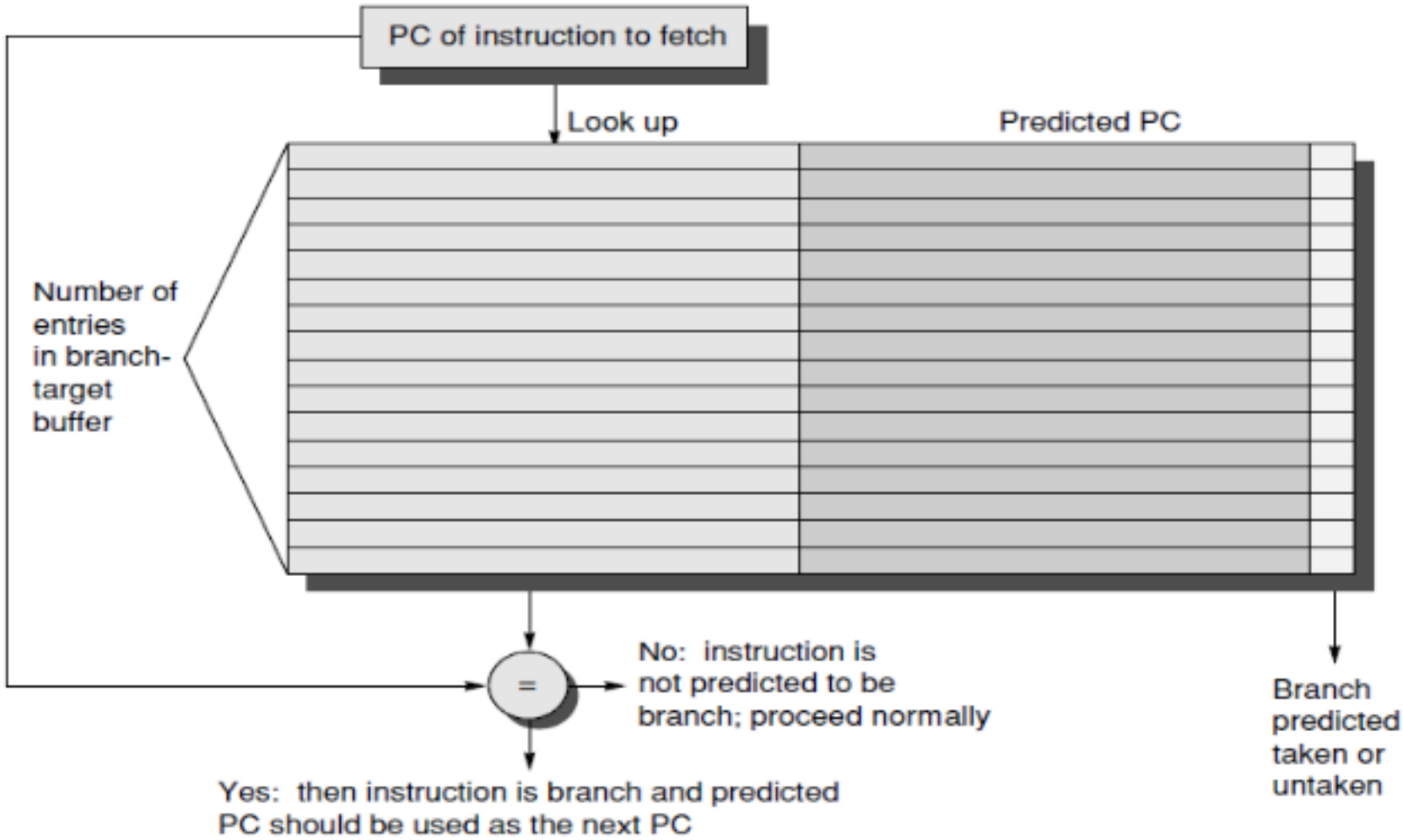
```
# x in %edx, result in %eax
  jmp   L34          #   goto Middle
L35:                 # Loop:
  imull %edx, %eax   #   result *= x
  decl  %edx         #   x--
L34:                 # Middle:
  cmpl  $1, %edx     #   x:1
  jg    L35          #   if >, goto Loop
```

特点：

- 避免了双重测试
  - 无条件跳转指令的处理器运行开销非常低（可以忽略）

Δ 微体系结构背景*

条件跳转指令往往会引起一定的性能损失，Branch Prediction技术被引入来进行优化。



Branch Prediction的表项数有限，且其依据跳转与否的历史信息来做预测。因此条件跳转指令越多（一般以指令地址来识别），跳转历史信息越碎片化，就越不利于提升预测精确度。

Branch Prediction继续发展，采用了循环预测器技术（US Patent 5909573），能够对loop进行专门的预测——即对于“循环入口”的预测基本为真。

Switch语句

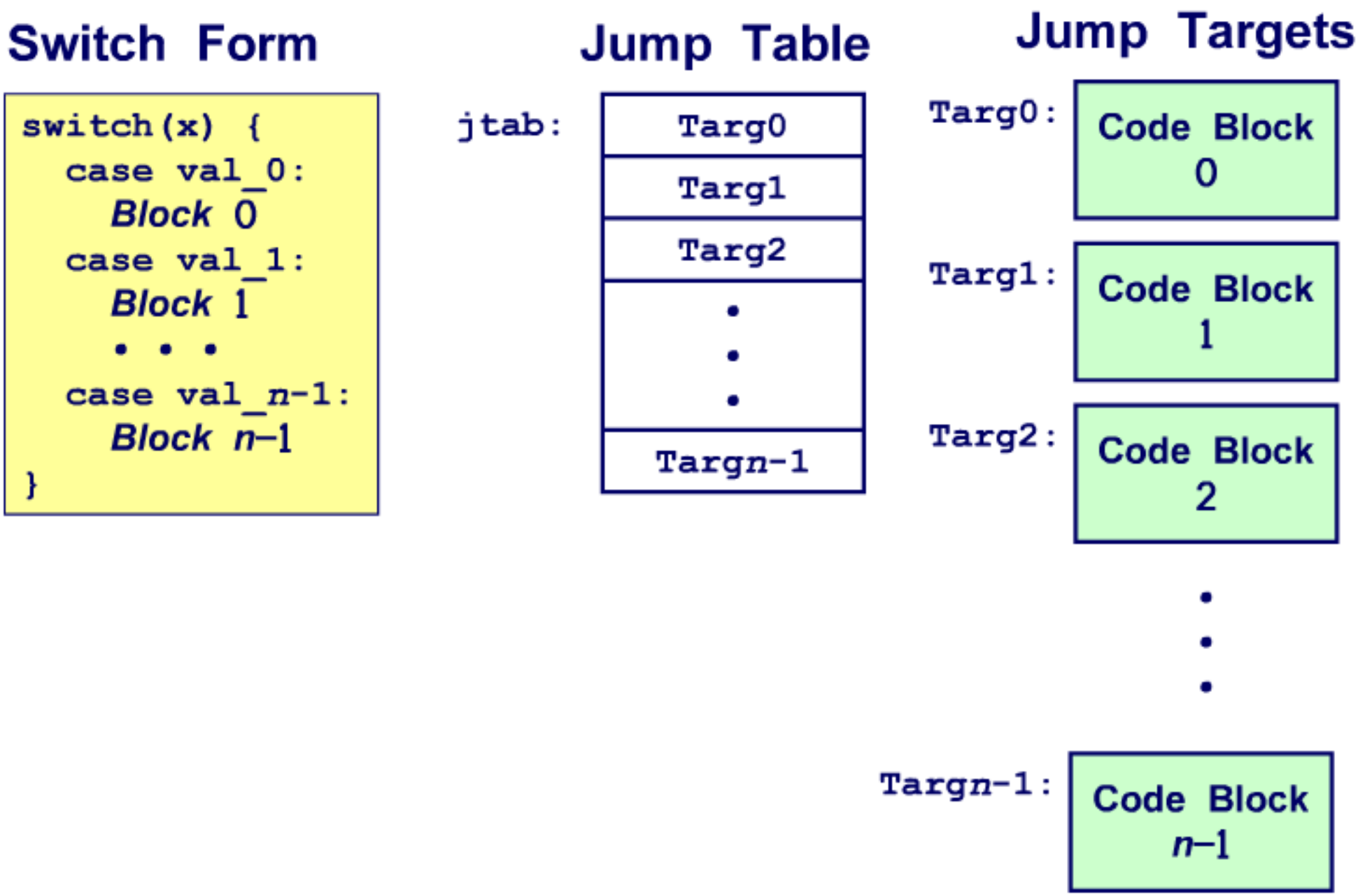
- 依据不同情况来采用不同的实现技术
  - 使用一组**if-then-else**语句来实现
  - 使用跳转表

```
long switch_eg
    (long x, long y, long z)
{
     long w = 1;
     switch(x) {
     case 1:
          w = y*z;
          break;
     case 2:
          w = y/z;
          /* Fall Through */
     case 3:
          w += z;
          break;
     case 5:
     case 6:
          w -= z;
          break;
     default:
          w = 2;
     }
     return w;
}
```

Switch 语句

- 多个case对应同一段处理语句
- “Fall through”
- Case值并不连续

跳转表


Switch Form
switch(x) {
case val_0:
Block 0
case val_1:
Block 1
• • •
case val_n-1:
Block n–1
}
Jump Table
jtab:
Targ0
Targ1
Targ2
•
•
•
Targn-1
Jump Targets
Targ0:
Code Block 0
Targ1:
Code Block 1
Targ2:
Code Block 2
•
•
•
Targn-1:
Code Block n–1

Switch 语句示例（x86-32）

```
long switch_eg
    (long x, long y, long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

Setup:

```
switch_eg:
   pushl %ebp              # Setup
   movl  %esp, %ebp        # Setup
   pushl %ebx              # Setup
   movl  $1, %ebx          # w = 1
   movl  8(%ebp), %edx     # edx = x
   movl  16(%ebp), %ecx    # ecx = z
   cmpl  $6, %edx          # x:6
   ja    .L61              # if > goto default
   jmp   *.L62(,%edx,4)    # goto JTab[x]
```

表结构

- 每个表项（及跳转地址）占**4** 字节
- 基地址是 **.L62**

无条件跳转指令

jmp .L61

- Jump target is denoted by label **.L61**

jmp *.L62(,%edx,4)

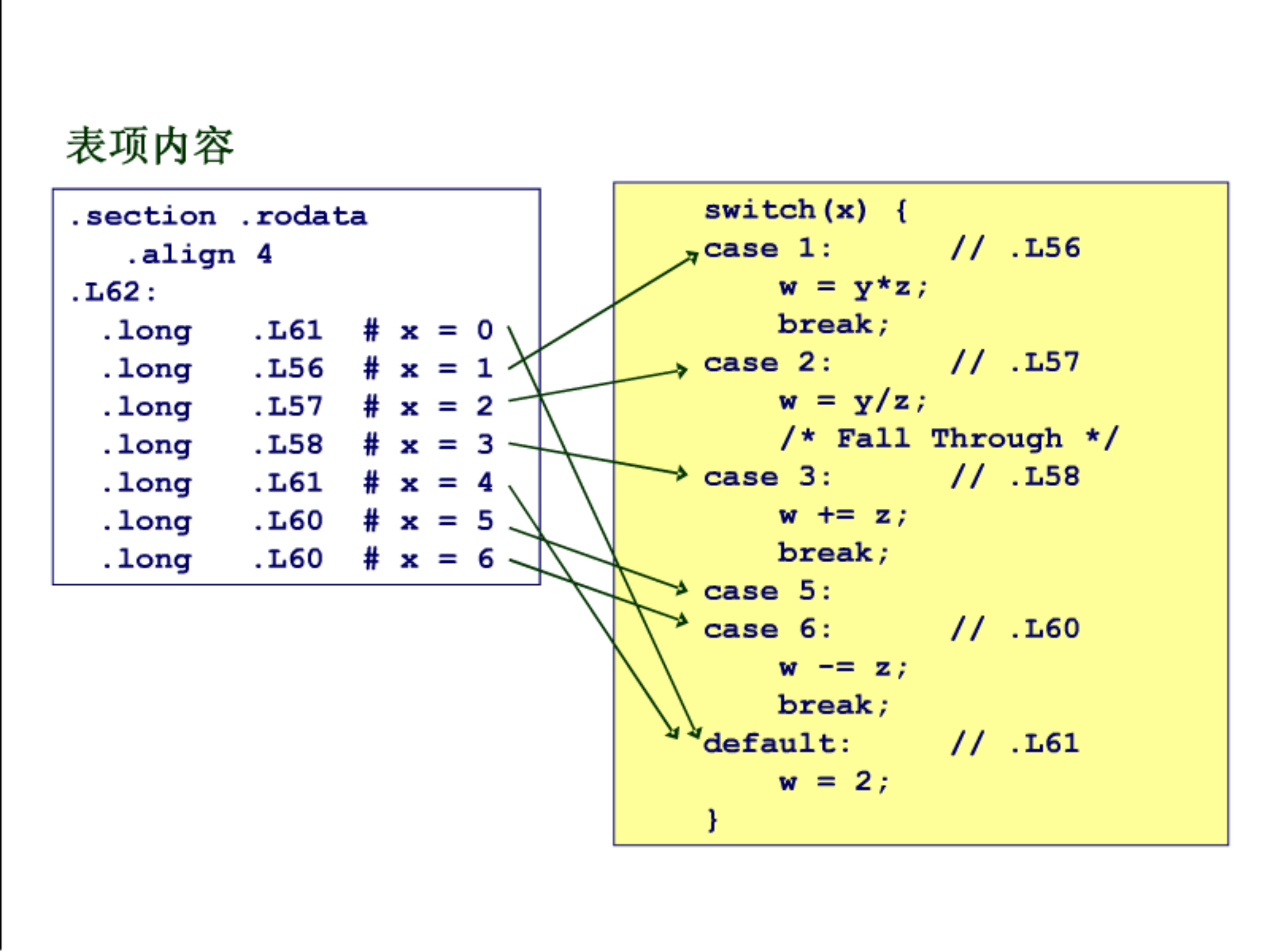
- Start of jump table denoted by label **.L62**
- Register **%edx** holds **x**
- Must scale by factor of **4** to get offset into table
- Fetch target from effective Address **.L61 + x*4**
  - Only for **0 ≤ x ≤ 6**

*表明这是一个间接跳转，即目标地址存于内存地址中

表项内容

```
.section .rodata
   .align 4
.L62:
  .long    .L61   # x = 0
  .long    .L56   # x = 1
  .long    .L57   # x = 2
  .long    .L58   # x = 3
  .long    .L61   # x = 4
  .long    .L60   # x = 5
  .long    .L60   # x = 6
```

```
switch(x) {
case 1:          // .L56
    w = y*z;
    break;
case 2:          // .L57
    w = y/z;
    /* Fall Through */
case 3:          // .L58
    w += z;
    break;
case 5:
case 6:          // .L60
    w -= z;
    break;
default:         // .L61
    w = 2;
}
```



```
switch(x) {
  . . .
case 2:        // .L57
    w = y/z;
    /* Fall Through */
case 3:        // .L58
    w += z;
    break;
  . . .
default:       // .L61
    w = 2;
}
```

```
.L61:  // Default case
  movl  $2, %ebx     # w = 2
  movl  %ebx, %eax   # Return w
  popl  %ebx
  leave
  ret
.L57:  // Case 2:
  movl  12(%ebp), %eax  # y
  cltd                  # Div prep
  idivl %ecx            # y/z
  movl  %eax, %ebx # w = y/z
# Fall through
.L58:  // Case 3:
  addl  %ecx, %ebx # w+= z
  movl  %ebx, %eax # Return w
  popl  %ebx
  leave
  ret
```

```
switch(x) {
case 1:          // .L56
    w = y*z;
    break;
  . . .
case 5:
case 6:          // .L60
    w -= z;
    break;
  . . .
}
```

```
.L60: // Cases 5&6:
  subl  %ecx, %ebx  # w -= z
  movl  %ebx, %eax  # Return w
  popl  %ebx
  leave
  ret
.L56: // Case 1:
  movl  12(%ebp), %ebx # w = y
  imull %ecx, %ebx     # w*= z
  movl  %ebx, %eax  # Return w
  popl  %ebx
  leave
  ret
```

x86-64 下的Switch语句

- 基本与32位版本一样
  - 地址长度64位

Jump Table

```
.section .rodata
   .align 8
.L62:
  .quad    .L55   # x = 0
  .quad    .L50   # x = 1
  .quad    .L51   # x = 2
  .quad    .L52   # x = 3
  .quad    .L55   # x = 4
  .quad    .L54   # x = 5
  .quad    .L54   # x = 6
```

```
    switch(x) {
    case 1:        // .L50
        w = y*z;
        break;
      . . .
    }
```

```
.L50: // Case 1:
   movq    %rsi, %r8   # w = y
   imulq   %rdx, %r8   # w *= z
   movq    %r8, %rax   # Return w
   ret
```

Switch语句实例（case值很稀疏）

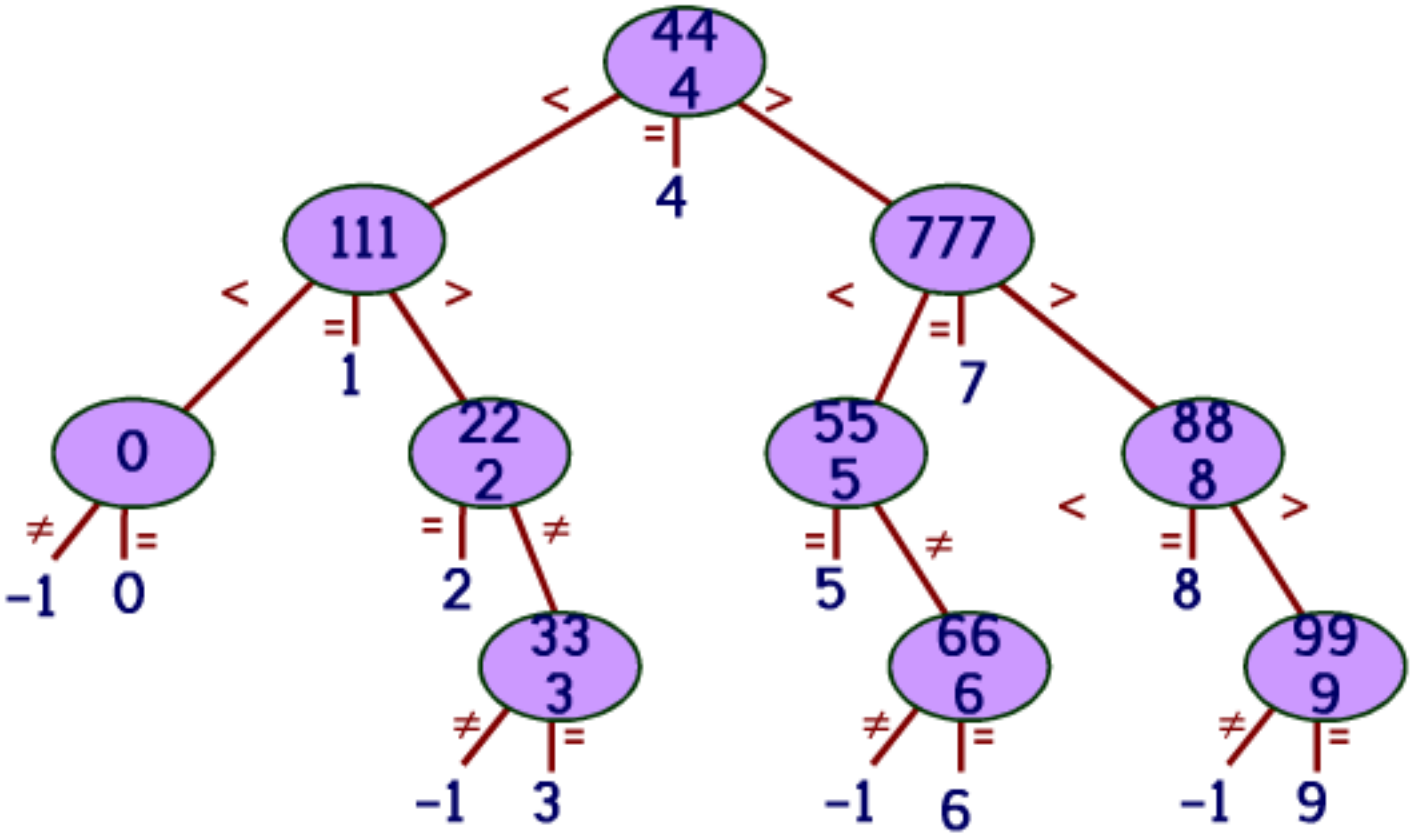
```
/* Return x/111 if x is multiple
   && <= 999.  -1 otherwise */
int div111(int x)
{
  switch(x) {
  case   0: return 0;
  case 111: return 1;
  case 222: return 2;
  case 333: return 3;
  case 444: return 4;
  case 555: return 5;
  case 666: return 6;
  case 777: return 7;
  case 888: return 8;
  case 999: return 9;
  default: return -1;
  }
}
```

- 因此不适合使用跳转表
  - 为什么？
- 如何高效的转换为一系列的if-then-else 语句？

X86-32下的实例

```
    movl 8(%ebp),%eax  # get x
    cmpl $444,%eax     # x:444
    je L8
    jg L16
    cmpl $111,%eax     # x:111
    je L5
    jg L17
    testl %eax,%eax    # x:0
    je L4
    jmp L14

    . . .
```

```
    . . .
L5:
    movl $1,%eax
    jmp L19
L6:
    movl $2,%eax
    jmp L19
L7:
    movl $3,%eax
    jmp L19
L8:
    movl $4,%eax
    jmp L19
    . . .
```



- 以二叉树的结构组织，提升性能

小结

- 条件码
  - 设置
  - 读取
  - 条件跳转指令
  - 条件传送指令

- 程序控制流
  - If-then-else
  - 循环结构
    - Do-while
    - While
    - for
  - switch语句

80X86汇编语言与C语言-3

栈与过程

x86-32

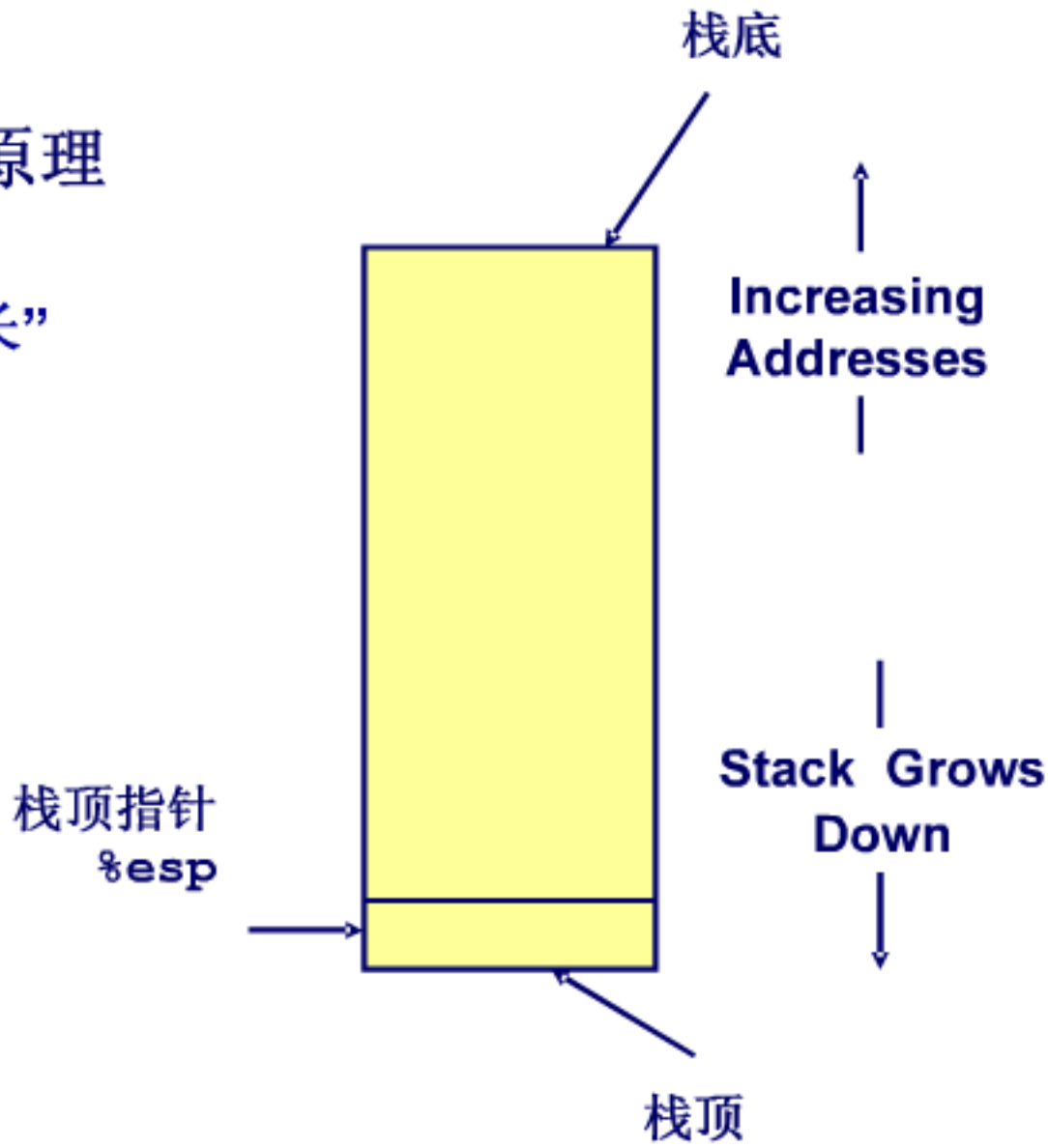
- 栈的工作规律
- 过程调用中的寄存器使用惯例
- 栈也用于存储局部变量

x86-64

- （部分）过程参数通过寄存器传递
- 尽量减少对于栈的使用

x86-32 的程序栈

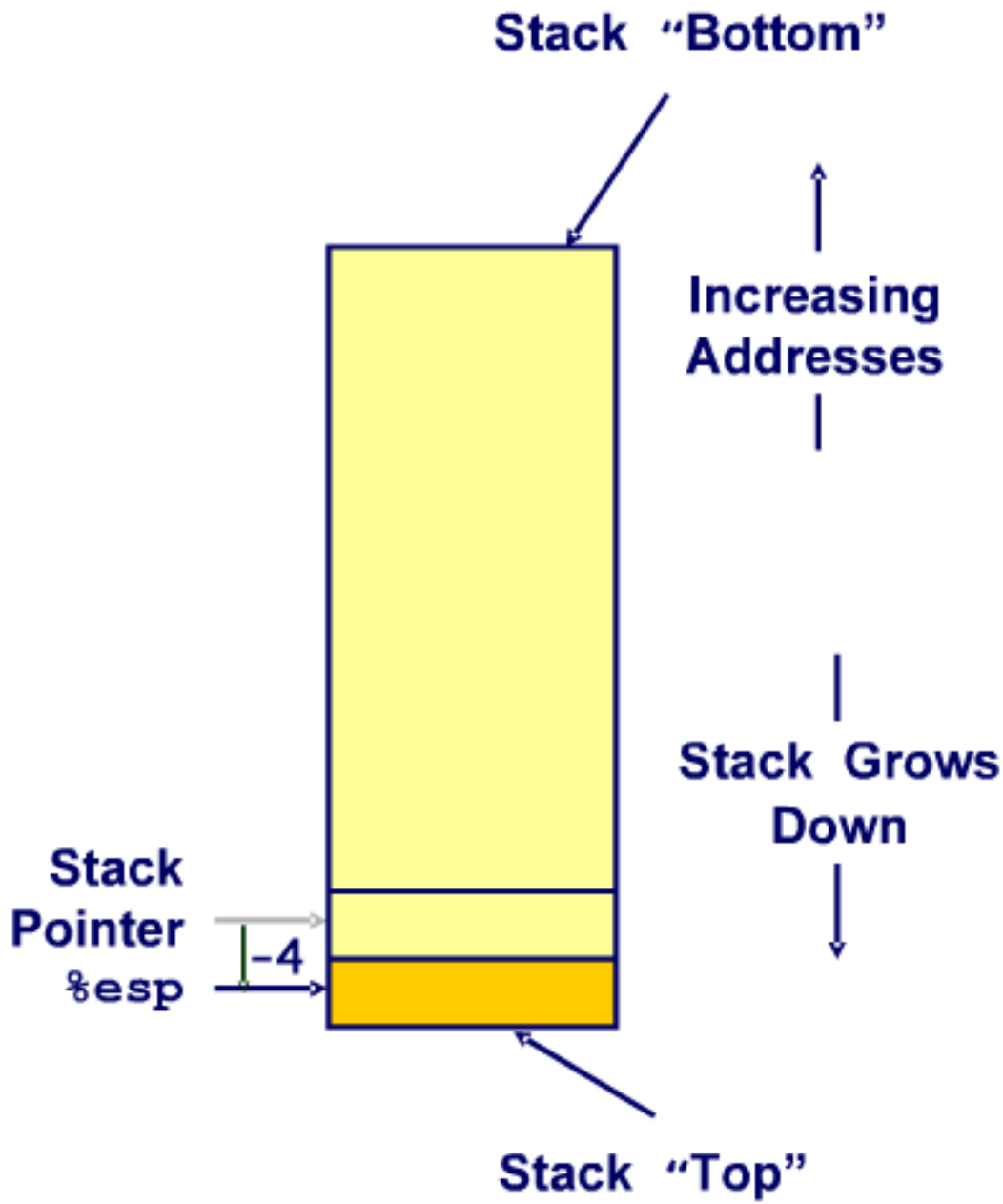
- 符合"栈(stack)"工作原理的一块内存区域
  - 从高地址向低地址"增长"
- %esp存储栈顶地址



压栈操作

`pushl` *Src*

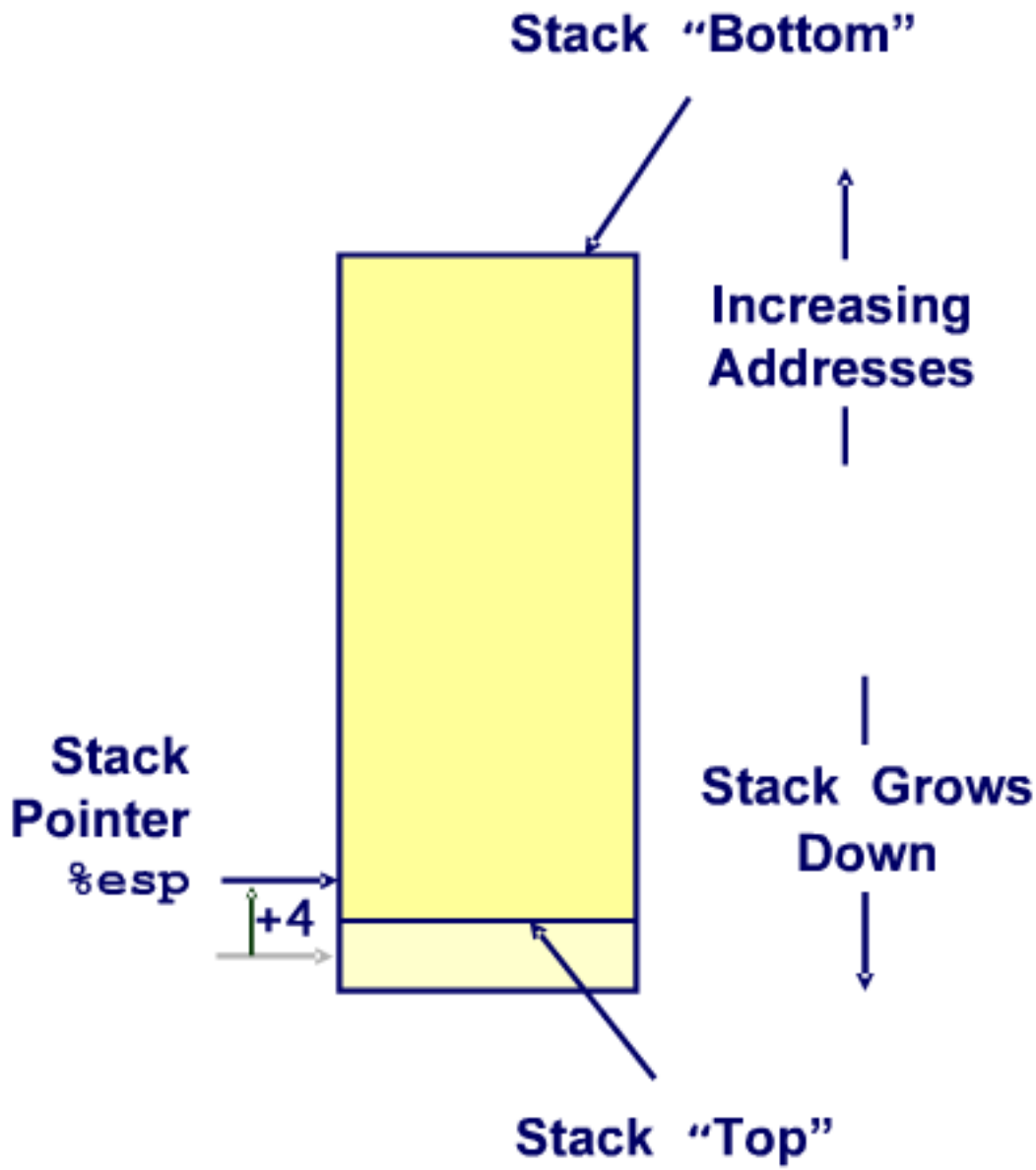
- 从*Src*取得操作数
- `%esp = %esp - 4`
- 写入栈顶地址（`%esp`）



出栈操作

`popl` ***Dest***

- 读取栈顶数据(`%esp`)
- `%esp = %esp + 4`
- 写入***Dest***



过程调用

- 利用栈支持过程调用与返回

过程调用指令:

`call` *`label`*　　将返回地址压入栈，跳转至*`label`*

返回地址

- Call指令的下一条指令地址
- 汇编实例

```
804854e:  e8 3d 06 00 00    call   8048b90 <main>
8048553:  50                pushl  %eax
```

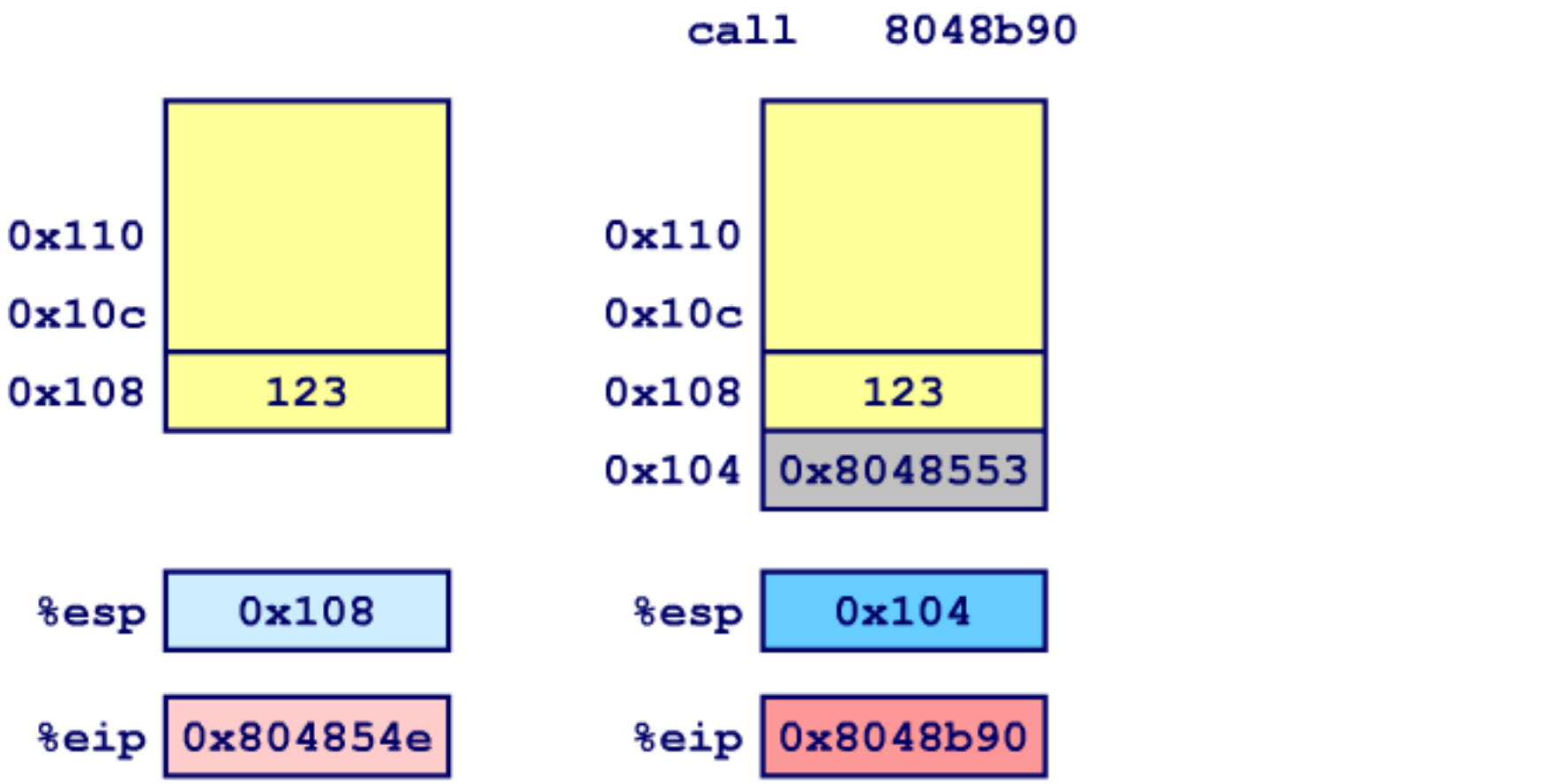
  - Return address = 0x8048553

过程返回指令:

- `ret`　　跳转至栈顶的返回地址

过程调用实例

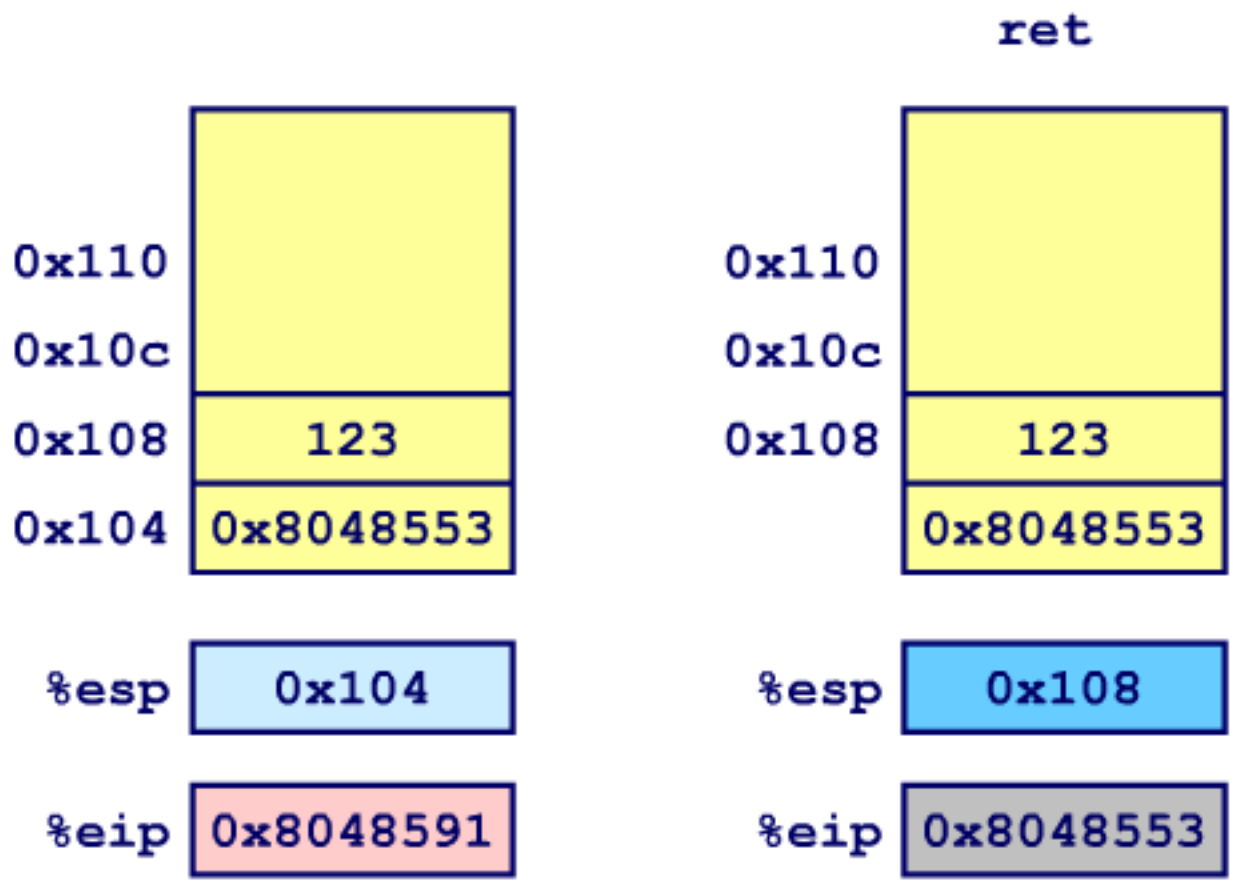
```
804854e:   e8 3d 06 00 00        call   8048b90 <main>
8048553:   50                    pushl  %eax
```



%eip 是指令存储器（program counter）

过程返回实例

```
8048591:   c3                    ret
```



%eip 是指令存储器（program counter）

基于栈的编程语言

支持递归

- e.g., C, Pascal, Java
- 代码是可重入的（*Reentrant*）
  - 同时有同一个过程的多个实例在运行
- 因此需要有一块区域来存储每个过程实例的数据
  - 参数
  - 局部变量
  - 返回地址

栈的工作规律

- 每个过程实例的运行时间是有限的，即栈的有效时间有限
  - From when called to when return
- 被调用者先于调用者返回（一般情况下）

每个过程实例在栈中维护一个栈帧（stack frame）

连续过程调用示例

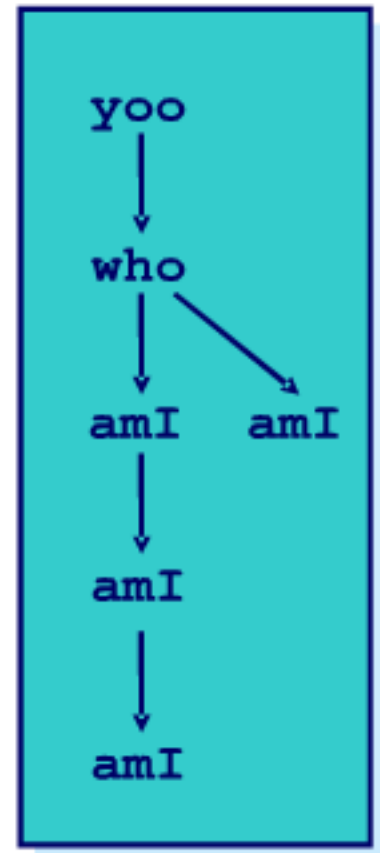
Code Structure

```
yoo(…)
{
    •
    •
    who();
    •
    •
}
```

```
who(…)
{
    • • •
    amI();
    • • •
    amI();
    • • •
}
```

```
amI(…)
{
    •
    •
    amI();
    •
    •
}
```

Call Chain

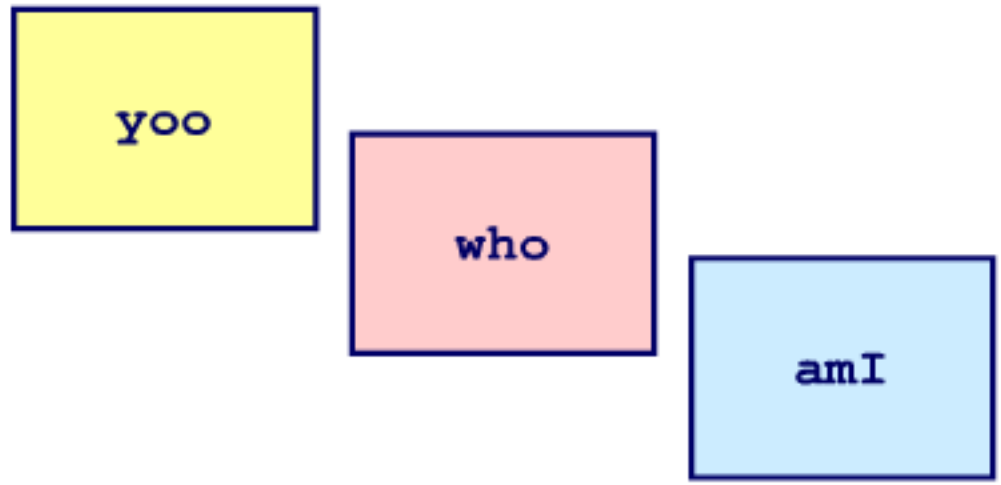


- 过程amI递归了

栈帧(stack frame)

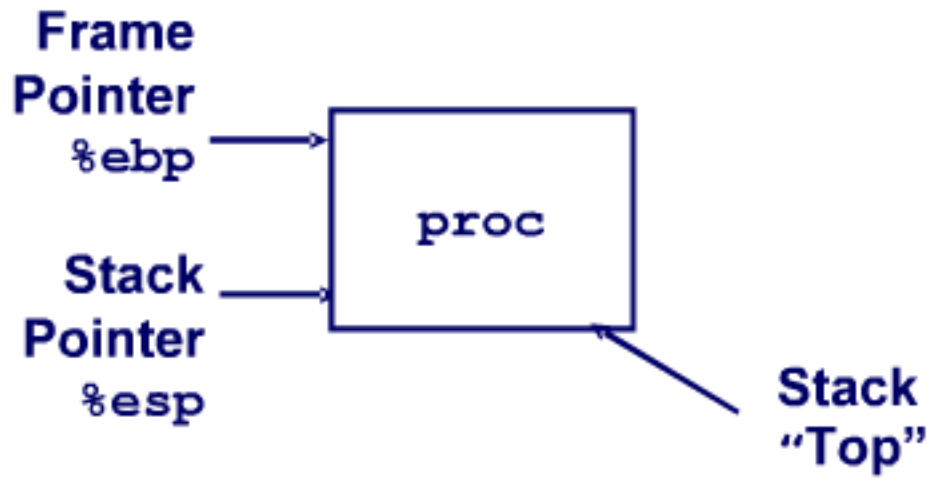
存储的内容

- 局部变量
- 返回地址
- 临时空间



栈帧的分配与释放

- 进入过程后先“分配”栈帧空间
  - “Set-up” code
- 过程返回时”释放”
  - “Finish” code



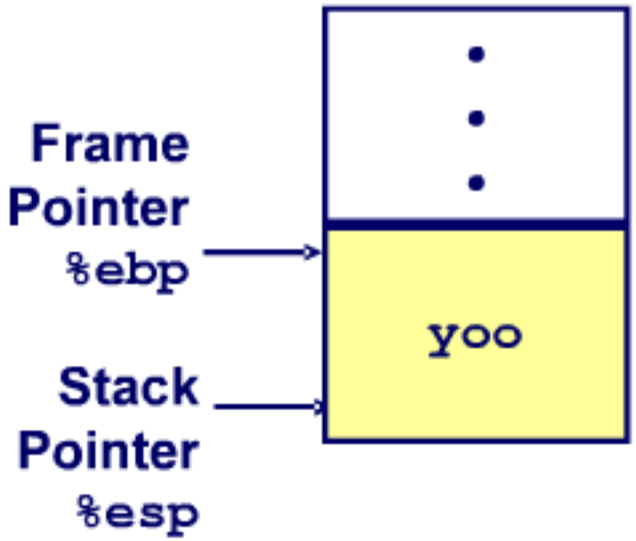
- 寄存器%ebp 指向当前栈帧的起始地址

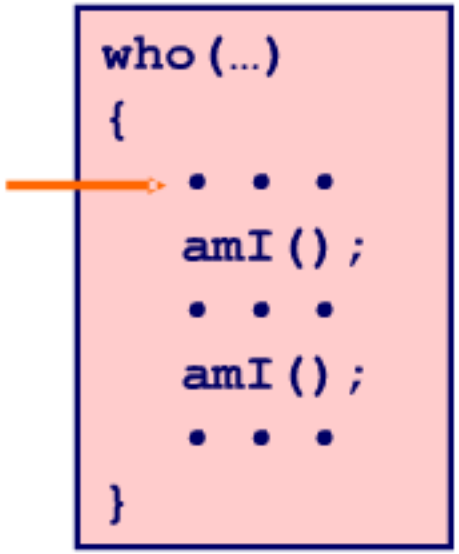
过程调用时栈的变化

```
yoo(...)
{
    •
    •
    who();
    •
    •
}
```

Call Chain

yoo

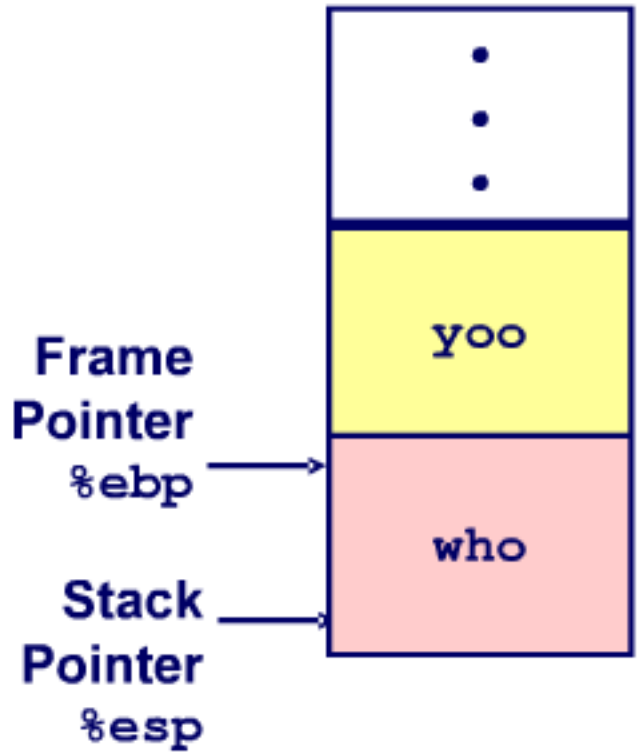


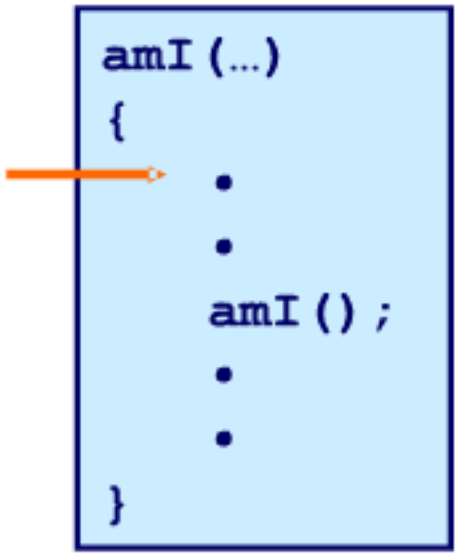


Call Chain

yoo

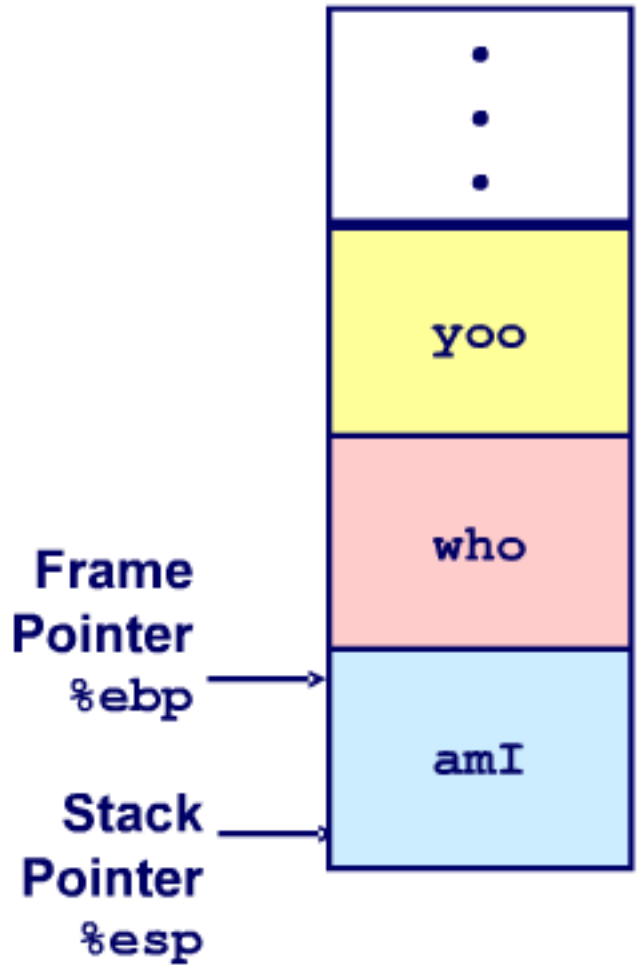
who

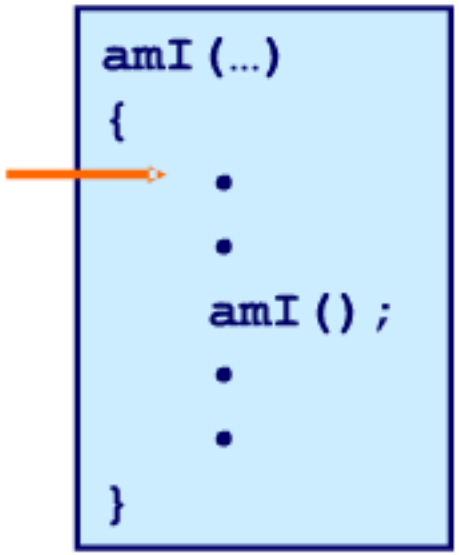




Call Chain

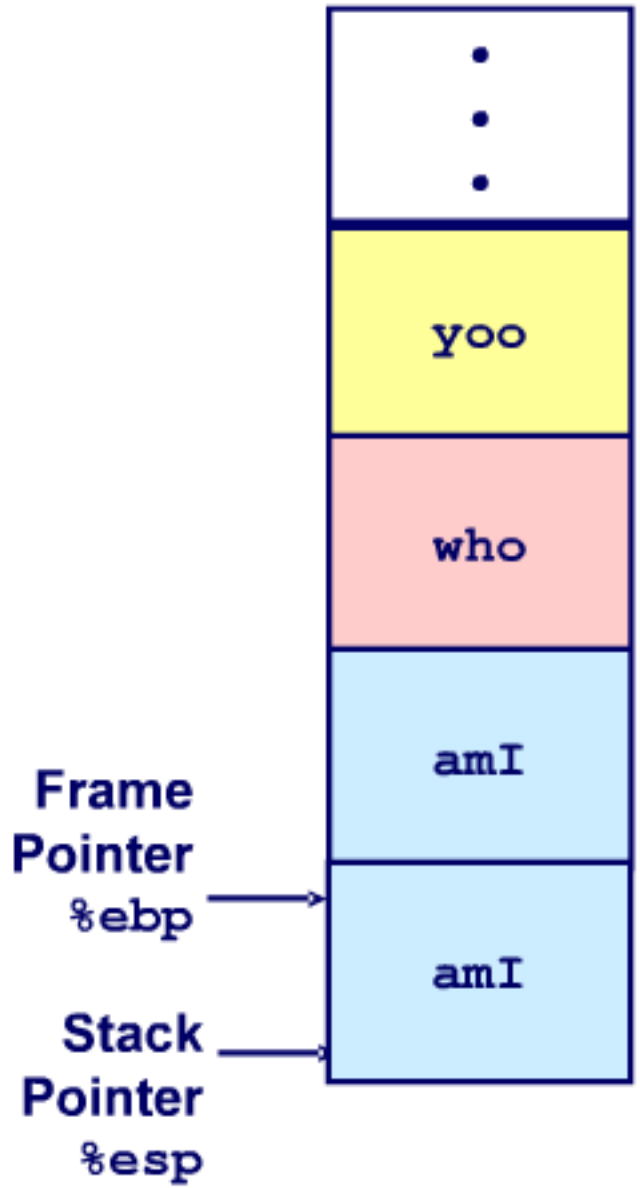
yoo
who
amI

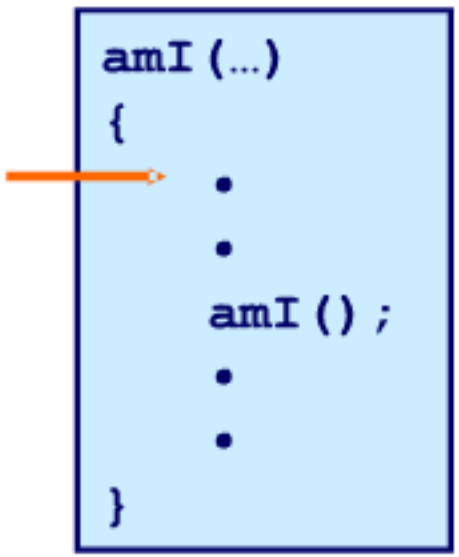




Call Chain

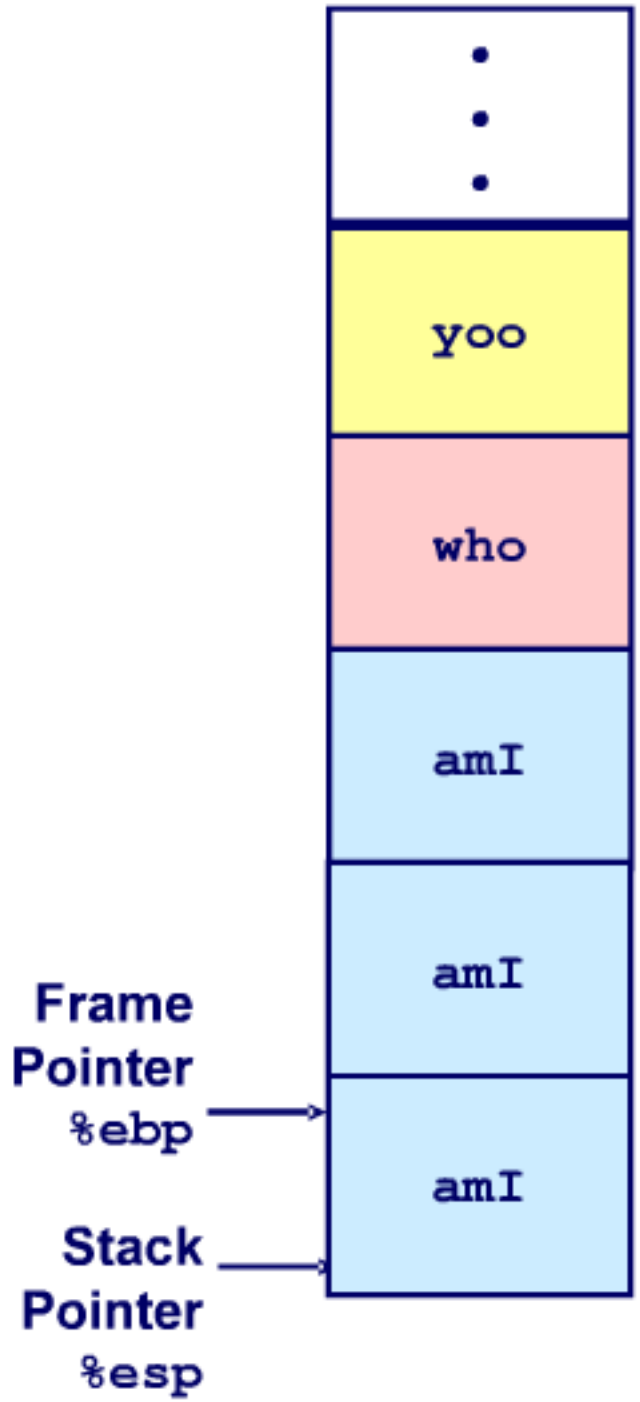
yoo
who
amI
amI

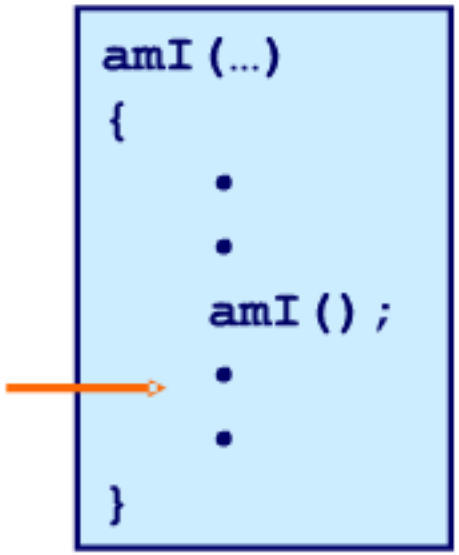




Call Chain

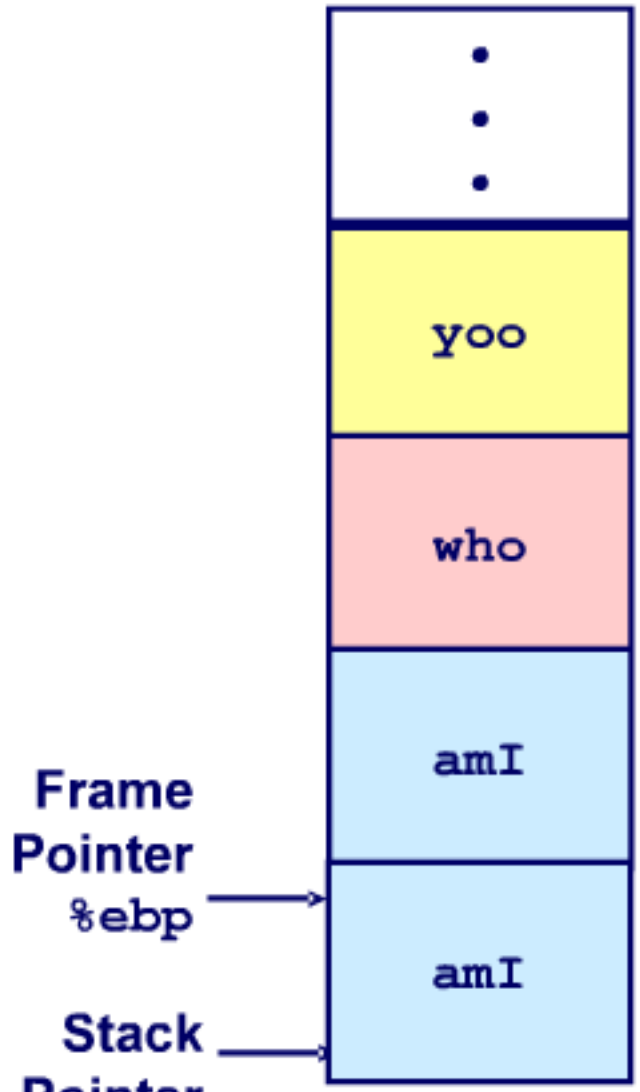
yoo
who
amI
amI
amI

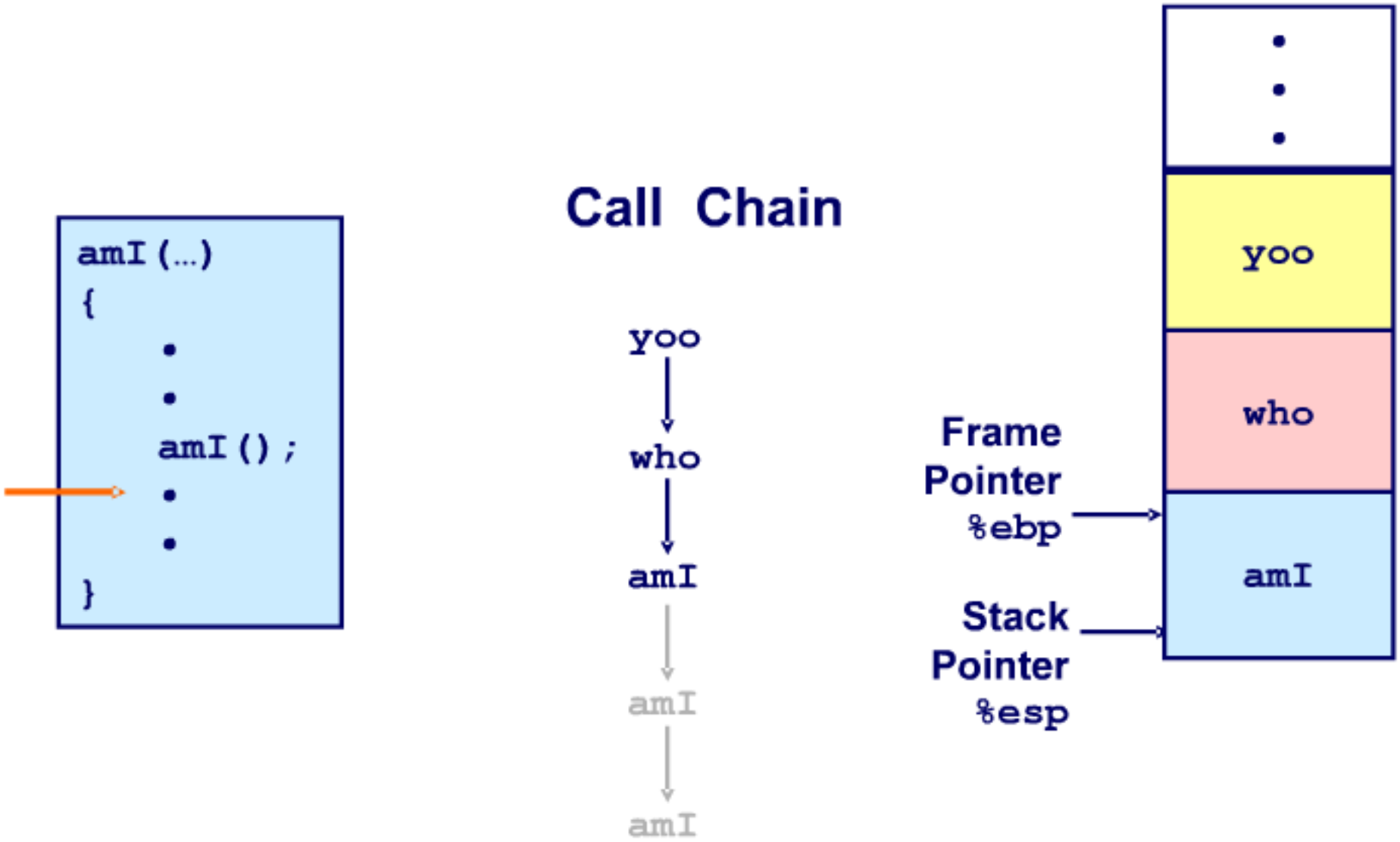


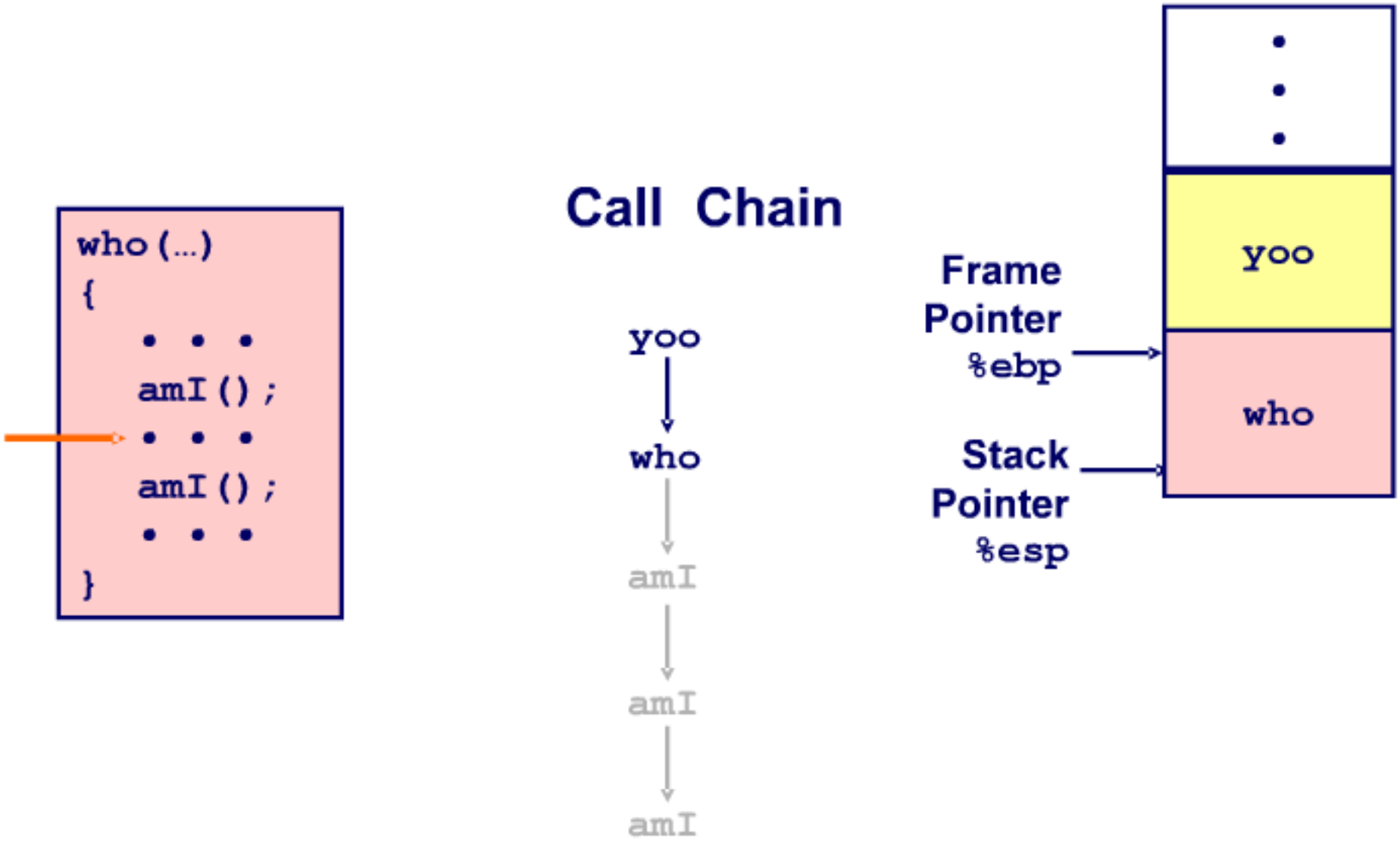


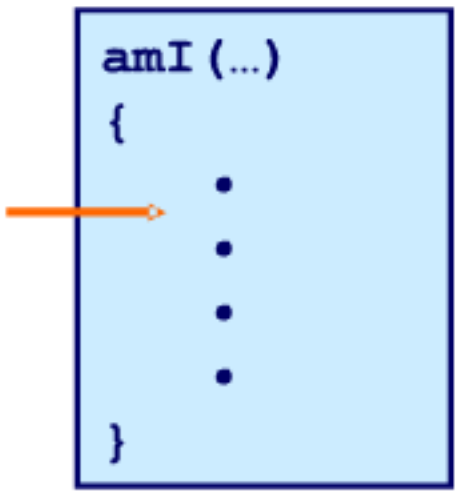
Call Chain

yoo
who
amI
amI
amI



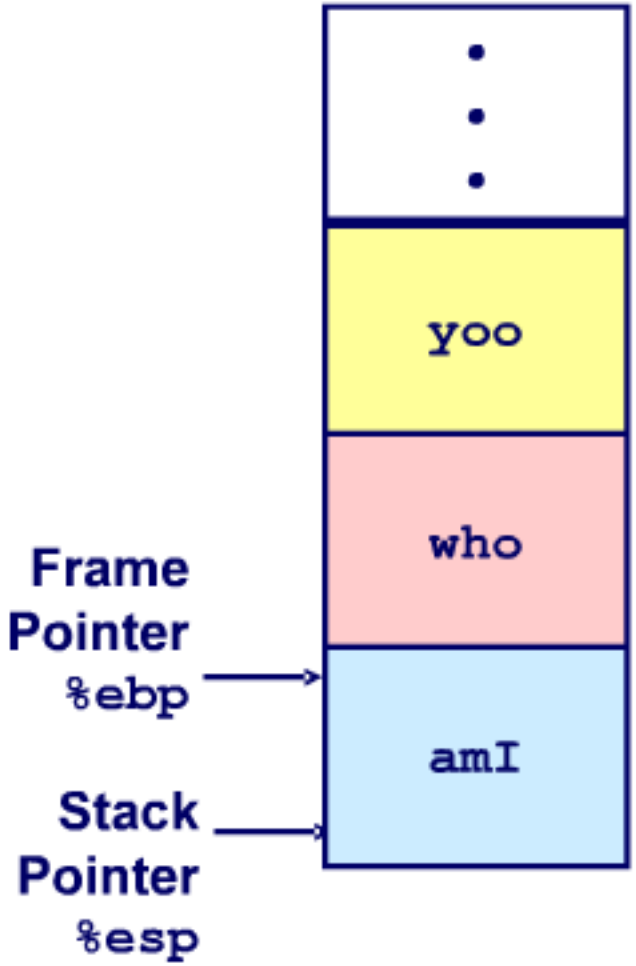

amI(...)
{
amI();
}
Call Chain
yoo
who
amI
amI
amI
yoo
who
amI
Frame Pointer %ebp
Stack Pointer %esp


who(...)
{
• • •
amI();
• • •
amI();
• • •
}
Call Chain
yoo
who
amI
amI
amI
Frame
Pointer
%ebp
Stack
Pointer
%esp
yoo
who


amI(...)
{
}

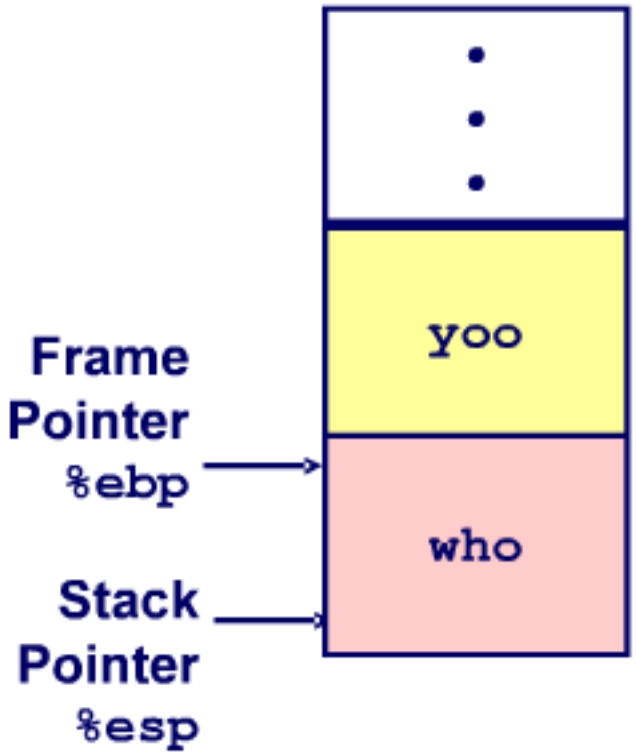
Call Chain

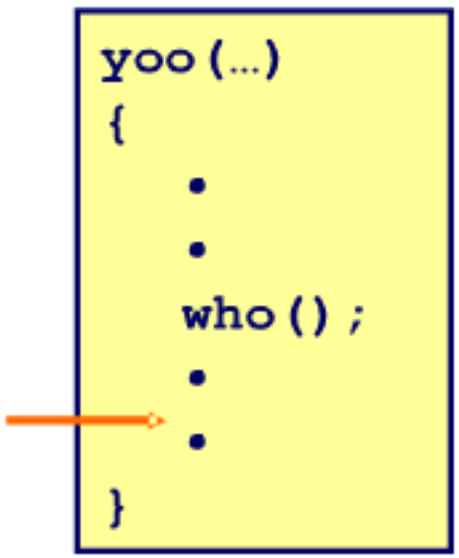

yoo
who
amI
amI
amI
amI


yoo
who
amI
Frame Pointer %ebp
Stack Pointer %esp

```
who(...)
{
    • • •
    amI();
    • • •
    amI();
    • • •
}
```

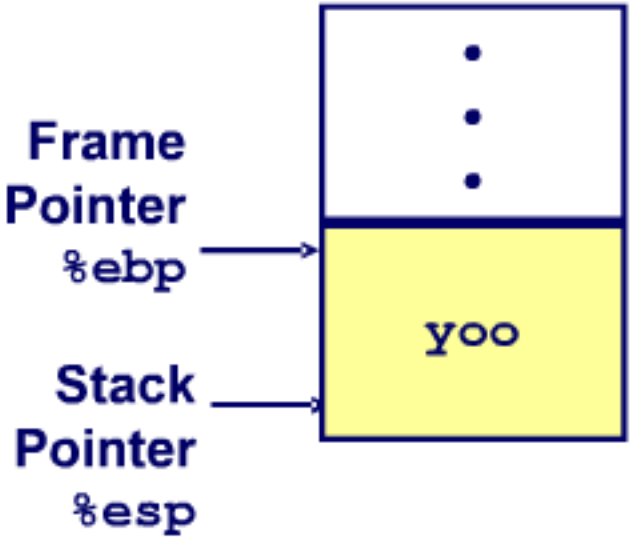
Call Chain




yoo(...)
{
•
•
who();
•
•
}

Call Chain


yoo
who
amI
amI
amI
amI


Frame
Pointer
%ebp
Stack
Pointer
%esp
yoo

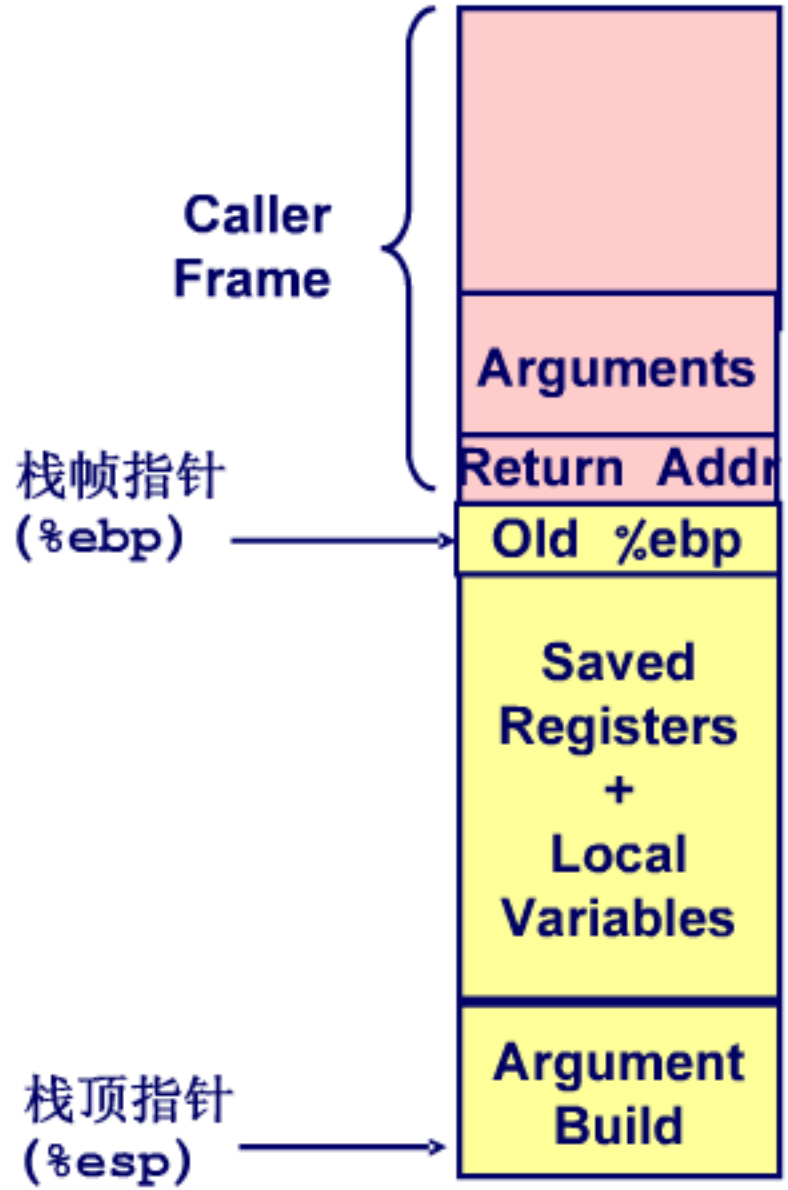
x86-32/Linux 下的栈帧

当前栈帧的内容(自“顶”向下)

- 子过程参数
  - “Argument build”
- 局部变量
  - 因为通用寄存器个数有限
- 被保存的寄存器值
- 父过程的栈帧起始地址（old %ebp）

父过程的栈帧中与当前过程相关的内容

- 返回地址
  - 由call 指令存入
- 当前过程的输入参数
- ...



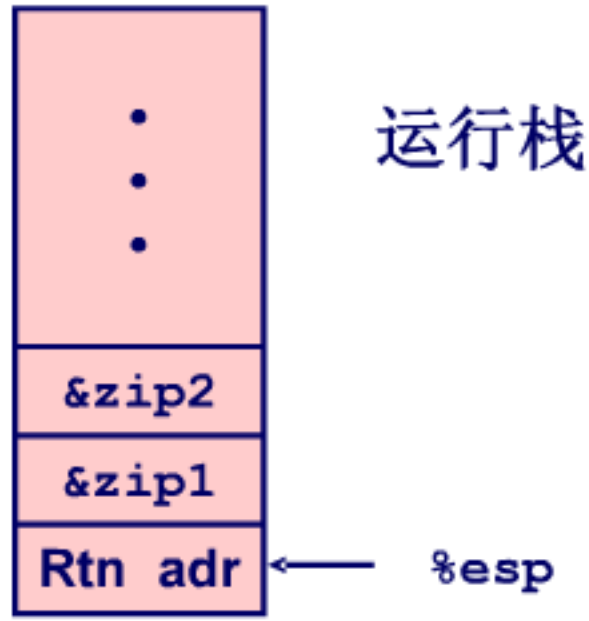
回顾下swap过程

```
int zip1 = 15213;
int zip2 = 91125;

void call_swap()
{
  swap(&zip1, &zip2);
}
```

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

Calling swap from call_swap

```
call_swap:
    . . .
    pushl $zip2    # Global Var
    pushl $zip1    # Global Var
    call swap
    . . .
```



```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;

  *xp = t1;
  *yp = t0;
}
```

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx

    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)

    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

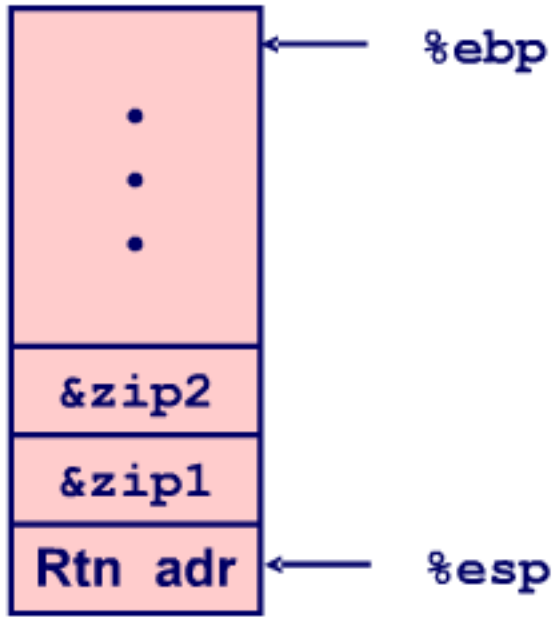
Set Up

Body

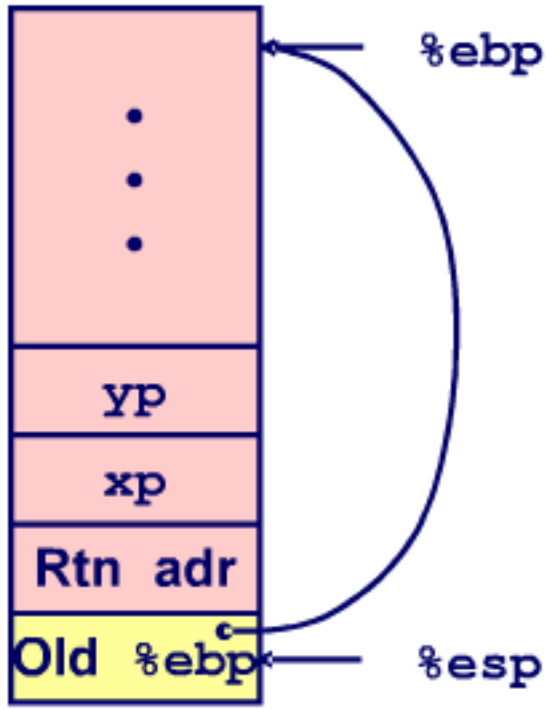
Finish

swap Setup #1

Entering Stack



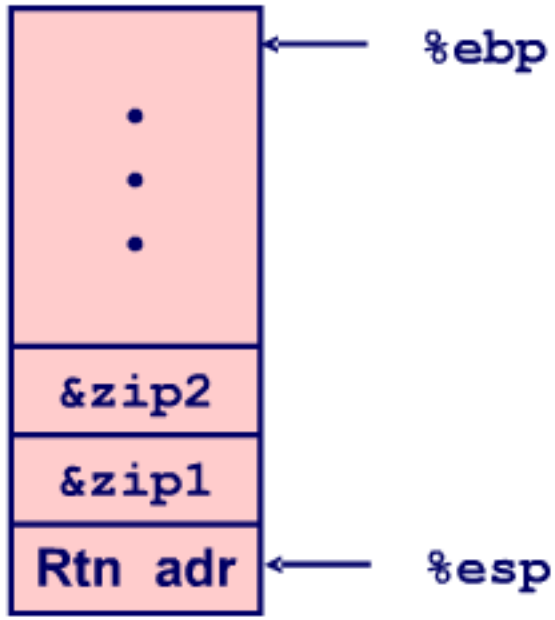
Resulting Stack



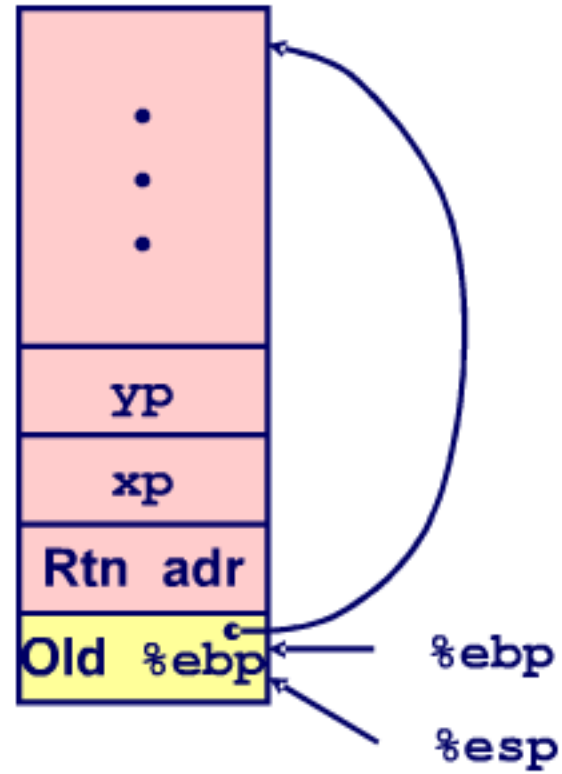

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

swap Setup #2

Entering Stack



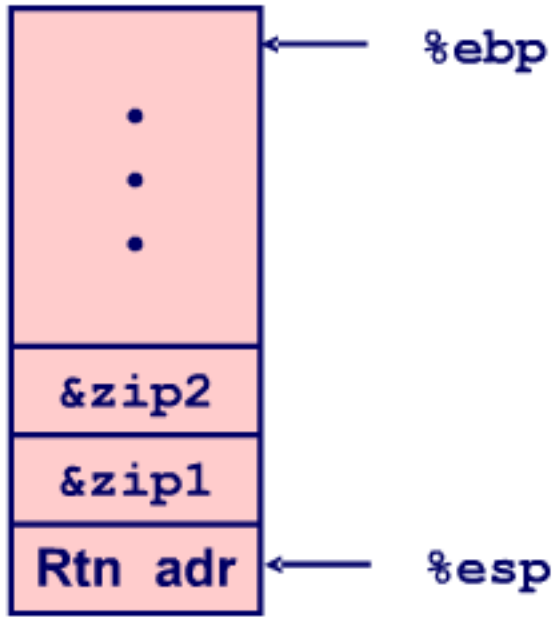
Resulting Stack



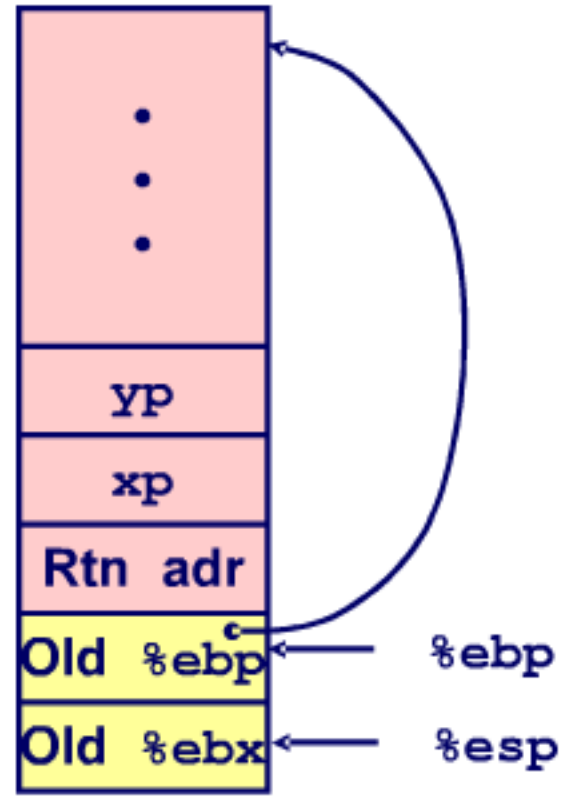

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

swap Setup #3

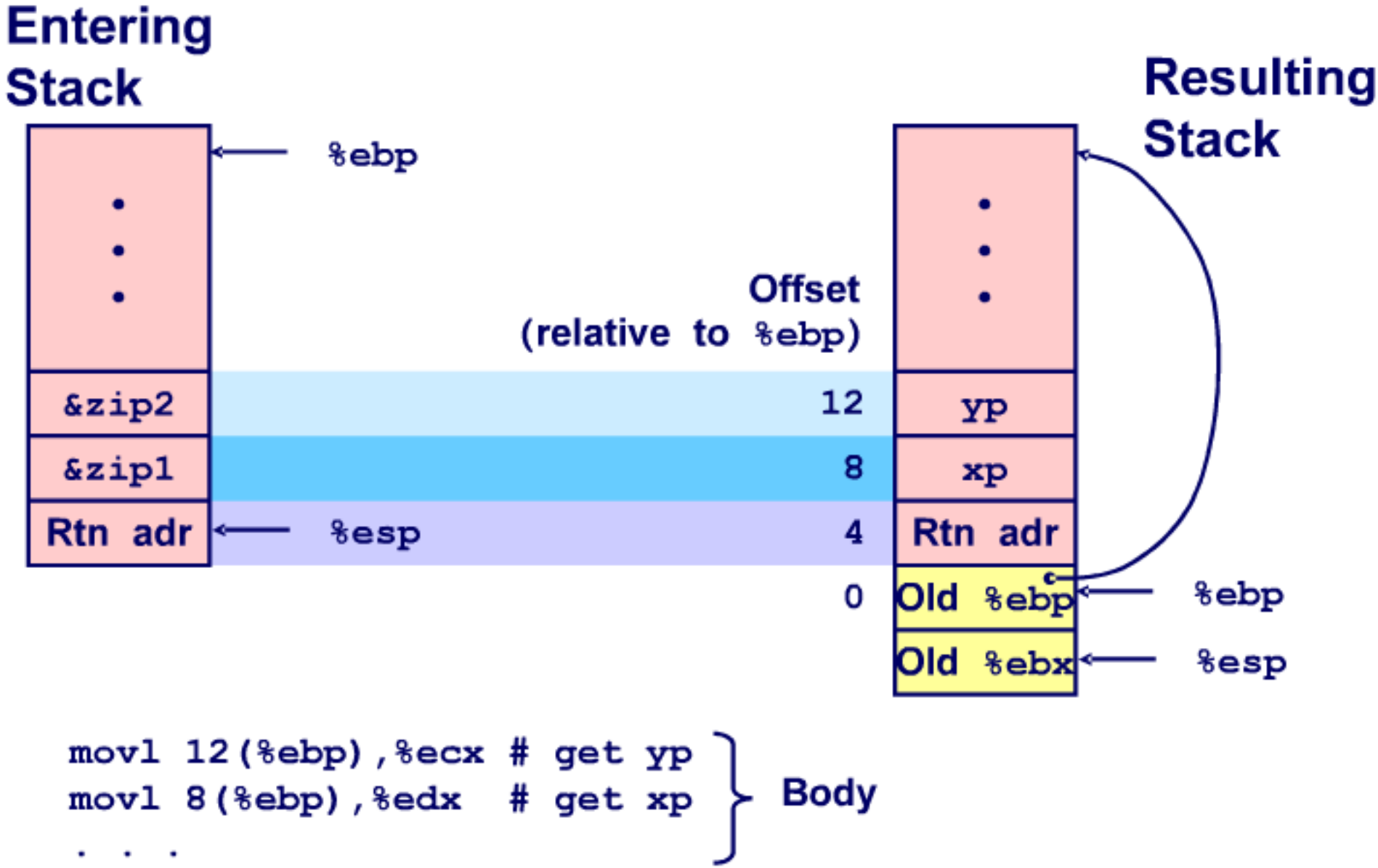
Entering Stack



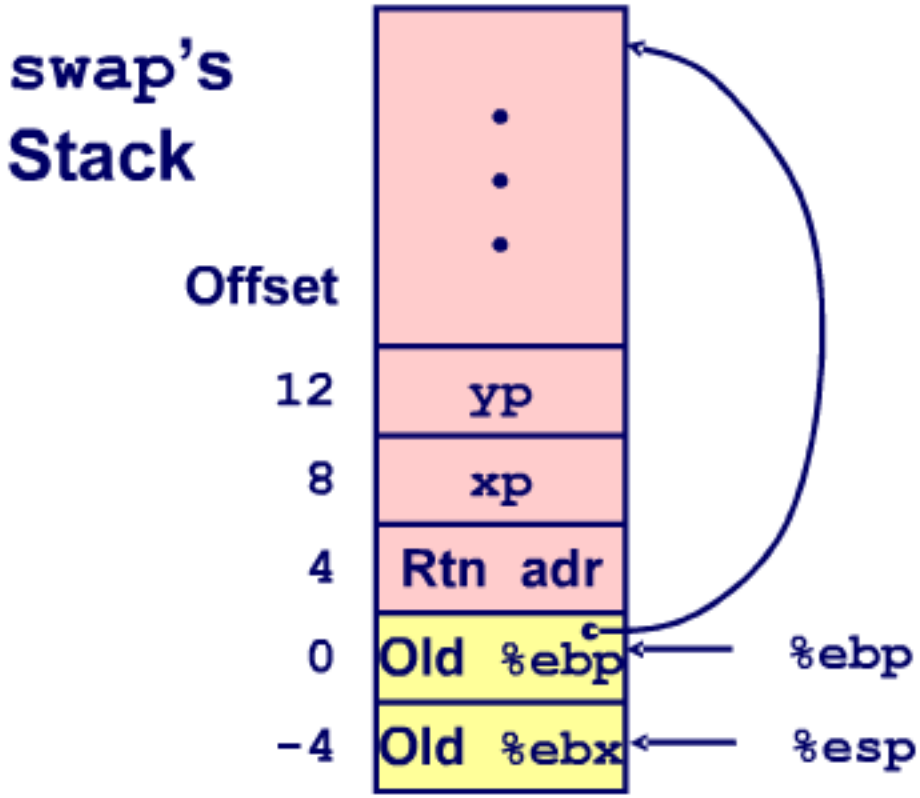
Resulting Stack

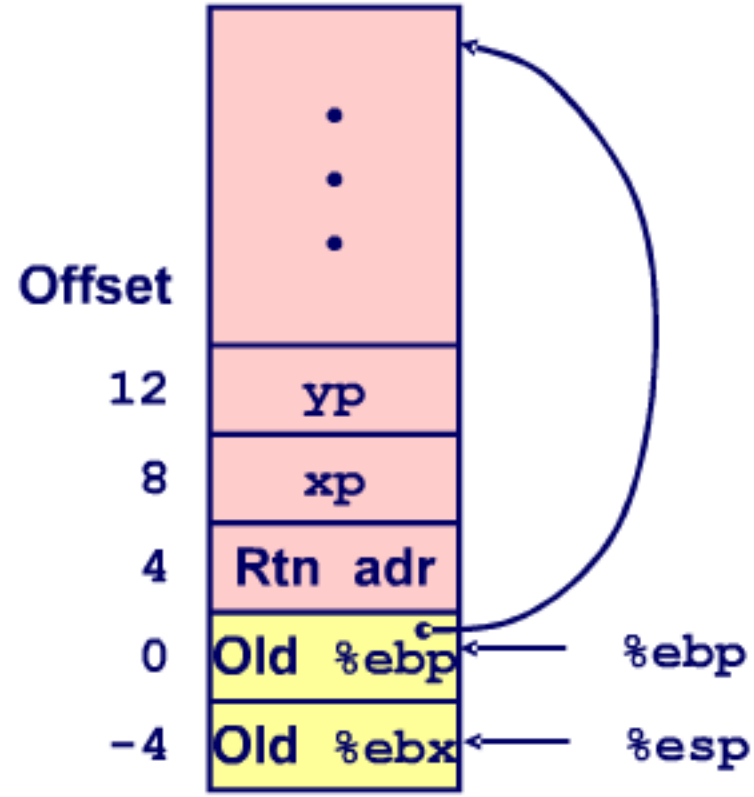



```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```


Entering Stack
Resulting Stack
%ebp
Offset (relative to %ebp)
&zip2
12
yp
&zip1
8
xp
Rtn adr
%esp
4
Rtn adr
0
Old %ebp
%ebp
Old %ebx
%esp
movl 12(%ebp),%ecx # get yp
movl 8(%ebp),%edx # get xp
. . .
Body

swap Finish #1



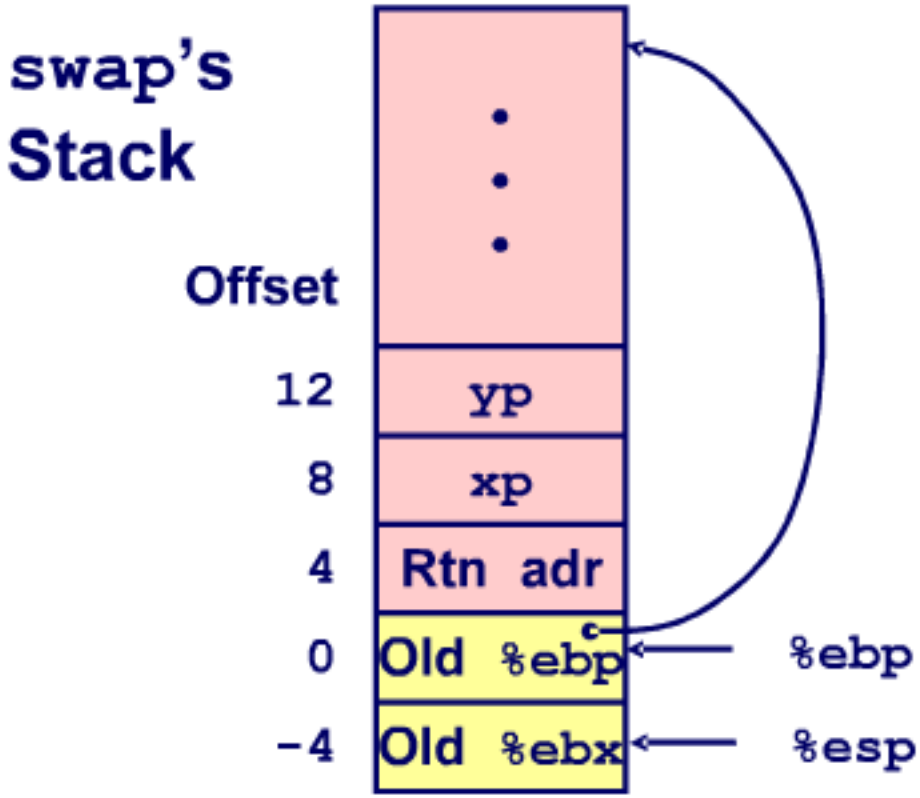


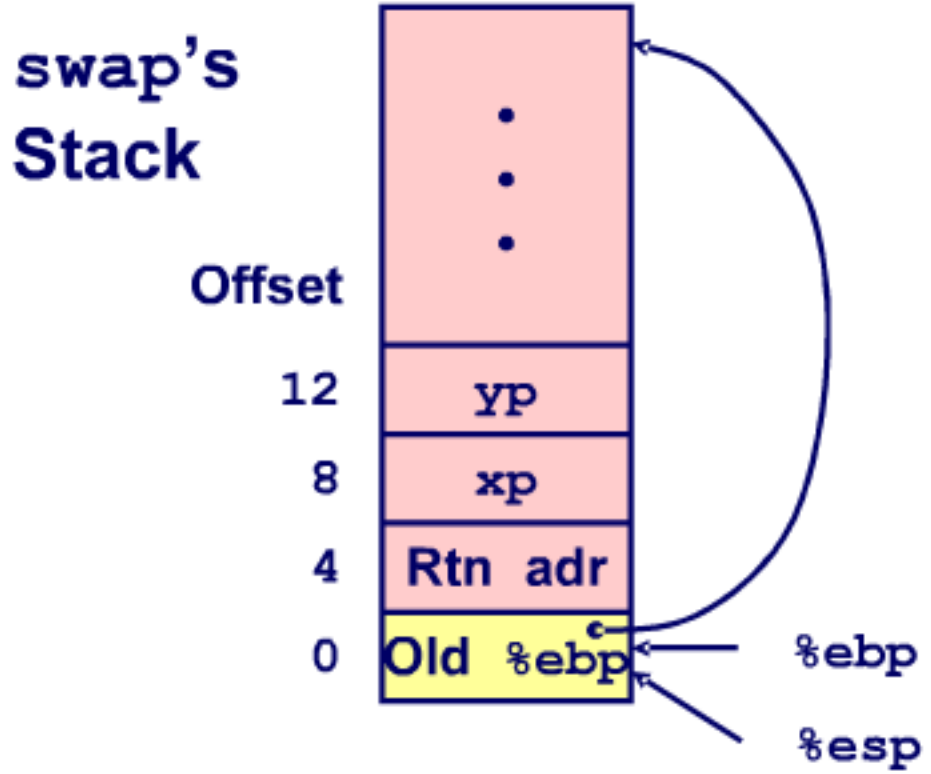

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

Observation

- Saved & restored register `%ebx`

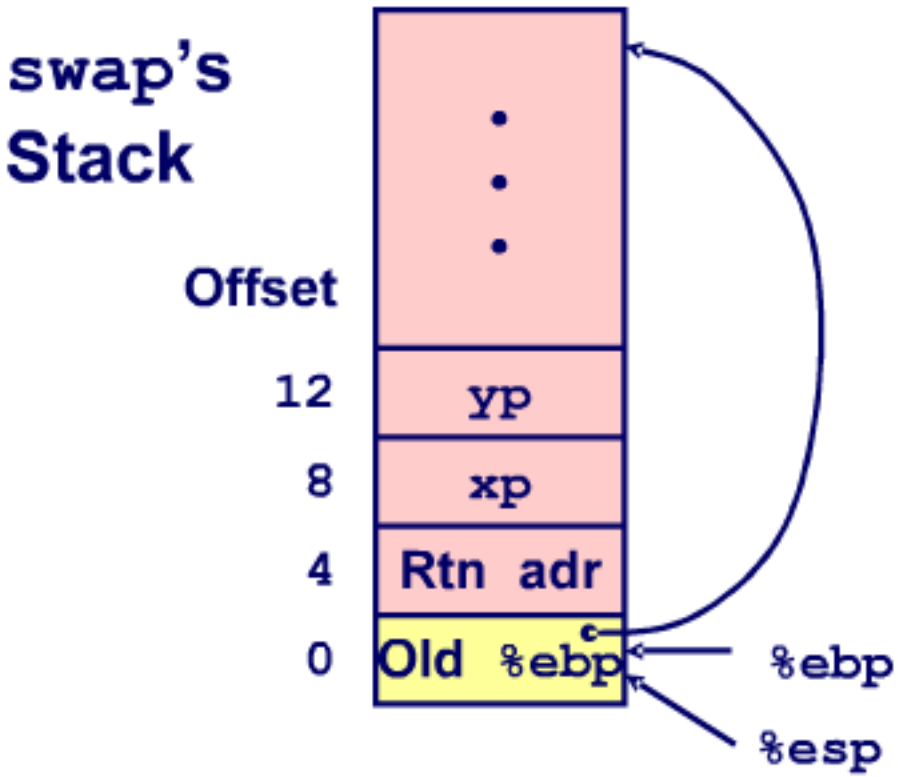
swap Finish #2

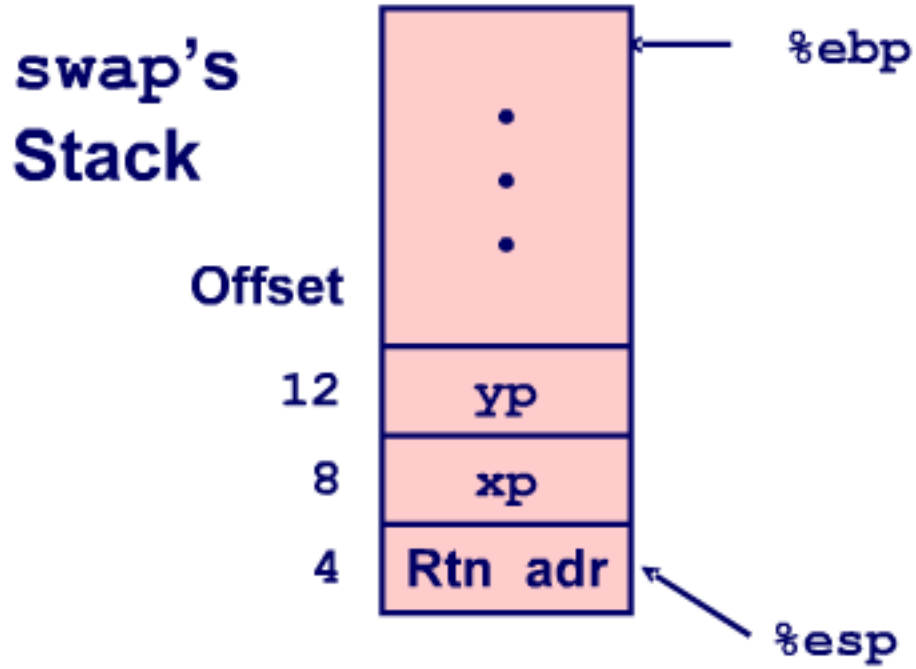





```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

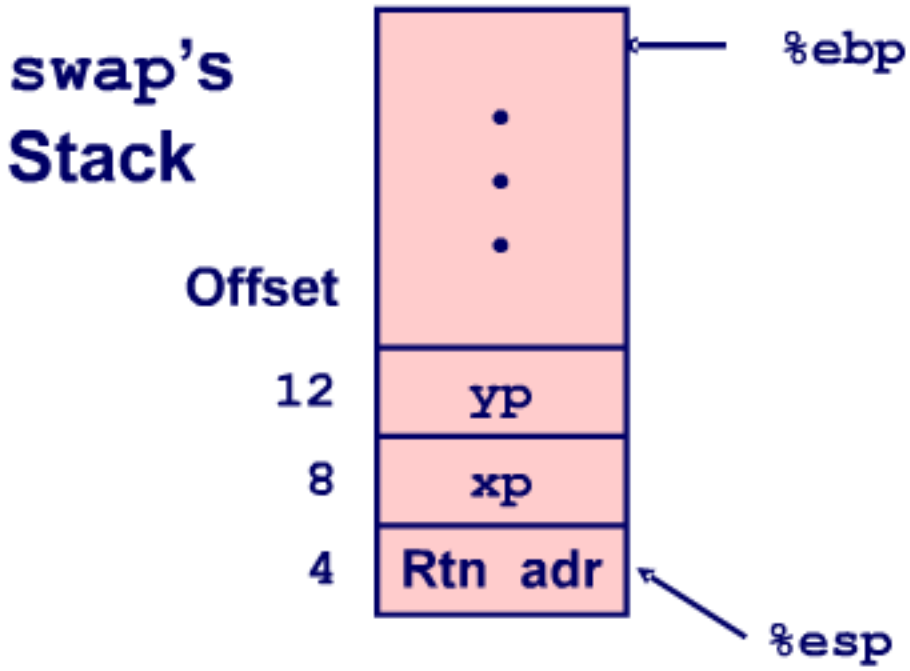
swap Finish #3

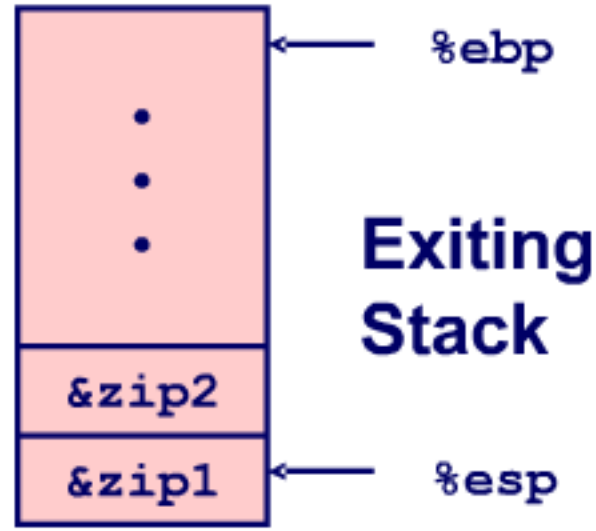





```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

swap Finish #4






```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

Observation

- Saved & restored register `%ebx`
- Didn't do so for `%eax`, `%ecx`, or `%edx`

寄存器使用惯例

过程yoo调用who

- yoo : *caller*　　who : *callee*

如何使用寄存器作为程序的临时存储?

```
yoo:
    • • •
    movl $15213, %edx
    call who
    addl %edx, %eax
    • • •
    ret
```

```
who:
    • • •
    movl 8(%ebp), %edx
    addl $91125, %edx
    • • •
    ret
```

- %edx ? ? ?

过程yoo调用who

- `yoo` : *caller*　　`who` : *callee*

如何使用寄存器作为程序的临时存储?

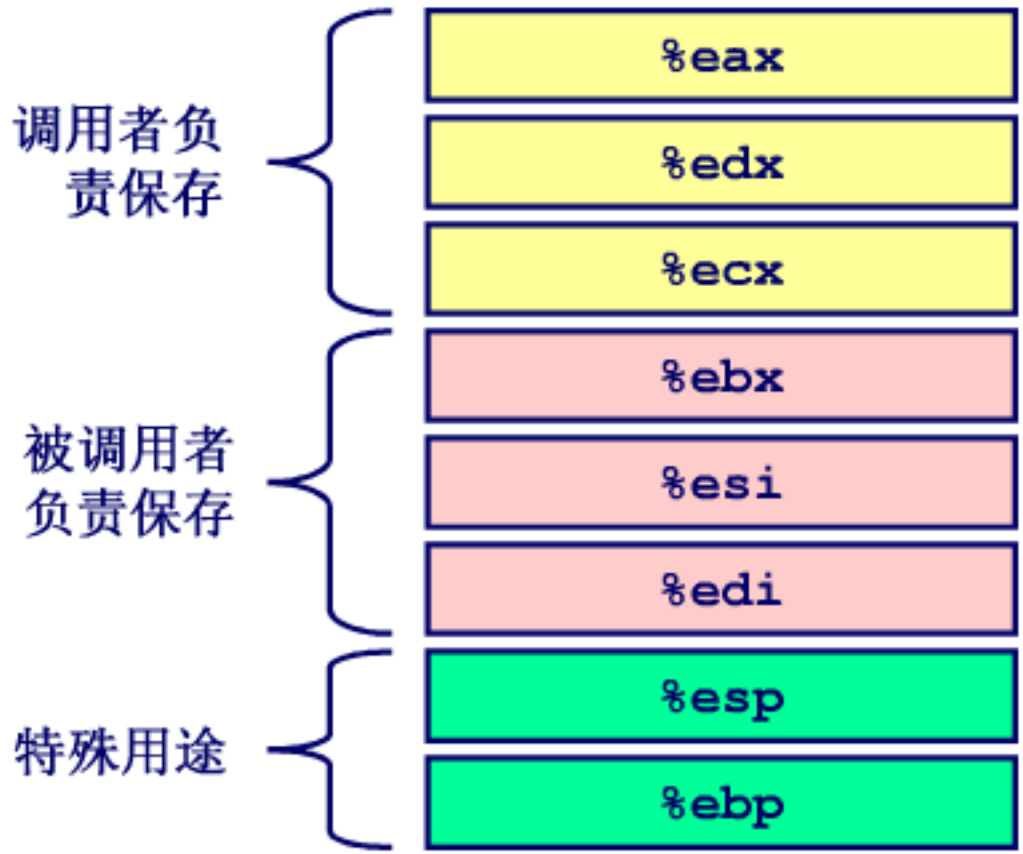
使用惯例——通用寄存器分为两类

- “调用者负责保存”
  - Caller在调用子过程之前将这些寄存器内容存储在它的栈帧内
- “被调用者负责保存”
  - Callee 在使用这些寄存器之前将其原有内容存储在它的栈帧内

X86-32/Linux 下的使用惯例

8个Registers

- 两个特殊寄存器
  `%ebp, %esp`
- 三个由被调用者负责保存
  `%ebx, %esi, %edi`
- 三个由调用者负责保存
  `%eax, %edx, %ecx`
- `%eax`用于保存过程返回值



递归调用

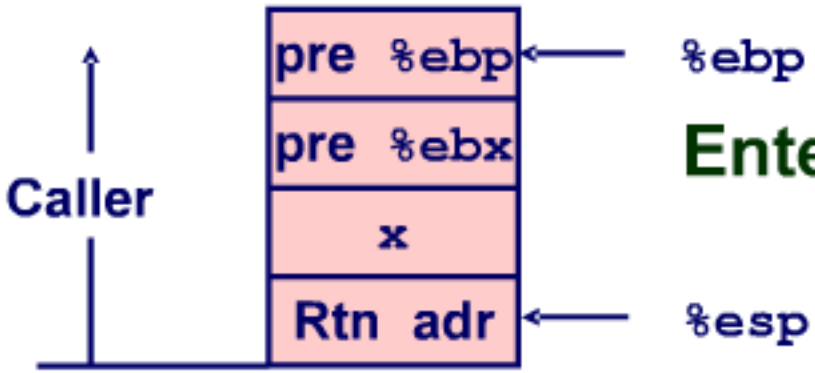
```
int rfact(int x)
{
  int rval;
  if (x <= 1)
    return 1;
  rval = rfact(x-1);
  return rval * x;
}
```

寄存器使用情况

- %eax 直接使用
- %ebx 使用前保存旧值，退出前恢复

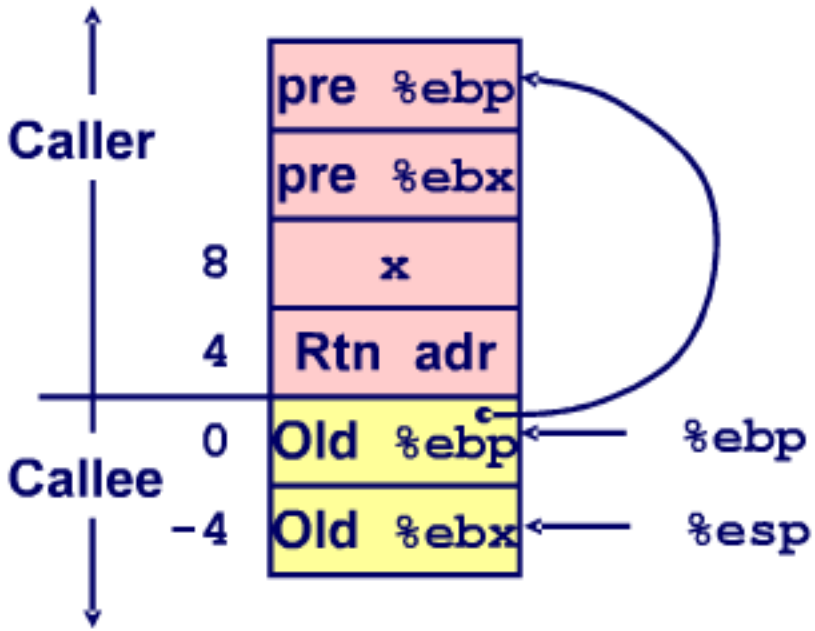
```
.globl rfact
    .type
rfact,@function
rfact:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 8(%ebp),%ebx
    cmpl $1,%ebx
    jle .L78
    leal -1(%ebx),%eax
    pushl %eax
    call rfact
    imull %ebx,%eax
    jmp .L79
    .align 4
.L78:
    movl $1,%eax
.L79:
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

栈帧的建立（分配）过程



Entering Stack

```
rfact:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```



递归体

```
  movl 8(%ebp),%ebx    # ebx = x
  cmpl $1,%ebx         # Compare x : 1
  jle .L78             # If <= goto Term
  leal -1(%ebx),%eax   # eax = x-1
  pushl %eax           # Push x-1
  call rfact           # rfact(x-1)
  imull %ebx,%eax      # rval * x
  jmp .L79             # Goto done
.L78:                  # Term:
  movl $1,%eax         # return val = 1
.L79:                  # Done:
```

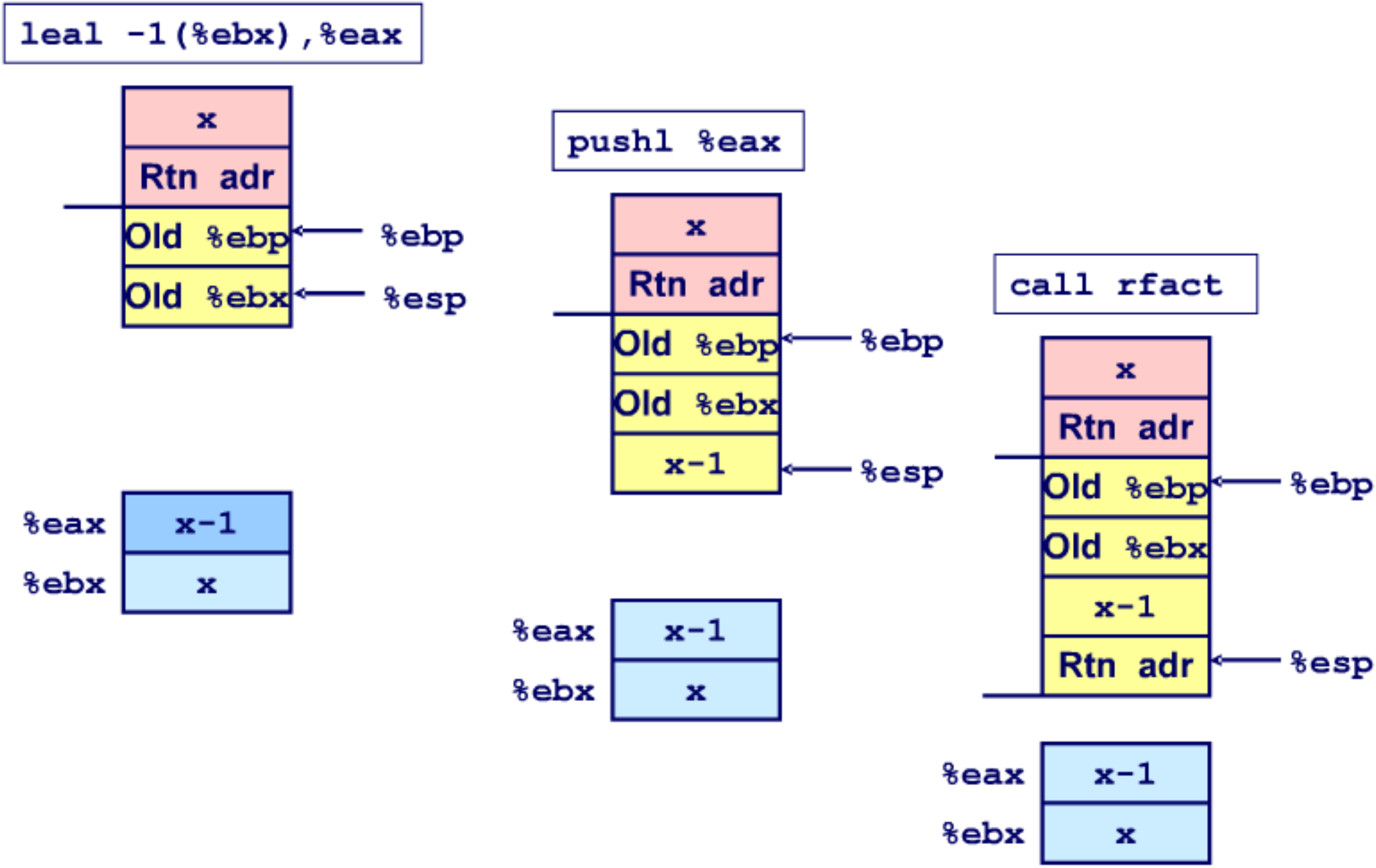
```
int rfact(int x)
{
  int rval;
  if (x <= 1)
    return 1;
  rval = rfact(x-1) ;
  return rval * x;
}
```

寄存器的使用

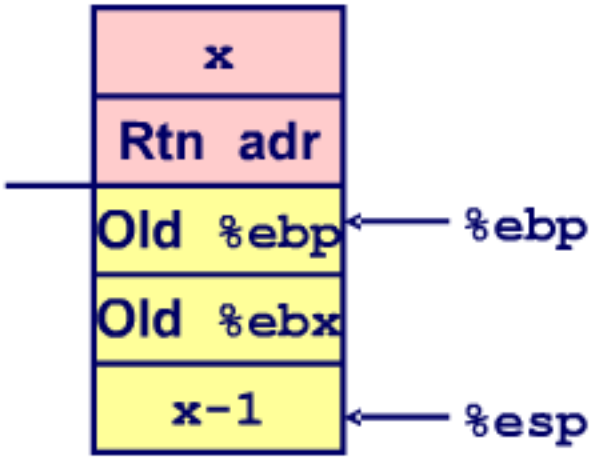
%ebx 存储x的值

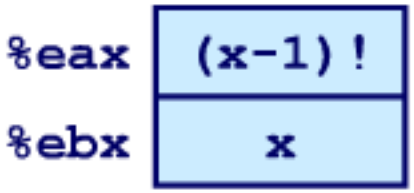
%eax

- 临时存储x-1
- 过程实例rfact(x-1)的返回值
- 本过程的返回值


leal -1(%ebx),%eax
x
Rtn adr
Old %ebp
Old %ebx
%ebp
%esp
%eax
x-1
%ebx
x
pushl %eax
x
Rtn adr
Old %ebp
Old %ebx
x-1
%ebp
%esp
%eax
x-1
%ebx
x
call rfact
x
Rtn adr
Old %ebp
Old %ebx
x-1
Rtn adr
%ebp
%esp
%eax
x-1
%ebx
x

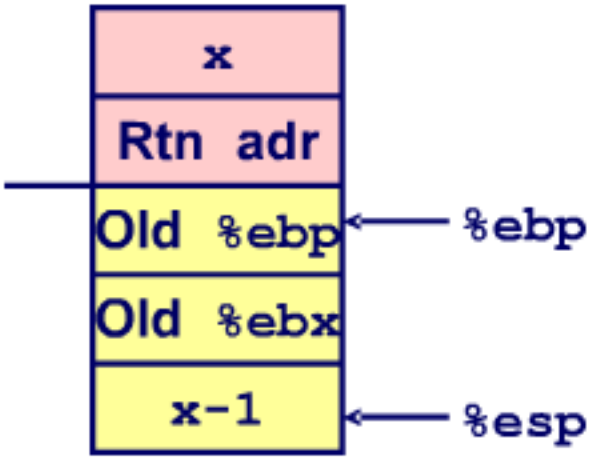
Return from Call

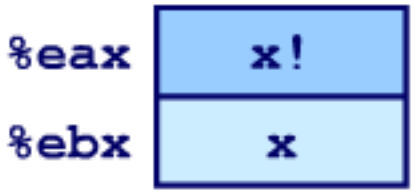




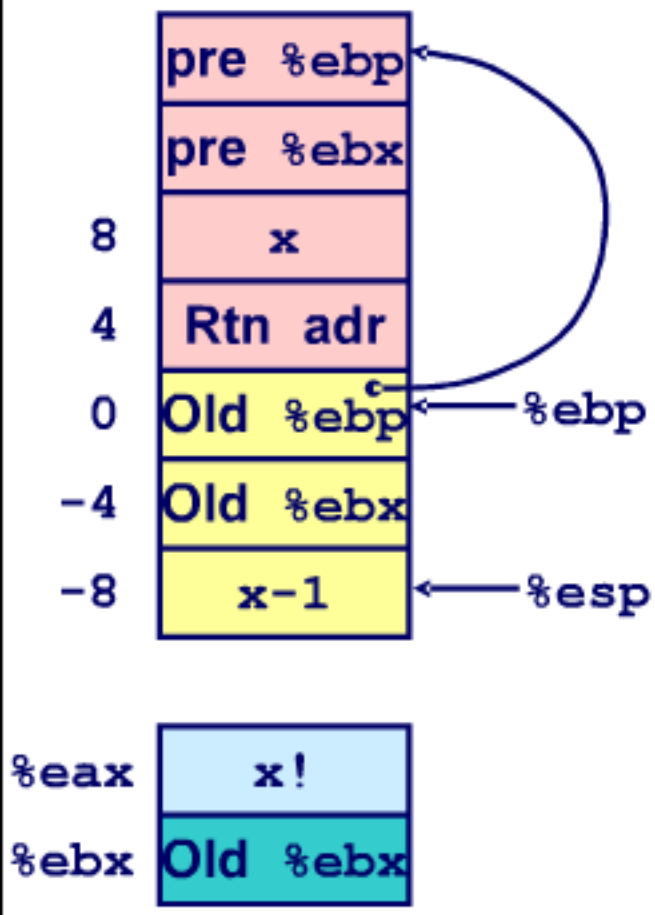
Assume that `rfact(x-1)` returns `(x-1)!` in register `%eax`

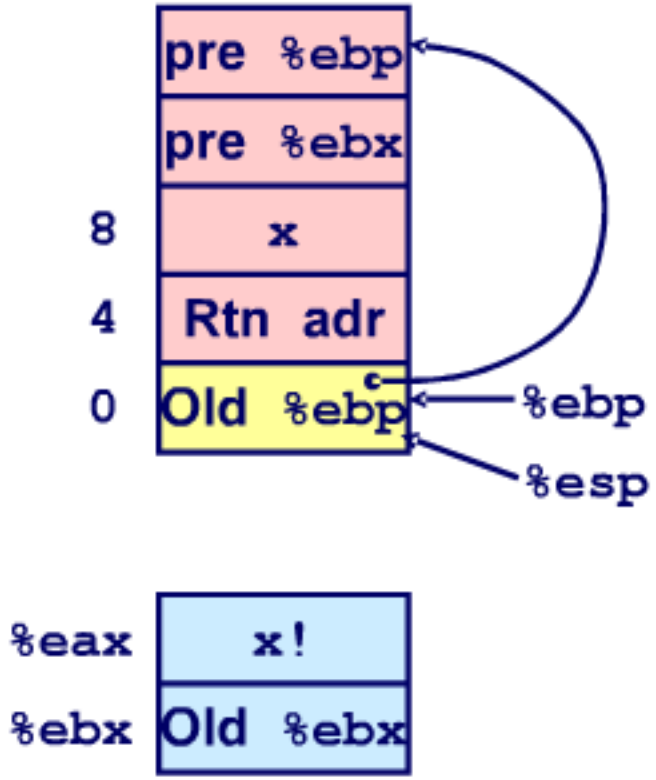
`imull %ebx,%eax`

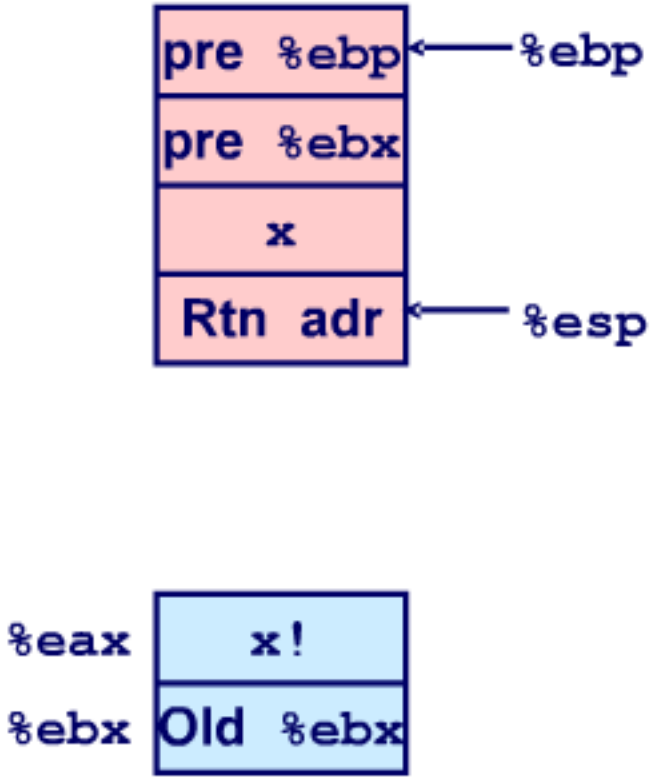




```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```







带指针的“阶乘”过程

Recursive Procedure

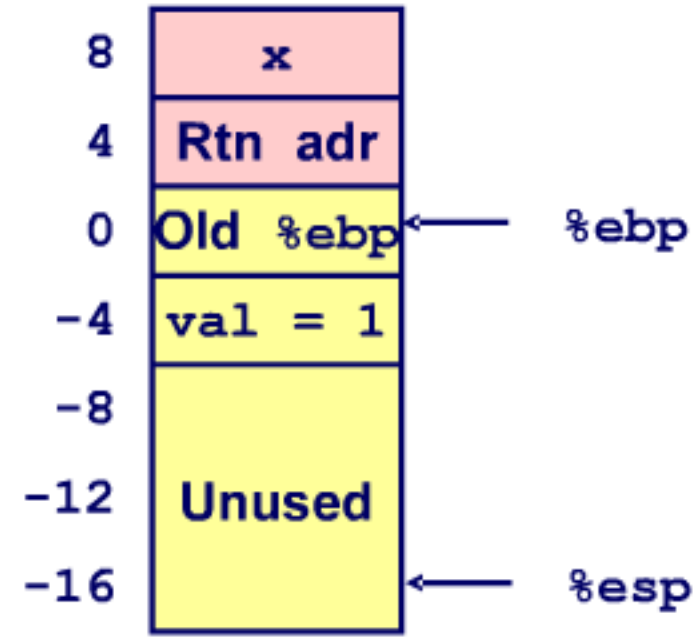
```
void s_helper
  (int x, int *accum)
{
  if (x <= 1)
    return;
  else {
    int z = *accum * x;
    *accum = z;
    s_helper (x-1,accum);
  }
}
```

Top-Level Call

```
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
```

创建指针

```
_sfact:
   pushl %ebp           # Save %ebp
   movl %esp,%ebp       # Set %ebp
   subl $16,%esp        # Add 16 bytes
   movl 8(%ebp),%edx    # edx = x
   movl $1,-4(%ebp)     # val = 1
```



使用栈存储临时变量

- Val必须存在栈内
  - 因为需要一个指针
- Val的地址为%ebp-4
- 并将其压入栈（作为第二个参数）

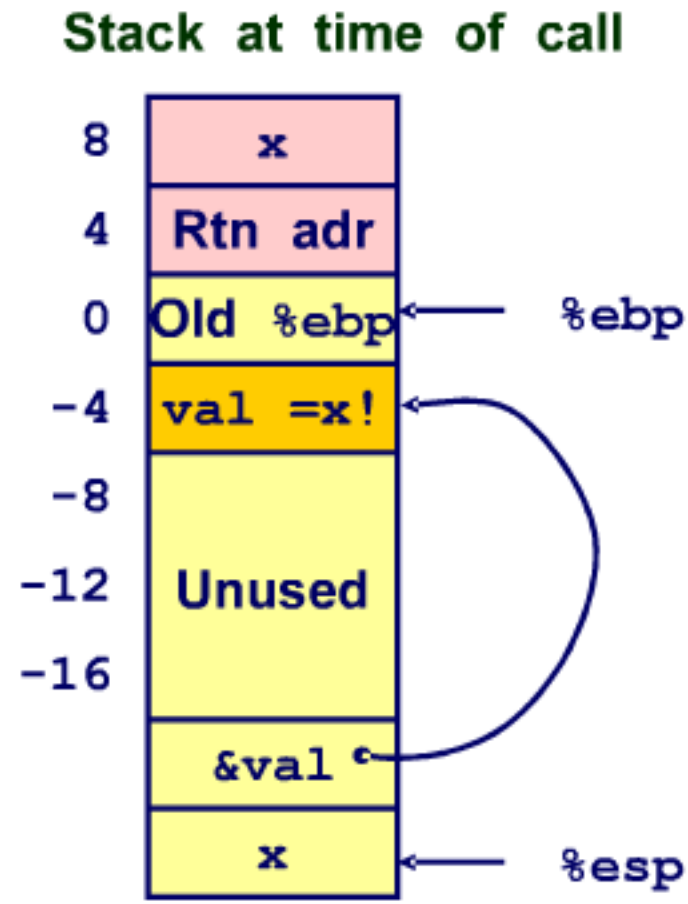
```
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
```

传递指针

调用s_helper 过程

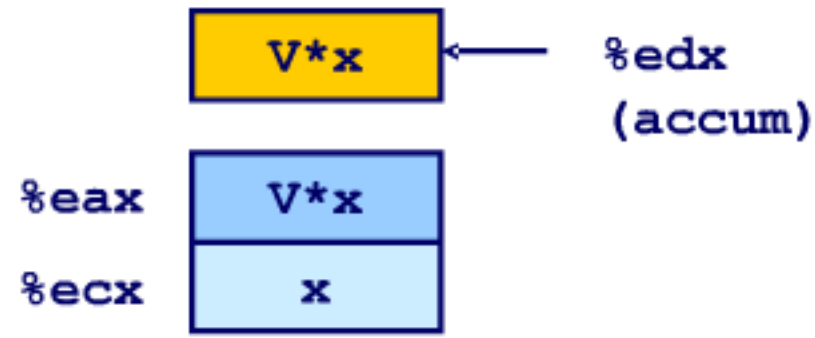
```
leal -4(%ebp),%eax  # Compute &val
pushl %eax          # Push on stack
pushl %edx          # Push x
call s_helper       # call
movl -4(%ebp),%eax  # Return val
• • •               # Finish
```

```
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
```



使用指针

```
void s_helper
   (int x, int *accum)
{
   • • •
     int z = *accum * x;
     *accum = z;
   • • •
}
```




```
   • • •
movl %ecx,%eax     # z = x
imull (%edx),%eax  # z *= *accum
movl %eax,(%edx)   # *accum = z
   • • •
```

- %ecx 存储变量x
- %edx 存储变量accum

X86-32 过程调用小结

程序栈

- 各个过程运行实例的似有空间
  - 不同实例间避免相互干扰
  - 过程本地变量与参数存于栈内（采用相对于栈帧基址%ebp的寻址）
- 符合栈的基本工作规律
  - 过程返回顺序与过程调用的顺序相反

相关指令与寄存器使用惯例

- **Call / Ret 指令**
- 寄存器使用惯例
  - 调用者/ 被调用者 保存
  - %ebp / %esp两个特殊寄存器
- 栈帧的存储内容

x86-64 通用寄存器

- 相比于32位结构，寄存器个数加倍
  - Accessible as 8, 16, 32, or 64 bits

x86-64 寄存器使用惯例

%rax	Return Value
%rbx	Callee Saved
%rcx	Argument #4
%rdx	Argument #3
%rsi	Argument #2
%rdi	Argument #1
%rsp	Stack Pointer
%rbp	Callee Saved

%r8	Argument #5
%r9	Argument #6
%r10	Callee Saved
%r11	Used for linking
%r12	C: Callee Saved
%r13	Callee Saved
%r14	Callee Saved
%r15	Callee Saved

x86-64 寄存器

过程参数（不超过6个）通过寄存器传递

- 大于6个的仍使用栈传递
- 这些传递参数的寄存器可以看成是“调用者保存”寄存器

所有对于栈帧内容的访问都是基于%esp完成的

- %ebp完全用作通用寄存器

x86-64 下的swap过程-1

```
void swap(long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    movq    (%rdi), %rdx
    movq    (%rsi), %rax
    movq    %rax, (%rdi)
    movq    %rdx, (%rsi)
    ret
```

- 参数由寄存器传递
  - First (xp) in %rdi, second (yp) in %rsi
  - 64位指针
- 无需任何栈操作
  - 局部变量也存储于寄存器中

x86-64 下的swap过程-2

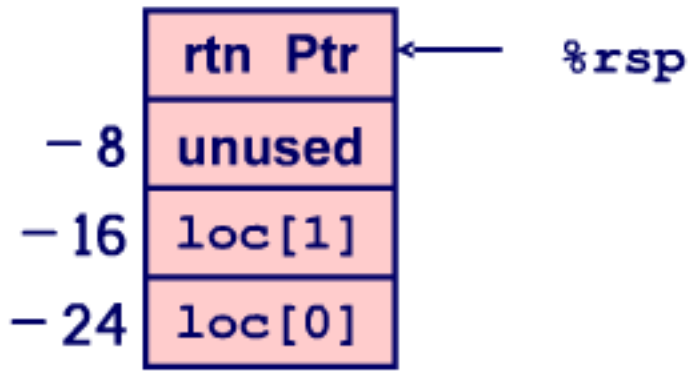
```
/* Swap, using local array */
void swap_a(long *xp, long *yp)
{
    volatile long loc[2];
    loc[0] = *xp;
    loc[1] = *yp;
    *xp = loc[1];
    *yp = loc[0];
}
```

```
swap_a:
   movq   (%rdi), %rax
   movq   %rax, -24(%rsp)
   movq   (%rsi), %rax
   movq   %rax, -16(%rsp)
   movq   -16(%rsp), %rax
   movq   %rax, (%rdi)
   movq   -24(%rsp), %rax
   movq   %rax, (%rsi)
   ret
```

强制使用栈空间

- volatile变量

但在实际使用中没有修改栈顶寄存器（%rsp）



x86-64 中的swap过程-3

```
long scount = 0;
/* Swap a[i] & a[i+1] */
void swap_ele_se
   (long a[], int i)
{
     swap(&a[i], &a[i+1]);
     scount++;
}
```

swap_ele_se没有分配栈帧，为什么？

```
swap_ele_se:
   movslq %esi,%rsi                  # Sign extend i
   leaq   (%rdi,%rsi,8), %rdi        # &a[i]
   leaq   8(%rdi), %rsi              # &a[i+1]
   call   swap                       # swap()
   incq   scount(%rip)               # scount++;
   ret
```

x86-64 中的swap过程-4

```
long scount = 0;
/* Swap a[i] & a[i+1] */
void swap_ele
  (long a[], int i)
{
    swap(&a[i], &a[i+1]);
}
```

使用jmp指令调用过程，为什么？swap将会返回到哪儿？

```
swap_ele:
  movslq %esi,%rsi              # Sign extend i
  leaq   (%rdi,%rsi,8), %rdi    # &a[i]
  leaq   8(%rdi), %rsi          # &a[i+1]
  jmp    swap                   # swap()
```

x86-64 的栈帧使用实例

```
long sum = 0;
/* Swap a[i] & a[i+1] */
void swap_ele_su
   (long a[], int i)
{
     swap(&a[i], &a[i+1]);
     sum += a[i];
}
```

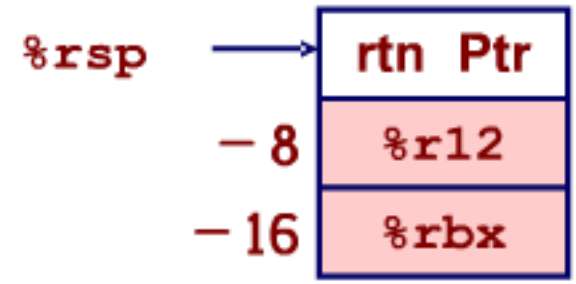
- 变量a与i的值存于“被调用者保存”的寄存器中
- 因此必须分配栈帧来保存这些寄存器
  - 与swap_ele_se有何不同?

```
swap_ele_su:
   movq     %rbx, -16(%rsp)
   movslq   %esi,%rbx
   movq     %r12, -8(%rsp)
   movq     %rdi, %r12
   leaq     (%rdi,%rbx,8), %rdi
   subq     $16, %rsp
   leaq     8(%rdi), %rsi
   call     swap
   movq     (%r12,%rbx,8), %rax
   addq     %rax, sum(%rip)
   movq     (%rsp), %rbx
   movq     8(%rsp), %r12
   addq     $16, %rsp
   ret
```

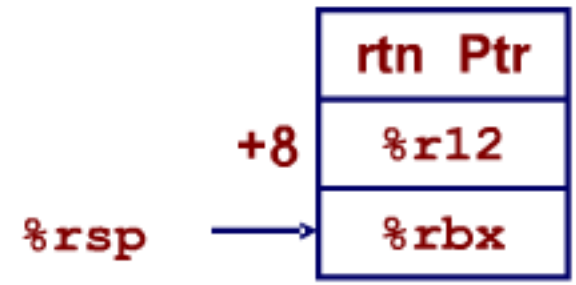
```
swap_ele_su:
   movq     %rbx, -16(%rsp)   # Save %rbx
   movslq   %esi,%rbx         # Extend & save i
   movq     %r12, -8(%rsp)    # Save %r12
   movq     %rdi, %r12        # Save a
   leaq     (%rdi,%rbx,8), %rdi  # &a[i]
   subq     $16, %rsp         # Allocate stack frame
   leaq     8(%rdi), %rsi     #      &a[i+1]
   call     swap              # swap()
   movq     (%r12,%rbx,8), %rax  # a[i]
   addq     %rax, sum(%rip)   # sum += a[i]
   movq     (%rsp), %rbx      # Restore %rbx
   movq     8(%rsp), %r12     # Restore %r12
   addq     $16, %rsp         # Deallocate stack frame
   ret
```

栈操作

```
movq   %rbx, -16(%rsp)   # Save %rbx
movq   %r12, -8(%rsp)    # Save %r12

subq   $16, %rsp         # Allocate stack frame
```




```
movq   (%rsp), %rbx      # Restore %rbx
movq   8(%rsp), %r12     # Restore %r12

addq   $16, %rsp         # Deallocate stack frame
```



X86-64下的栈帧的一些不同操作特性

一次性分配整个帧

- 将%rsp减去某个值（栈帧的大小）
- 对于栈帧内容的访问都是基于%rsp完成的
- 可以延迟分配
  - 可以直接访问不超过当前栈指针(%rsp)128字节的栈上空间

释放简单

- %rsp 直接加上某个值（栈帧的大小）

x86-64 过程调用小结

频繁使用寄存器

- 参数传递

减少对栈的使用

- 甚至不使用栈
- 一次性分配与释放栈

优化操作

- 可以直接使用部分的栈空间而不分配
- 使用jmp指令调用过程
- ...

练习题

一种C函数调用的参数传递与栈恢复模式被称为fastcall，其与我们课堂上学的默认的C函数参数传递与栈恢复方法不同(称为cdecl)。请根据下列的C代码与汇编代码的对应关系，写出fastcall的参数是如何传递的，即哪些参数是通过通用寄存器传递（包括对应关系如何），哪些是通过栈传递、压栈顺序如何，以及传递参数的栈是由被调者还是调用者负责恢复。

```
int __fastcall dummy(int w, int x, int y, int z )
{
    return (w*2 + (x+y)>>2 - y*23 ) * z;
}

void calldummy(int w, int x, int y, int z)
{
    int res = dummy(w, x, y, z);
}
```

```
_calldummy:
    pushl       %ebp
    movl        %esp, %ebp
    subl        $8, %esp
    ......
    call        dummy
    subl        $8, %esp
    leave
    ret
```

```
dummy:
        ......
        movl    8(%ebp), %ebx
        addl    %ebx, %edx
        leal    (%edx,%ecx,2), %eax
        leal    (%ebx,%ebx,2), %edx
        movl    $2, %ecx
        sall    $3, %edx
        subl    %ebx, %edx
        subl    %edx, %ecx
        sarl    %cl, %eax
        movl    12(%ebp), %ecx
        imull   %ecx, %eax
        ......
        ret     $8
    ......
```

fastcall：

(1)函数的第一个和第二个**DWORD**参数（或者尺寸更小的）
通过**ecx**和**edx**传递，其他参数通过从右向左的顺序压栈

(2)被调用函数清理栈

汇编实验

助教：王宏伟　陆思羽
88wanghongwei@gmail.com
lusiyuhere@gmail.com

2013-9-9

实验内容

- BombLab
- BufLab

BombLab背景

- 最近，贵系接到了一个神圣的任务，那就是要破解一批二进制炸弹。
- 比较严重的是，二进制炸弹的数目刚好和同学们的数目相当，这意味着每个人的炸弹都**不一样**。
- 特别严重的是，一旦破解失败，会造成非常严重的后果（实验0分）。

BombLab代码（部分公布）

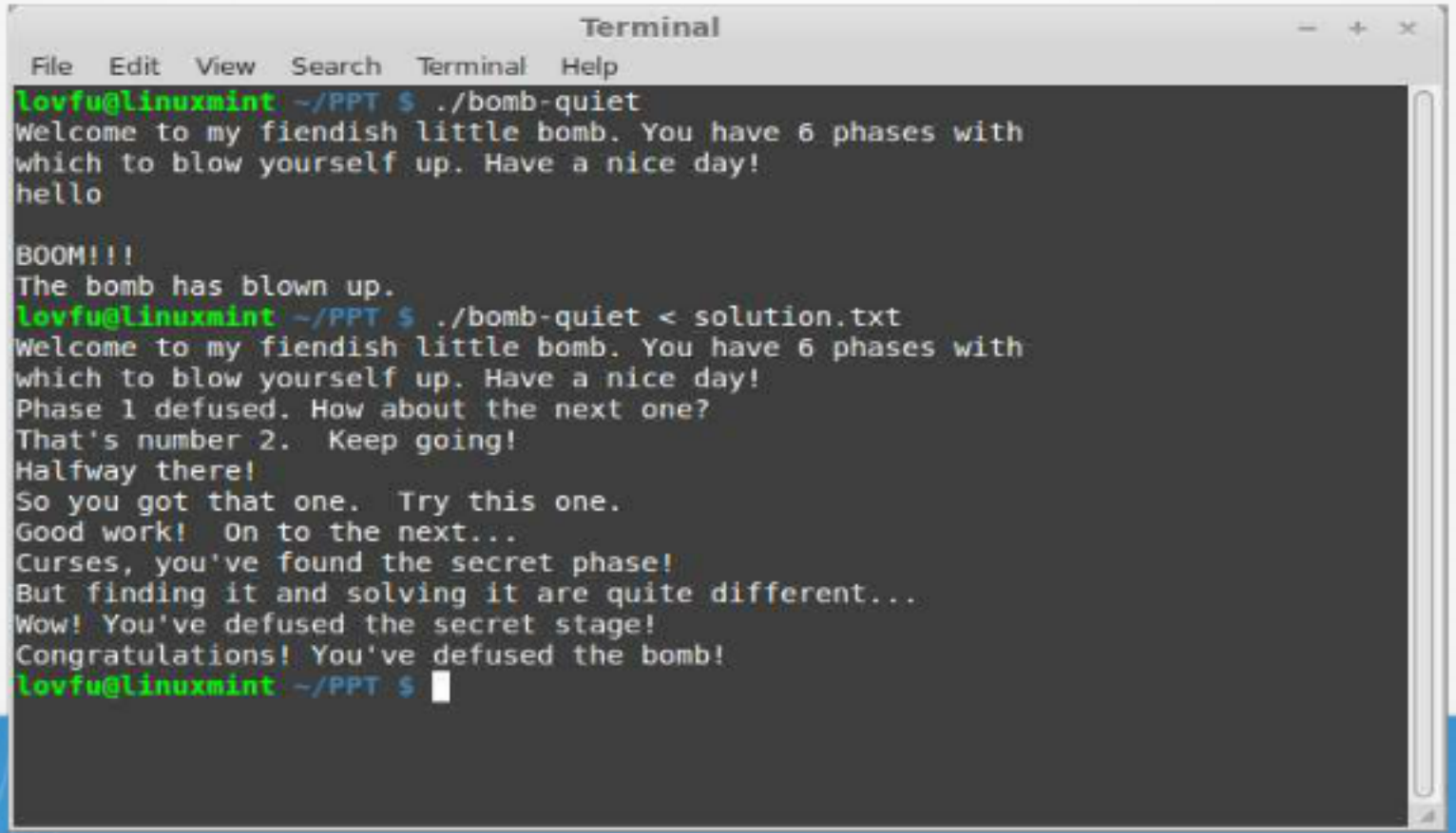
```
/* Hmm...  Six phases must be more secure than one phase! */
input = read_line();             /* Get input                  */
phase_1(input);                  /* Run the phase              */
phase_defused();                 /* Drat!  They figured it out!
                  * Let me know how they did it. */
printf("Phase 1 defused. How about the next one?\n");

/* The second phase is harder.  No one will ever figure out
 * how to defuse this... */
input = read_line();
phase_2(input);
phase_defused();
printf("That's number 2.  Keep going!\n");

/* I guess this is too easy so far.  Some more complex code will
 * confuse people. */
input = read_line();
phase_3(input);
phase_defused();
printf("Halfway there!\n");
```

BombLab实验

- Phase1：stringcomparison
- Phase2：loops
- Phase3：conditionals/switches
- Phase4：recursive calls and the stack discipline
- Phase5：pointers
- Phase6：linked lists/pointers/structs

BombLab背景

- 破解方法，即向stdin输入正确的字符串序列。



BufLab背景

- 本次试验，将利用**缓冲区溢出漏洞**攻击一个名为BufBomb的程序。
- 比较头疼的是，每个人所破解的BufBomb是不同的。
- 根本原因：C语言char[]数据结构的特性。
- 直接原因：字符串越界。

缓冲区溢出漏洞原理

输入=>n+8位

最后4位==>smoke()地址

BufLab代码（不公布）

```
void test()
{
    int val;
    /* Put canary on stack to detect possible corruption */
    volatile int local = uniqueval();

    val = getbuf();

    /* Check for corrupted stack */
    if (local != uniqueval()) {
        printf("Sabotaged!: the stack has been corrupted\n");
    }
    else if (val == cookie) {
        printf("Boom!: getbuf returned 0x%x\n", val);
        validate(3);
    } else {
        printf("Dud: getbuf returned 0x%x\n", val);
    }
}
```

```
int getbuf()
{
    char buf[NORMAL_BUFFER_SIZE];
    Gets(buf);
    return 1;
}
```

BufLab实验

- Level 0: Candle (10 pts)
- Level 1: Sparkler (10 pts)
- Level 2: Firecracker (15 pts)
- Level 3: Dynamite (20 pts)
- Level 4: Nitroglycerin (10 pts)

BufLab背景

- 攻击方法：即向stdin输入一定的字符串序列。

```
Terminal
File  Edit  View  Search  Terminal  Help
lovfu@linuxmint ~/PPT $ ./bufbomb -u 2012210853
Userid: 2012210853
Cookie: 0x2899448c
Type string:hello
Dud: getbuf returned 0x1
Better luck next time
lovfu@linuxmint ~/PPT $ ./bufbomb -u 2012210853
Userid: 2012210853
Cookie: 0x2899448c
Type string:hello hello hello hello hello hello hello hello hello hello hello he
llo hello hello hello hello hello hello hello hello hello hello hello hello hell
o hello hello hello hello hello hello hello hello hello hello hello hello hello
hello hello hello hello hello hello hello hello hello hello hello hello hello he
llo
Ouch!: You caused a segmentation fault!
Better luck next time
lovfu@linuxmint ~/PPT $ ./bufbomb -u 2012210853 < smoke-2012210853-raw.txt
Userid: 2012210853
Cookie: 0x2899448c
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
lovfu@linuxmint ~/PPT $
```

实验环境

- 服务器：166.111.68.242
- BombLab端口：15213
- BufLab端口：18213
- BombLab
  - 获取程序：166.111.68.242:15213
  - 实时成绩：166.111.68.242:15213/scoreboard
- BufLab
  - 获取程序：166.111.68.242:18213
  - 实时成绩：166.111.68.242:18213/scoreboard

实验环境

CS:APP Binary Bomb Request

Fill in the form and then click the Submit button.

Hit the Reset button to get a clean form.

Legal characters are spaces, letters, numbers, underscores ('_'), hyphens ('-'), at signs ('@'), and dots ('.').

User name
Enter your Unix login ID
2012210853

Email address
88wanghongwei@gmail.com

Submit Reset

Buffer Lab Scoreboard

Here is the latest information that we have received from your buffer bomb. If your submission is marked as **invalid**, then the testing code thinks your solution is invalid. Some possible reasons are:

- Your solution is not sufficiently robust, a good possibility at the nitroglycerin level. Try to design your exploit to be more tolerant of fluctuations in the stack position.
- You somehow bypassed the protocol used by our autograding service.

Last updated: Tue Aug 20 02:36:43 2013 (updated every 30 secs)

#	Cookie	Score (65)	Candle (0: 10pt)	Sparkler (1: 10pt)	Firecracker (2: 15pt)	Dynamite (3: 20pt)	Nitro (4: 10pt)
1	2899448c	10	10	-	-	-	-

1 students.
Valid submissions: Level 0: 1, Level 1: 0, Level 2: 0, Level 3: 0, Level 4: 0

实验环境

- Linux：linux mint/linux deepin/ubuntu
- 安装方式：双系统、wubi安装法、虚拟机

- windows：可以利用服务器（ssh）
- Xshell、Putty、SecureCRT
- 账号：学号
- 密码：学号

BombLab实验方式

- ./bomb < solution.txt
- ./bomb-quiet，本地实验（网络学堂下载）
- ./bomb，结果上传服务器（服务器下载）
- 可以多次下载炸弹，可以提交多个ID的炸弹，以最高成绩为准
- 通过网络学堂或服务器账户提交**BombID**、学号、solution.txt、 bomb、 Readme文件

BufLab 实验方式

./hex2raw<smoke-2012210853.txt>smoke-2012210853-raw.txt

./bufbomb -u StudentID < smoke-2012210853-raw.txt

./makecookie 2012210853

./bufbomb -u StudentID，本地实验

./bufbomb -u StudentID -s，结果上传服务器

通过网络学堂或服务器账户提交**Cookie**、学号、5个答案.txt、bufbomb、Readme

gdb 基础

- break(b) : 设置断点
  - break 57: 设置57行为断点(本实验用不着)
  - break *0x08048dfa: 设置0x08048dfa为断点

gdb 基础

- run (r): 运行
- continue(c): 从当前位置继续执行
- next(n): 不进入调用函数单步执行
- step(s): 进入调用函数单步执行
- list(l):查看代码
- info:查看相应信息
- quit(q):退出

gdb 基础

- delete(d) break(br): 删除所有断点
- set args [args]: 设置args
- disassemble(disas) [func_name]: 反汇编函数
- print 和display 打印变量的值

gdb 基础

- examine(x)/<n/f/u><addr>：显示内存地址的内容
- n:内存长度;f:格式;u:单位字节数
- 输出格式

  x：按十六进制格式显示变量。

  d ：按十进制格式显示变量。

  u：按十六进制格式显示无符号整型。

  o：按八进制格式显示变量。

  t：按二进制格式显示变量。

  c：按字符格式显示变量。

  f：按浮点数格式显示变量。

  s：按照字符串方式输出变量

gdb 基础

- 常用：

  x /4xh $ebp

  x /s $eax

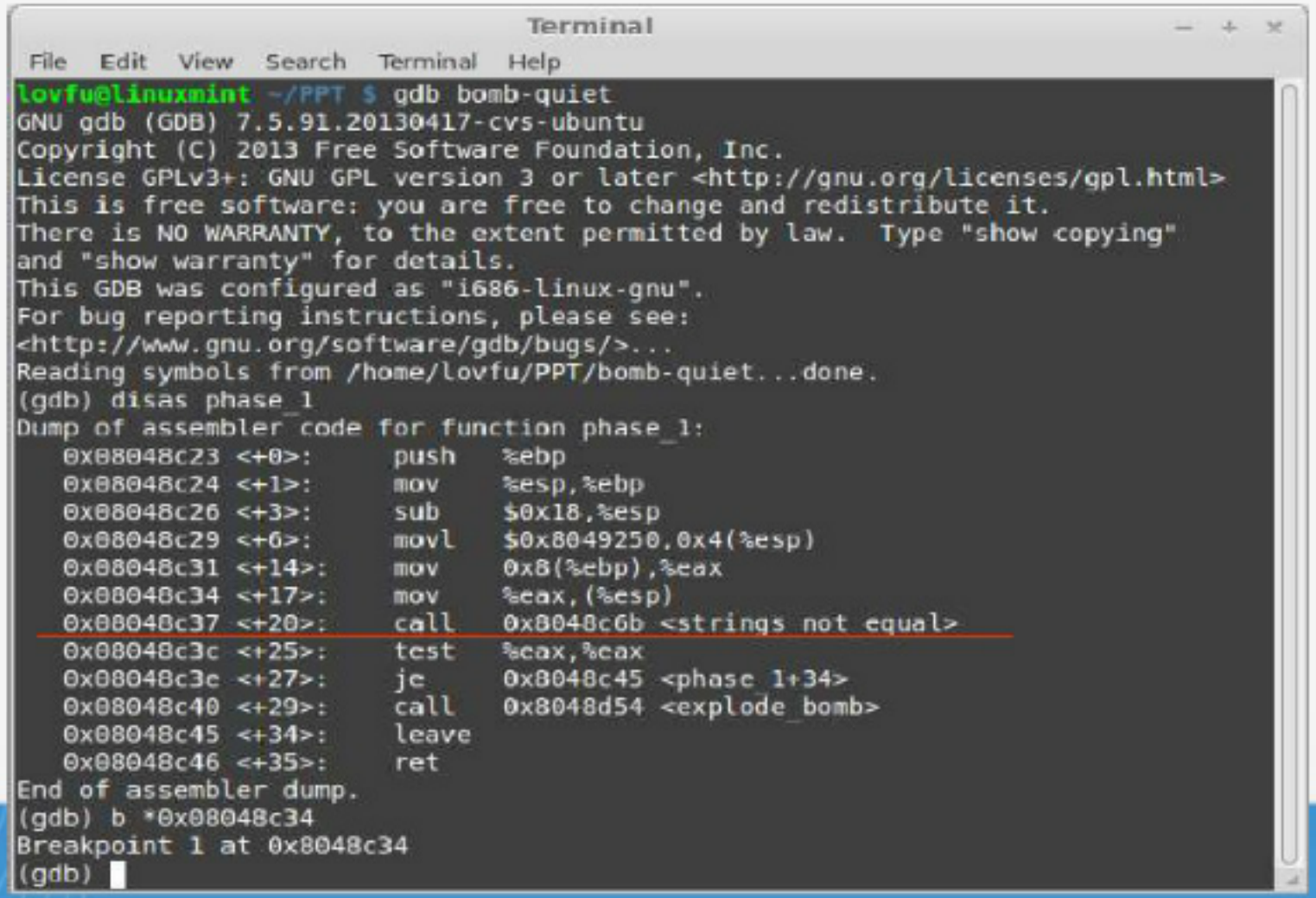
Objdump 基础

- 常用：

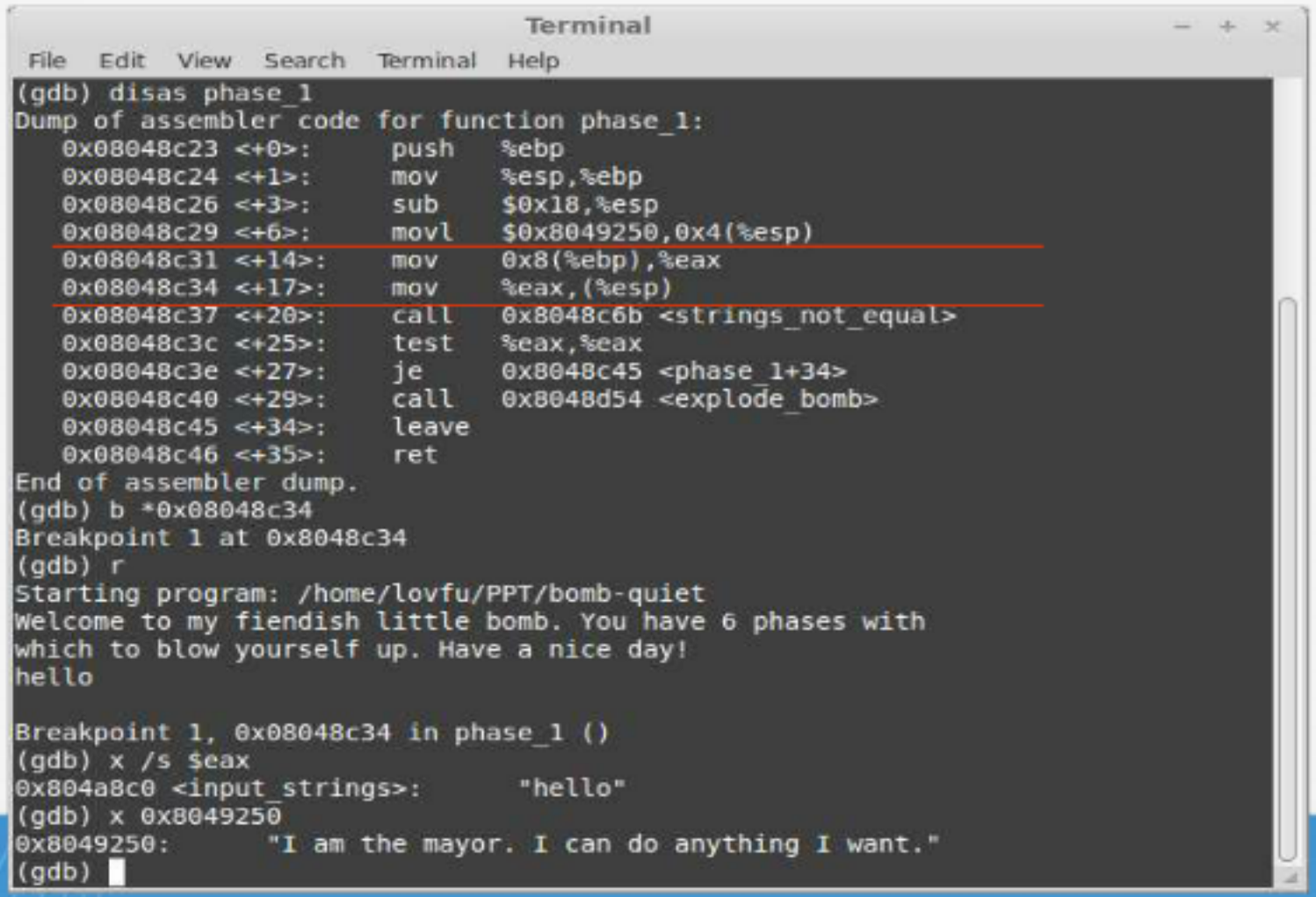
objdump -d bufbomb > bufbomb.s

```
08048dfa <main>:
 8048dfa:   55                      push   %ebp
 8048dfb:   89 e5                   mov    %esp,%ebp
 8048dfd:   83 e4 f0                and    $0xfffffff0,%esp
 8048e00:   57                      push   %edi
 8048e01:   56                      push   %esi
 8048e02:   53                      push   %ebx
 8048e03:   83 ec 24                sub    $0x24,%esp
 8048e06:   8b 75 08                mov    0x8(%ebp),%esi
 8048e09:   8b 5d 0c                mov    0xc(%ebp),%ebx
 8048e0c:   c7 44 24 04 ac 8a 04    movl   $0x8048aac,0x4(%esp)
 8048e13:   08
```

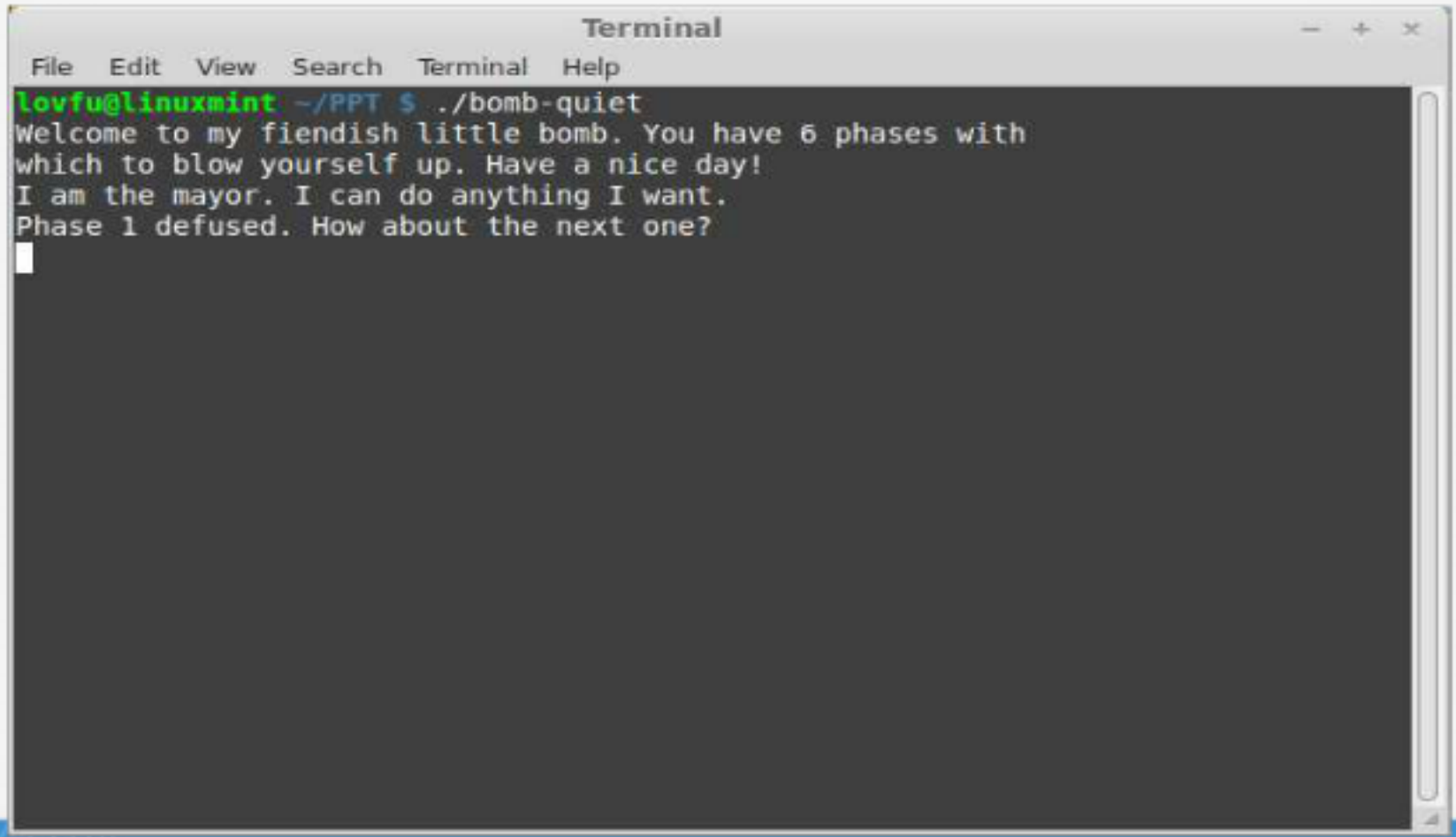
BombLab演示


Terminal
File Edit View Search Terminal Help
lovfu@linuxmint ~/PPT $ gdb bomb-quiet
GNU gdb (GDB) 7.5.91.20130417-cvs-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/lovfu/PPT/bomb-quiet...done.
(gdb) disas phase_1
Dump of assembler code for function phase_1:
0x08048c23 <+0>: push %ebp
0x08048c24 <+1>: mov %esp,%ebp
0x08048c26 <+3>: sub $0x18,%esp
0x08048c29 <+6>: movl $0x8049250,0x4(%esp)
0x08048c31 <+14>: mov 0x8(%ebp),%eax
0x08048c34 <+17>: mov %eax,(%esp)
0x08048c37 <+20>: call 0x8048c6b <strings_not_equal>
0x08048c3c <+25>: test %eax,%eax
0x08048c3e <+27>: je 0x8048c45 <phase_1+34>
0x08048c40 <+29>: call 0x8048d54 <explode_bomb>
0x08048c45 <+34>: leave
0x08048c46 <+35>: ret
End of assembler dump.
(gdb) b *0x08048c34
Breakpoint 1 at 0x8048c34
(gdb)

BombLab演示


Terminal
File Edit View Search Terminal Help
(gdb) disas phase_1
Dump of assembler code for function phase_1:
0x08048c23 <+0>: push %ebp
0x08048c24 <+1>: mov %esp,%ebp
0x08048c26 <+3>: sub $0x18,%esp
0x08048c29 <+6>: movl $0x8049250,0x4(%esp)
0x08048c31 <+14>: mov 0x8(%ebp),%eax
0x08048c34 <+17>: mov %eax,(%esp)
0x08048c37 <+20>: call 0x8048c6b <strings_not_equal>
0x08048c3c <+25>: test %eax,%eax
0x08048c3e <+27>: je 0x8048c45 <phase_1+34>
0x08048c40 <+29>: call 0x8048d54 <explode_bomb>
0x08048c45 <+34>: leave
0x08048c46 <+35>: ret
End of assembler dump.
(gdb) b *0x08048c34
Breakpoint 1 at 0x8048c34
(gdb) r
Starting program: /home/lovfu/PPT/bomb-quiet
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
hello
Breakpoint 1, 0x08048c34 in phase_1 ()
(gdb) x /s $eax
0x804a8c0 <input_strings>: "hello"
(gdb) x 0x8049250
0x8049250: "I am the mayor. I can do anything I want."
(gdb)

BombLab演示


Terminal
File Edit View Search Terminal Help
lovfu@linuxmint ~/PPT $ ./bomb-quiet
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
I am the mayor. I can do anything I want.
Phase 1 defused. How about the next one?

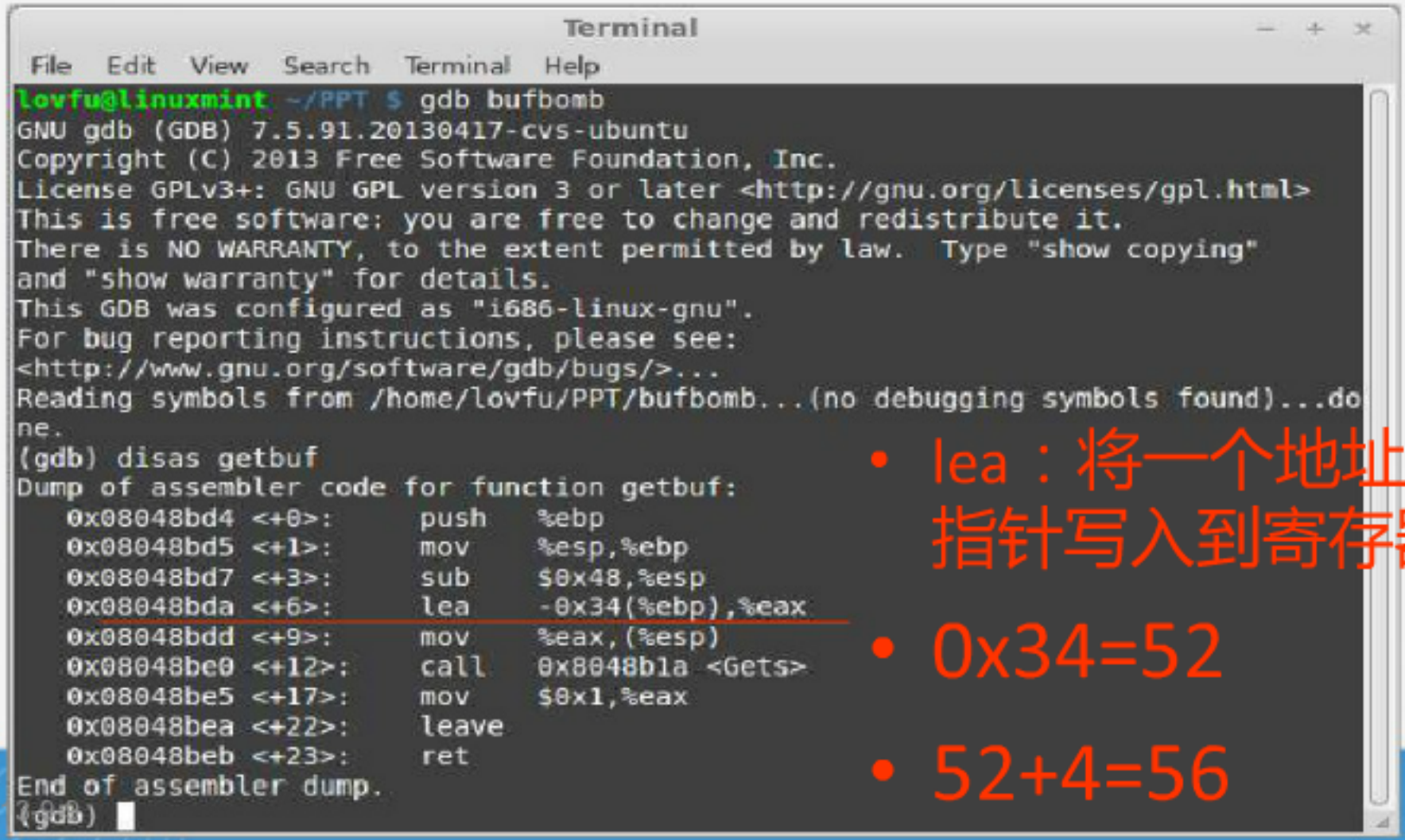
BombLab演示

```
void test()
{
    int val;
    /* Put canary on stack to detect possible corruption */
    volatile int local = uniqueval();

    val = getbuf();

    /* Check for corrupted stack */
    if (local != uniqueval()) {
        printf("Sabotaged!: the stack has been corrupted\n");
    }
    else if (val == cookie) {
        printf("Boom!: getbuf returned 0x%x\n", val);
        validate(3);
    } else {
        printf("Dud: getbuf returned 0x%x\n", val);
    }
}
```


```
int getbuf()
{
    char buf[NORMAL_BUFFER_SIZE];
    Gets(buf);
    return 1;
}
```

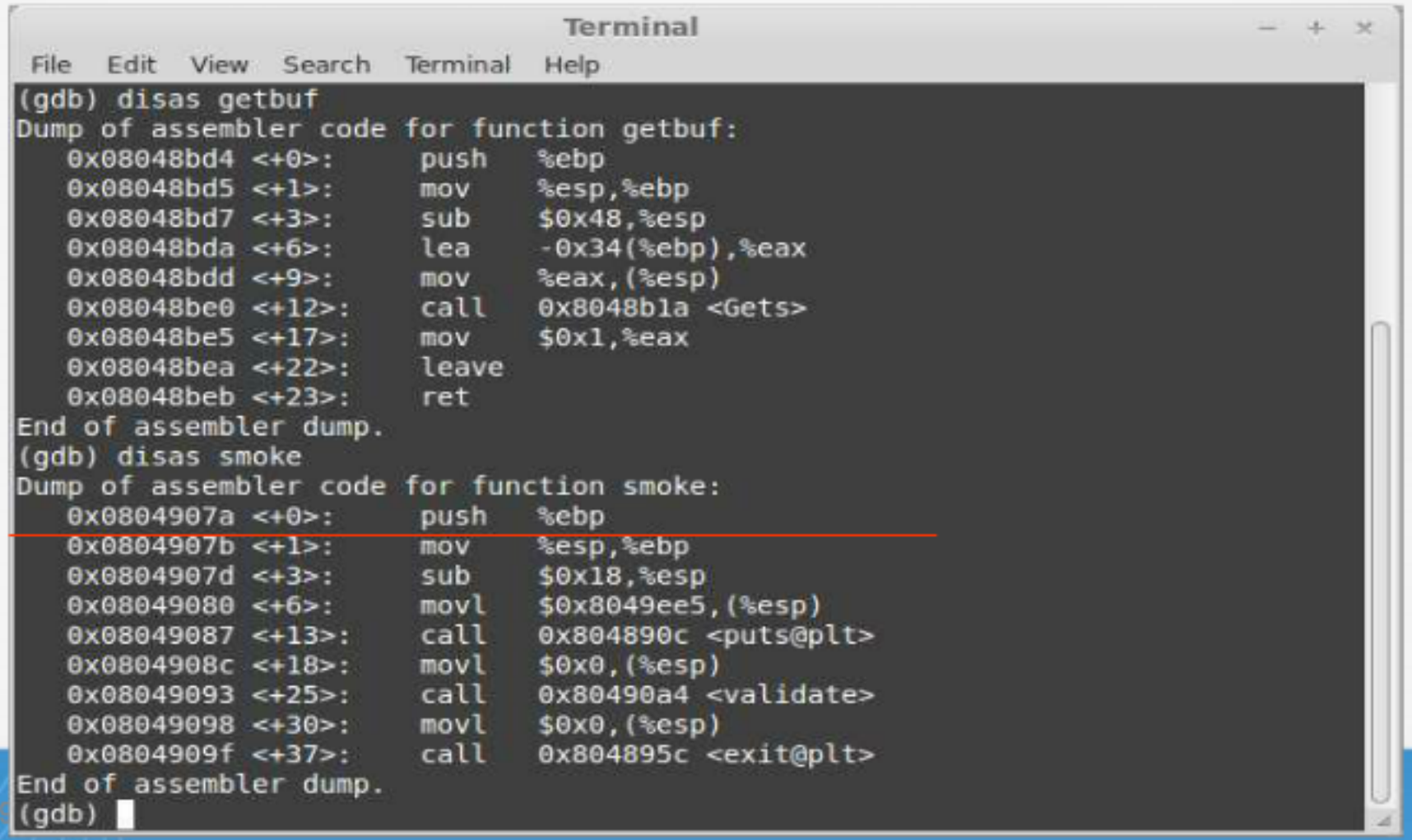
```
 * smoke - On return from getbuf(), the level 0 exploit executes
 * the code for smoke() instead of returning to test().
 */
  $begin smoke-c */
  d smoke()

   printf("Smoke!: You called smoke()\n");
   validate(0);
   exit(0);
}
/* $end smoke-c */
```

BombLab演示



- lea：将一个地址指针写入到寄存器
- 0x34=52
- 52+4=56

BombLab演示


Terminal
File Edit View Search Terminal Help
(gdb) disas getbuf
Dump of assembler code for function getbuf:
0x08048bd4 <+0>: push %ebp
0x08048bd5 <+1>: mov %esp,%ebp
0x08048bd7 <+3>: sub $0x48,%esp
0x08048bda <+6>: lea -0x34(%ebp),%eax
0x08048bdd <+9>: mov %eax,(%esp)
0x08048be0 <+12>: call 0x8048b1a <Gets>
0x08048be5 <+17>: mov $0x1,%eax
0x08048bea <+22>: leave
0x08048beb <+23>: ret
End of assembler dump.
(gdb) disas smoke
Dump of assembler code for function smoke:
0x0804907a <+0>: push %ebp
0x0804907b <+1>: mov %esp,%ebp
0x0804907d <+3>: sub $0x18,%esp
0x08049080 <+6>: movl $0x8049ee5,(%esp)
0x08049087 <+13>: call 0x804890c <puts@plt>
0x0804908c <+18>: movl $0x0,(%esp)
0x08049093 <+25>: call 0x80490a4 <validate>
0x08049098 <+30>: movl $0x0,(%esp)
0x0804909f <+37>: call 0x804895c <exit@plt>
End of assembler dump.
(gdb)

BombLab演示

- 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  00 00 00 00 00 00 00 00 00 00
  7a 90 04 08

Any Question?

- 任何疑惑或发现服务器Bug
- 请联系88wanghongwei@gmail.com或在网络学堂留言

80X86汇编语言与C语言-4

数据表示

- 数组
- 结构
- 联合

基本数据类型

整型

- 如何区分无符号数与符号数？

Intel	GAS	Bytes	C
byte	b	1	[unsigned] char
word	w	2	[unsigned] short
double word	l	4	[unsigned] int
quad word	q	8	[unsigned] long int (x86-64)

浮点数

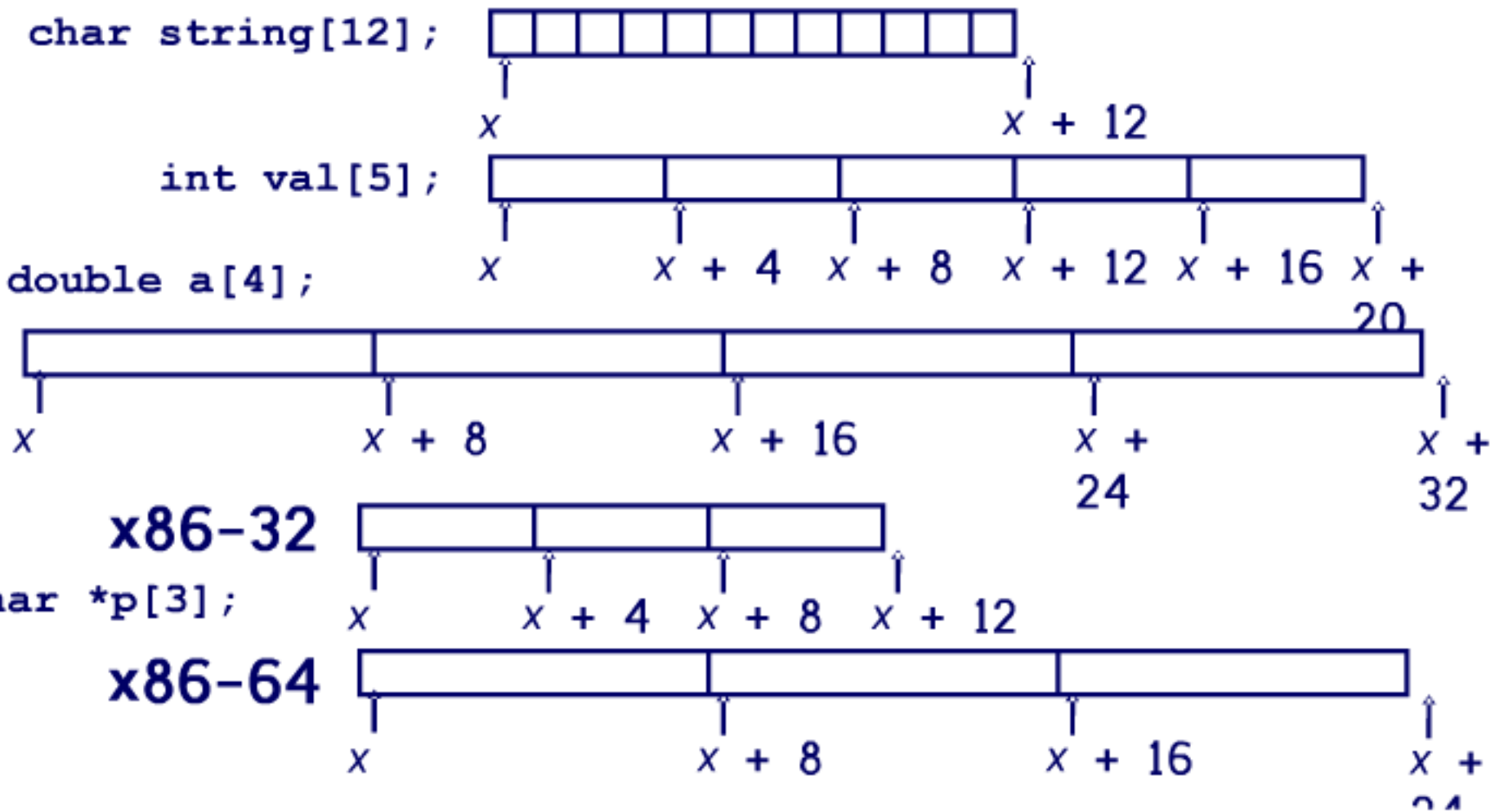
Intel	GAS	Bytes	C
Single	s	4	float
Double	l	8	double
Extended	t	10/12/16	long double

数组的内存存储

基本原则

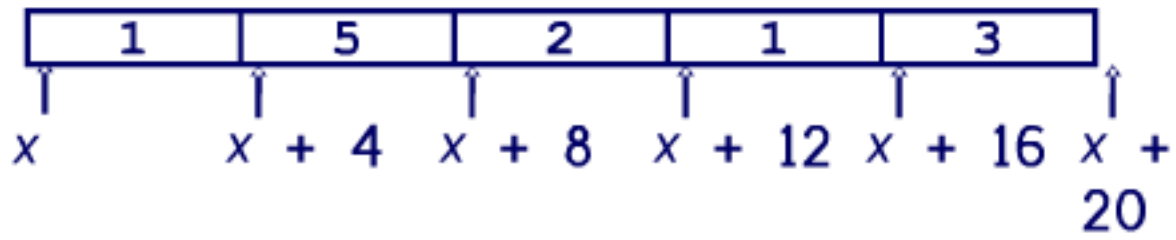
T **A[***L***];**

- 基本数据类型：*T* ；数组长度：*L*
- 连续存储在大小为 *L* * **sizeof(***T***)** 字节的空间内



数组访问

```
int val[5];
```



Reference	Type	Value
`val[4]`	`int`	3
`val`	`int *`	*x*
`val+1`	`int *`	*x* + 4
`&val[2]`	`int *`	*x* + 8
`val[5]`	`int`	??
`*(val+1)`	`int`	5
`val + i`	`int *`	*x* + 4 *i*

```
typedef int zip_dig[5];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

- 寄存器%edx 存储数组的起始地址
- 寄存器%eax 表示下标
- 对应的数组元素的内存地址是%edx + 4*%eax
  - (%edx,%eax,4)

X86-32 下的数组访问代码

```
# %edx = z
# %eax = dig
 movl (%edx,%eax,4),%eax # z[dig]
```

数组循环示例 (X86-32)

```
void zincr(zip_dig z) {
  int i;
  for (i = 0; i < 5; i++)
    z[i]++;
}
```

```
  # edx = z
   movl  $0, %eax              #   %eax = i
.L4:                           # loop:
   addl  $1, (%edx,%eax,4)     #   z[i]++
   addl  $1, %eax              #   i++
   cmpl  $5, %eax              #   i:5
   jne   .L4                   #   if !=, goto loop
```

指针循环示例 (X86-32)

```
void zincr_p(zip_dig z) {
  int *zend = z+5;
  do {
    (*z)++;
    z++;
  } while (z != zend);
}
```

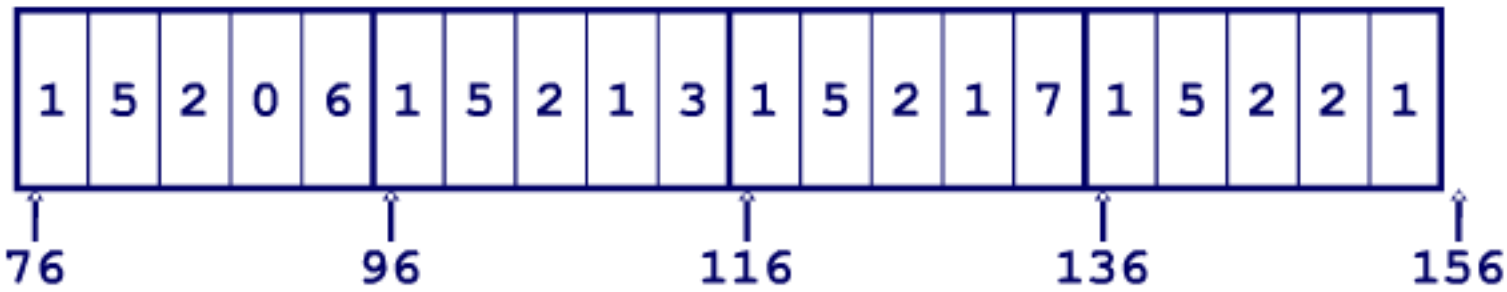
```
void zincr_v(zip_dig z) {
  void *vz = z;
  int i = 0;
  do {
    (*((int *) (vz+i)))++;
    i += ISIZE;/*sizeof(int)*/
  } while (i != ISIZE*5);
}
```

```
   # edx = z = vz
   movl  $0, %eax              #   i = 0
.L8:                           # loop:
   addl  $1, (%edx,%eax)       #   Increment vz+i
   addl  $4, %eax              #   i +=  4
   cmpl  $20, %eax             #   Compare i:20
   jne   .L8                   #   if !=, goto loop
```

嵌套数组

```
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```

zip_dig pgh[4];



访问嵌套数组中的“行”

```
int *get_pgh_zip(int index)
{
  return pgh[index];
}
```

行地址计算

- pgh[index]的数据类型是int [5]，即pgh每个元素的大小是5 * sizeof(int) = 20
- 因此行地址是# pgh + (20 * index)

相关汇编代码

- pgh + 4*(index+4*index)

```
# %eax = index
 leal (%eax,%eax,4),%eax # 5 * index
 leal pgh(,%eax,4),%eax  # pgh + (20 * index)
```

访问嵌套数组的单个元素

- pgh[index][dig] 的地址是:

```
pgh + 20*index + 4*dig
        //sizeof(int)
```

```
int get_pgh_digit
   (int index, int dig)
{
  return pgh[index][dig];
}
```

相关汇编代码

- pgh + 4*dig + 4*(index+4*index)

```
# %ecx = dig
# %eax = index
leal 0(,%ecx,4),%edx          # 4*dig
leal (%eax,%eax,4),%eax       # 5*index
movl pgh(%edx,%eax,4),%eax    # *(pgh + 4*dig + 20*index)
```

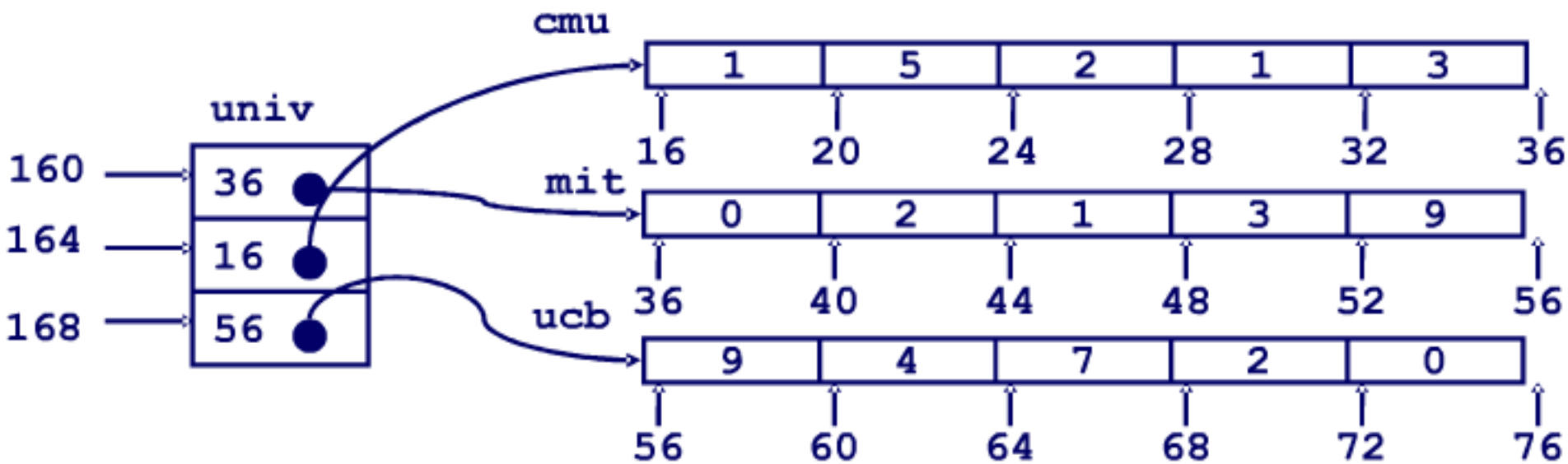
Multi-Level Array

- 变量univ 是一个指针数组，数组长度为3，数组元素长度为4字节

```
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

- 每个指针指向一个整数数组

```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```



访问Multi-Level Array中的元素

```
int get_univ_digit
   (int index, int dig)
{
   return univ[index][dig];
}
```

数组元素的地址计算

- Mem[Mem[univ+4*index]+4*dig]
- 至少进行两次内存读取
  - 首先获得行地址
  - 再访问该行中的元素

```
# %ecx = index
# %eax = dig
leal 0(,%ecx,4),%edx     # 4*index
movl univ(%edx),%edx     # Mem[univ+4*index]
movl (%edx,%eax,4),%eax  # Mem[...+4*dig]
```

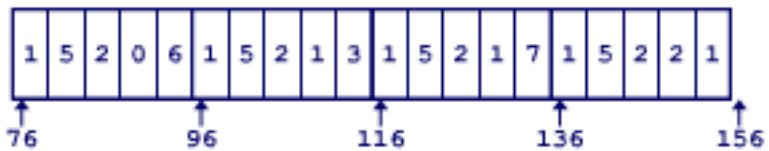
与嵌套数组访问的不同

Nested Array

```
int get_pgh_digit
   (int index, int dig)
{
   return pgh[index][dig];
}
```

- 元素地址

Mem[pgh+20*index+4*dig]

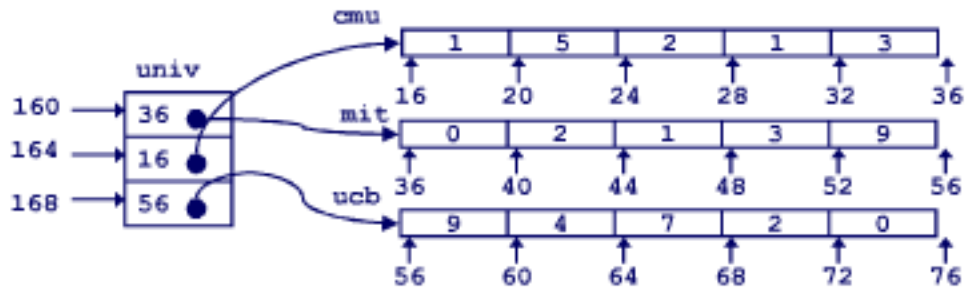


Multi-Level Array

```
int get_univ_digit
   (int index, int dig)
{
   return univ[index][dig];
}
```

- 元素地址

Mem[Mem[univ+4*index]+4*dig]



N X N Matrix Code

Fixed dimensions

- Know value of N at compile time

```
#define N 16
typedef int fix_matrix[N][N];
/* Get element a[i][j] */
int fix_ele
   (fix_matrix a, int i, int j)
{
  return a[i][j];
}
```

Variable dimensions, explicit indexing

- Traditional way to implement dynamic arrays

```
#define IDX(n, i, j) ((i)*(n)+(j))
/* Get element a[i][j] */
int vec_ele
 (int n, int *a, int i, int j)
{
  return a[IDX(n,i,j)];
}
```

Variable dimensions, implicit indexing

- Now supported by gcc

```
/* Get element a[i][j] */
int var_ele
 (int n, int a[n][n], int i, int j) {
  return a[i][j];
}
```

16 X 16 Matrix Access

- **Array Elements**
  - Address **A** + $i * (C * K) + j * K$
  - C = 16, K = 4

```
/* Get element a[i][j] */
int fix_ele(fix_matrix a, int i, int j) {
  return a[i][j];
}
```

```
   movl   12(%ebp), %edx     # i
   sall   $6, %edx           # i*64
   movl   16(%ebp), %eax     # j
   sall   $2, %eax           # j*4
   addl   8(%ebp), %eax      # a + j*4
   movl   (%eax,%edx), %eax  # *(a + j*4 + i*64)
```

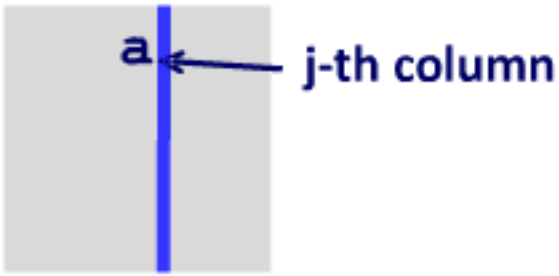
n X n Matrix Access

- **Array Elements**
  - Address **A** + $i * (C * K) + j * K$
  - C = n, K = 4

```
/* Get element a[i][j] */
int var_ele(int n, int a[n][n], int i, int j) {
  return a[i][j];
}
```

```
  movl   8(%ebp), %eax      # n
  sall   $2, %eax           # n*4
  movl   %eax, %edx         # n*4
  imull  16(%ebp), %edx     # i*n*4
  movl   20(%ebp), %eax     # j
  sall   $2, %eax           # j*4
  addl   12(%ebp), %eax     # a + j*4
  movl   (%eax,%edx), %eax  # *(a + j*4 + i*n*4)
```

Optimizing Fixed Array Access




```
#define N 16
typedef int fix_matrix[N][N];
```

Computation

- Step through all elements in column j

Optimization

- Retrieving successive elements from single column

```
/* Retrieve column j from array */
void fix_column
  (fix_matrix a, int j, int *dest)
{
  int i;
  for (i = 0; i < N; i++)
    dest[i] = a[i][j];
}
```

Optimization

- Compute ajp = &a[i][j]
  - Initially = a + 4*j
  - Increment by 4*N

Register	Value
%ecx	ajp
%ebx	dest
%edx	i

```
/* Retrieve column j from array */
void fix_column
  (fix_matrix a, int j, int *dest)
{
  int i;
  for (i = 0; i < N; i++)
    dest[i] = a[i][j];
}
```

```
.L8:                              # loop:
   movl  (%ecx), %eax             #   Read *ajp
   movl  %eax, (%ebx,%edx,4)      #   Save in dest[i]
   addl  $1, %edx                 #   i++
   addl  $64, %ecx                #   ajp += 4*N
   cmpl  $16, %edx                #   i:N
   jne   .L8                      #   if !=, goto loop
```

Optimizing Variable Array Access

- Compute ajp = &a[i][j]
  - Initially = a + 4*j
  - Increment by 4*n

Register	Value
%ecx	ajp
%edi	dest
%edx	i
%ebx	4*n
%esi	n

```
/* Retrieve column j from array */
void var_column
   (int n, int a[n][n],
    int j, int *dest)
{
   int i;
   for (i = 0; i < n; i++)
     dest[i] = a[i][j];
}
```

```
.L18:                              # loop:
   movl   (%ecx), %eax             #   Read *ajp
   movl   %eax, (%edi,%edx,4)      #   Save in dest[i]
   addl   $1, %edx                 #   i++
   addl   $ebx, %ecx               #   ajp += 4*n
   cmpl   $edx, %esi               #   n:i
   jg     .L18                     #   if >, goto loop
```

```
_function:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %edi
        xorl    %edi, %edi
        pushl   %esi
        xorl    %esi, %esi
        pushl   %ebx
        xorl    %ebx, %ebx
L11:
        xorl    %ecx, %ecx
        leal    _N(%edi), %edx
        jmp     L10
L18:
        movl    (%edx), %eax
        incl    %ecx
        addl    $4, %edx
        addl    %eax, %ebx
        cmpl    $9, %ecx
        jg      L17
L10:
        leal    (%esi,%ecx), %eax
        testb   $1, %al
        jne     L18
        movl    (%edx), %eax
        incl    %ecx
        addl    $4, %edx
        subl    %eax, %ebx
        cmpl    $9, %ecx
        jle     L10
```

```
L17:
        incl    %esi
        addl    $40, %edi
        cmpl    $9, %esi
        jle     L11
        movl    %ebx, %eax

        popl    %ebx
        popl    %esi
        popl    %edi
        popl    %ebp

        ret

.comm   _N, 400   # 400
```

```
int N[10][10];

//初始化数组省略!
int function(int n)
{
    int res = 0;
    int i ,j;
    for( i = 0; i < 10; i++)
    {
        //????
    }
    return res;
}
```

练习题

```
int  N[10][10];

int  function(int  n)
{
     int  res  =  0;
     int  i  ,j;
     for(  i  =  0;  i  <  10;  i++)
     for(  j  =  0;  j  <  10;  j++)
     {
          if((i  +  j  )  %  2)
                    res  =  res  +  N[i][j];
          else
                    res  =  res  -  N[i][j];
     }
     return  res;
}
```

结构

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```

存储布局(memory-layout)

连续分配的内存区域;
内部元素通过名字访问;
且元素可以是不同的数据类型。

访问结构中的元素

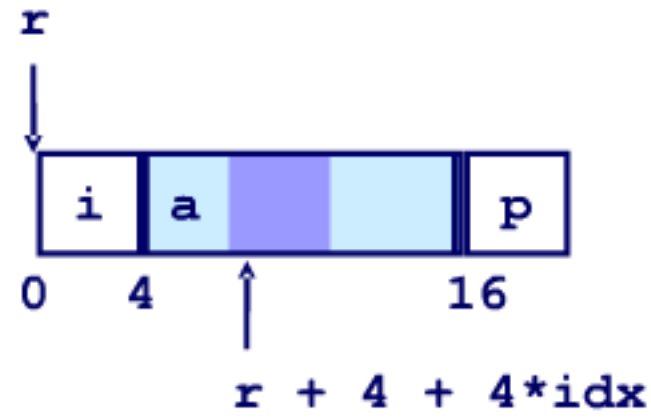
```
void
set_i(struct rec *r,
      int val)
{
  r->i = val;
}
```

相应的汇编代码

```
# %eax = val
# %edx = r
movl %eax,(%edx)    # Mem[r] = val
```

结构中元素的地址计算

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```



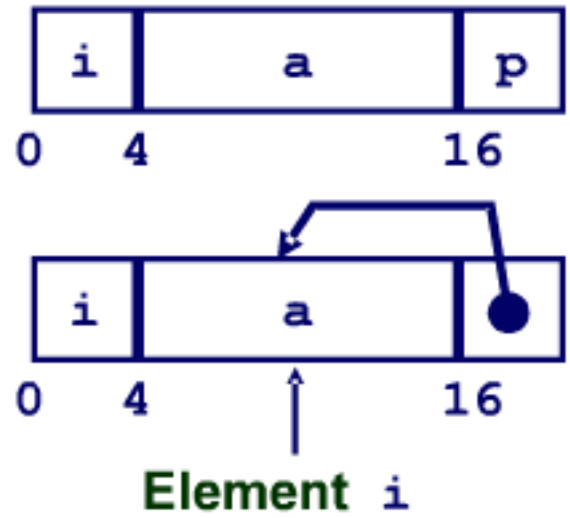
- 每个元素在结构中的相对地址在编译时就已确定

```
int *
find_a
 (struct rec *r, int idx)
{
  return &r->a[idx];
}
```

```
# %ecx = idx
# %edx = r
leal 0(,%ecx,4),%eax    # 4*idx
leal 4(%eax,%edx),%eax  # r+4*idx+4
```

续前

```
struct rec {
   int i;
   int a[3];
   int *p;
};
```




```
void
set_p(struct rec *r)
{
   r->p =
     &r->a[r->i];
}
```

```
# %edx = r
movl (%edx),%ecx          # r->i
leal 0(,%ecx,4),%eax      # 4*(r->i)
leal 4(%edx,%eax),%eax    # r+4+4*(r->i)
movl %eax,16(%edx)        # Update r->p
```

数据存储位置对齐

对齐的一般原则

- 已知某种基本数据类型的大小为 K 字节
- 那么，其存储地址必须是K的整数倍
- X86-32的对齐要求基本上就是这样
  - 但是64位系统、linux与windows系统的要求都略有不同

为何需要对齐

- 计算机访问内存一般是以内存块为单位的，块的大小是地址对齐的，如4、8、16字节对齐等
  - 如果数据访问地址跨越“块”边界会引起额外的内存访问

编译器的工作

- 在结构的各个元素间插入额外空间来满足不同元素的对齐要求

X86-32下不同元素的对齐要求

基本数据类型:

- 1 byte (e.g., char)
  - 无要求
- 2 bytes (e.g., short)
  - 2字节对齐。地址最后一位为0
- 4 bytes (e.g., int, float, char *, etc.)
  - 地址最后两位为0
- 8 bytes (e.g., double)
  - Windows 系统(以及大多数操作系统):
    - » 地址最后三位为0
  - Linux:
    - » 地址最后两位为0
    - » 即4对齐
- 12 bytes (long double)
  - Windows, Linux:
    - » 地址最后两位为0
    - » 即4对齐

X86-64下不同元素的对齐要求

基本数据类型:

- 1 byte (e.g., char)
  - 同x86-32
- 2 bytes (e.g., short)
  - 同x86-32
- 4 bytes (e.g., int, float)
  - 同x86-32
- 8 bytes (e.g., double, char *)
  - Windows & Linux:
    - » 地址最后三位为0
- 16 bytes (long double)
  - Linux:
    - » 地址最后三位为0
    - » 即8对齐

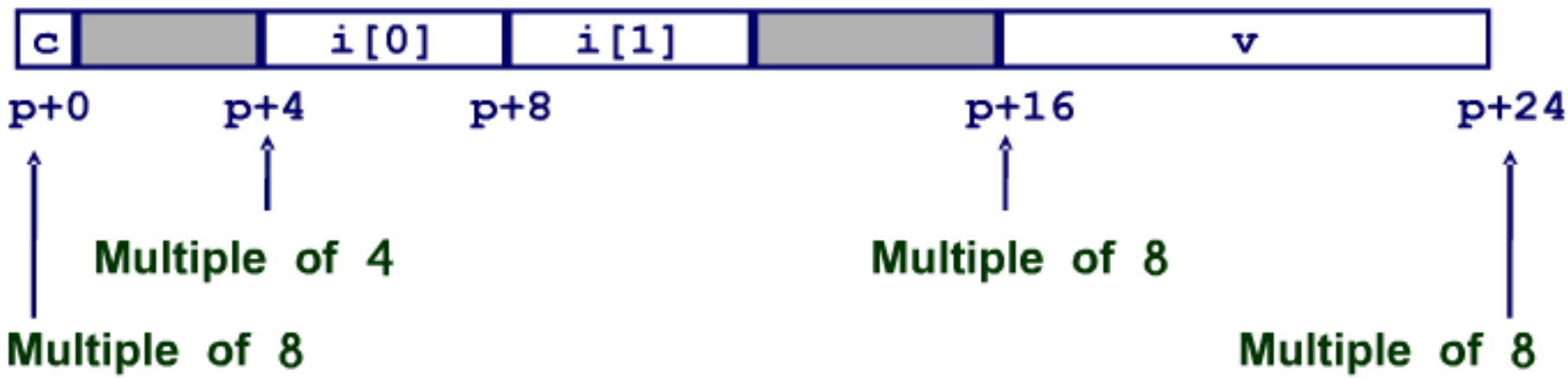
结构的存储对齐要求

- 必须满足结构中各个元素的对齐要求
- 结构自身的对齐要求等同于其各个元素中对齐要求最高的那个，设为K字节
- 结构的起始地址与结构长度必须是K的整数倍

示例（Windows 系统或者 x86-64系统下）：

- K = 8（why?）

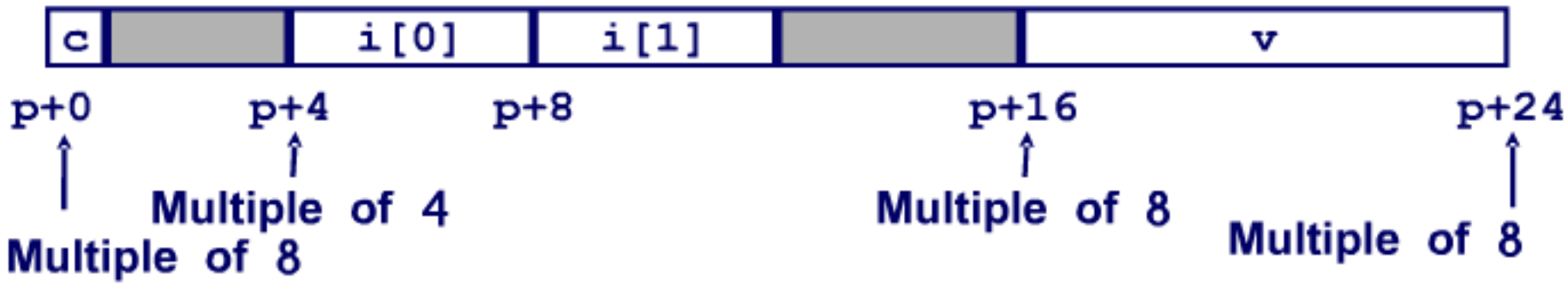
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

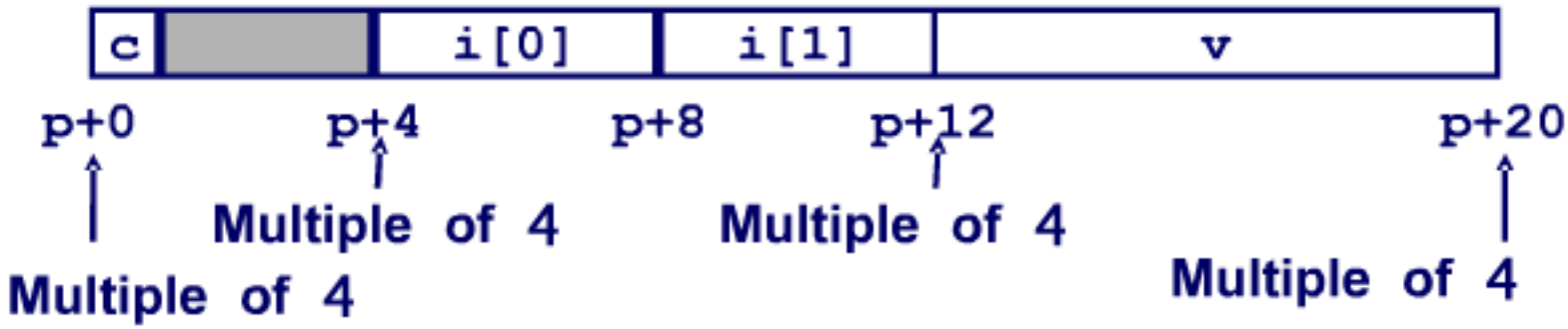
Windows 系统或者 x86-64系统下:

- K = 8, （double 类型是8对齐）



X86-32的Linux系统下:

- K = 4



结构自身的对齐要求

```
struct S2 {
  double x;
  int i[2];
  char c;
} *p;
```

p 必须是8或者4字节的整数倍:
8 : x86-64 或x86-32 Windows
4 : x86-32 Linux

p+0 p+8 p+12 p+16 Windows: p+24
Linux: p+20

```
struct S3 {
  float x[2];
  int i[2];
  char c;
} *p;
```

p必须是4字节的整数倍(无论何种系统)

p+0 p+4 p+8 p+12 p+16 p+20

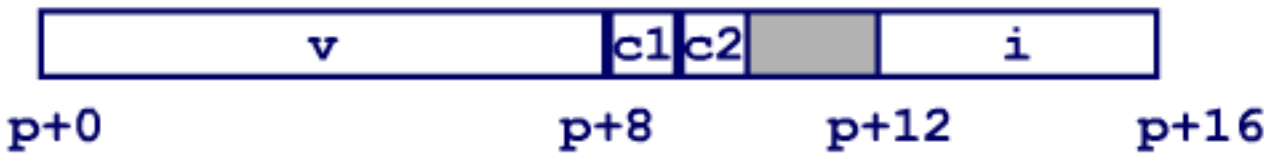
结构内元素不同的先后顺序...

```
struct S4 {
   char c1;
   double v;
   char c2;
   int i;
} *p;
```

在Windows或者 x86-64系统下，共有几个字节被浪费？


```
struct S5 {
   double v;
   char c1;
   char c2;
   int i;
} *p;
```

变换顺序后呢？



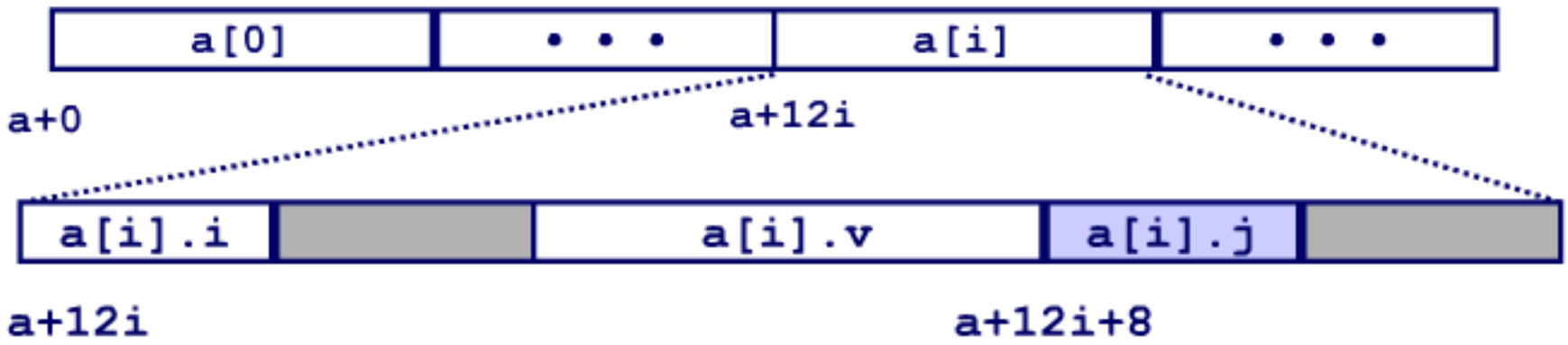
结构数组

- 首先计算数组元素（即结构）的地址
  - $12*i = 4*(i+2i)$
- 然后访问该结构中的元素
  - 偏移量为8

```
struct S6 {
  short i;
  float v;
  short j;
} a[10];
```

```
short get_j(int idx)
{
  return a[idx].j;
}
```

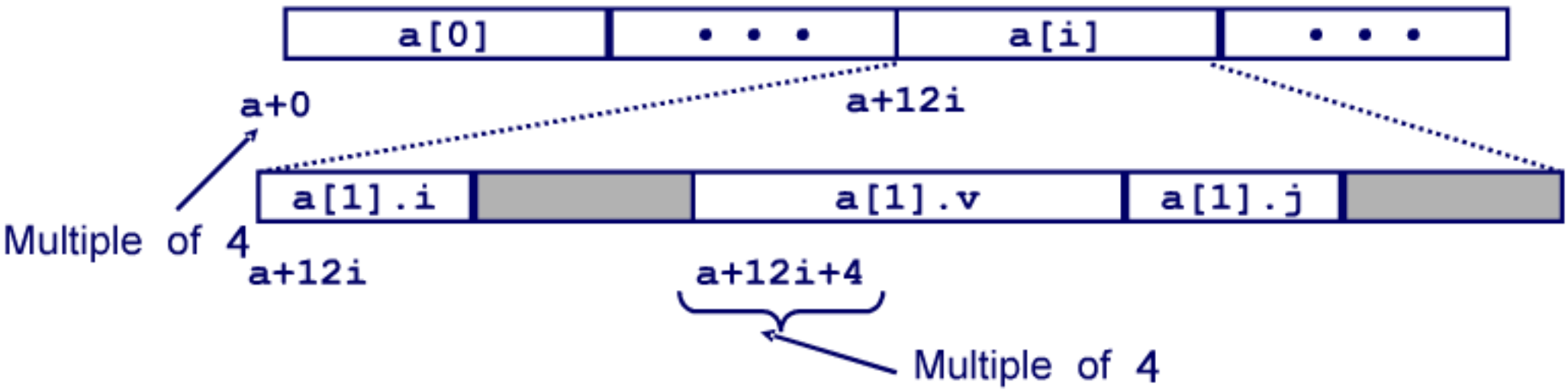
```
# %eax = idx
leal (%eax,%eax,2),%eax # 3*idx
movswl a_start_addr+8(,%eax,4),%eax
```



结构的存储对齐要求小结

- 结构起始地址的对齐要求等同于该结构各个元素中对齐要求最高的那个
  - a中每个元素的地址是4的整数倍
- 结构中元素的对齐要求必须满足该元素自身的对齐要求
  - v必须是4对齐
- 结构的长度必须是该结构各个元素中对齐要求最高的那个元素长度的整数倍

```
struct S6 {
   short i;
   float v;
   short j;
} a[10];
```

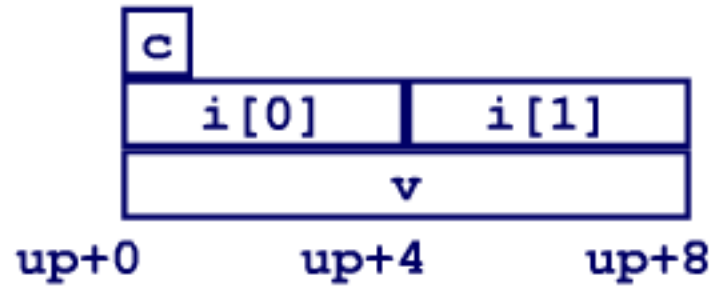


联合

union中可以定义多个成员，union的大小由最大的成员的大小决定。

union成员共享同一块大小的内存，一次只能使用其中的一个成员

```
union U1 {
  char c;
  int i[2];
  double v;
} *up;
```




```
struct S1 {
  char c;
  int i[2];
  double v;
} *sp;
```

(Windows alignment)

小结

C语言数组的汇编访问

- 连续存储
- 访问代码优化
- 无边界检查

结构

- 对齐要求
- 以及相应的汇编代码

联合

练习题

C函数返回struct类型是如何实现的？

```
typedef struct{
int age; int bye; int coo; int ddd; int eee;
} TEST_Struct;

int i = 2;

TEST_Struct __cdecl return_struct(int n )
{
      TEST_Struct local_struct;
      local_struct.age = n;
      local_struct.bye = n;
      local_struct.coo = 2*n;
      local_struct.ddd = n;
      local_struct.eee = n;

      i = local_struct.eee + local_struct.age
      *2 ;

      return local_struct;
}

int function1()
{
      TEST_Struct main_struct =
            return_struct(i);

      return 0;
}
```

```
_return_struct:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx

        subl    $36, %esp

        movl    12(%ebp), %ecx
        movl    8(%ebp), %eax
        leal    (%ecx,%ecx), %ebx
        leal    (%ecx,%ebx), %edx
        movl    %edx, _i
        movl    %ecx, (%eax)
        movl    %ecx, 4(%eax)
        movl    %ebx, 8(%eax)
        movl    %ecx, 12(%eax)
        movl    %ecx, 16(%eax)

        addl    $36, %esp

        popl    %ebx
        popl    %ebp
        ret     $4
```

```
_ function1:
        pushl   %ebp
        movl    %esp, %ebp

        subl    $56, %esp
        movl    _i, %eax
        leal    -40(%ebp), %edx
        movl    %edx, (%esp)
        movl    %eax, 4(%esp)
        call    _return_struct
        subl    $4, %esp

        xorl    %eax, %eax
        leave
        ret
```

C函数是如何传入struct类型参数的？

```
typedef struct{
int age; int bye; int coo; int ddd; int eee;
} TEST_Struct;

int i = 2;
int input_struct(TEST_Struct in_struct)
{
      return in_struct.eee + in_struct.age*2 ;
}
int function2()
{
      TEST_Struct main_struct;
      main_struct.age = i;
      main_struct.bye = i;
      main_struct.coo = 2*i;
      main_struct.ddd = i;
      main_struct.eee = I;
      return input_struct(main_struct);
}
```

```
_input_struct:
        pushl           %ebp
        movl            %esp, %ebp
        movl            8(%ebp), %eax
        movl            24(%ebp), %edx
        popl            %ebp
        leal            (%edx,%eax,2), %eax
        ret

_function2:
        pushl           %ebp
        movl            %esp, %ebp
        subl            $60, %esp
        movl            _i, %eax
        leal            (%eax,%eax), %edx
        movl            %eax, (%esp)
        movl            %eax, 4(%esp)
        movl            %edx, 8(%esp)
        movl            %eax, 12(%esp)
        movl            %eax, 16(%esp)
        call            _input_struct
        leave
        ret
```