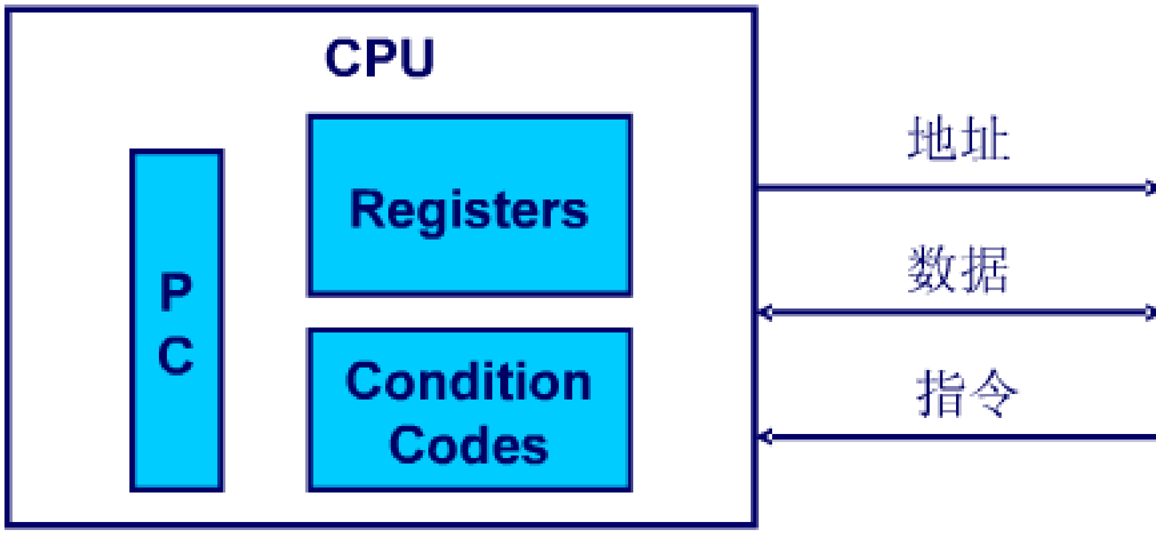
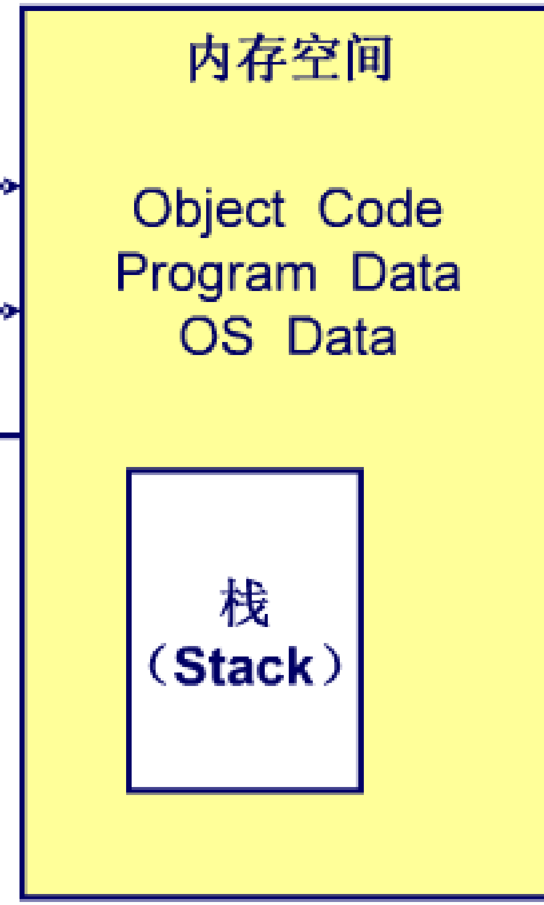
80X86汇编语言与C语言-1

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
```

类似于表达式:

```
x += y
```

或者

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

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

```
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

`%eax`
`%edx`
`%ecx`
`%ebx`
`%esi`
`%edi`
`%esp`
`%ebp`

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

```c
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

```c
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
```

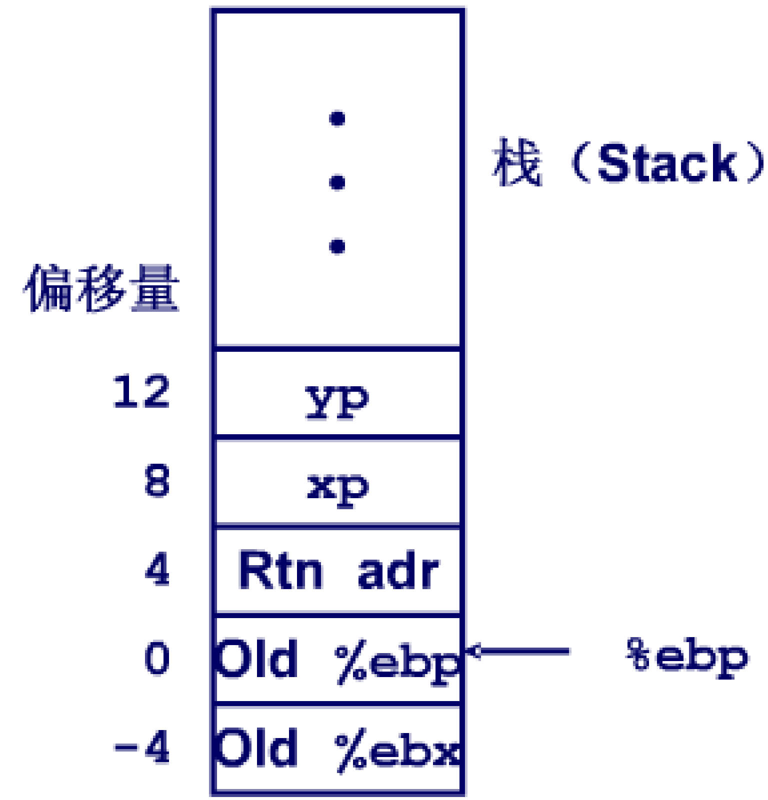
Set Up

Body

Finish

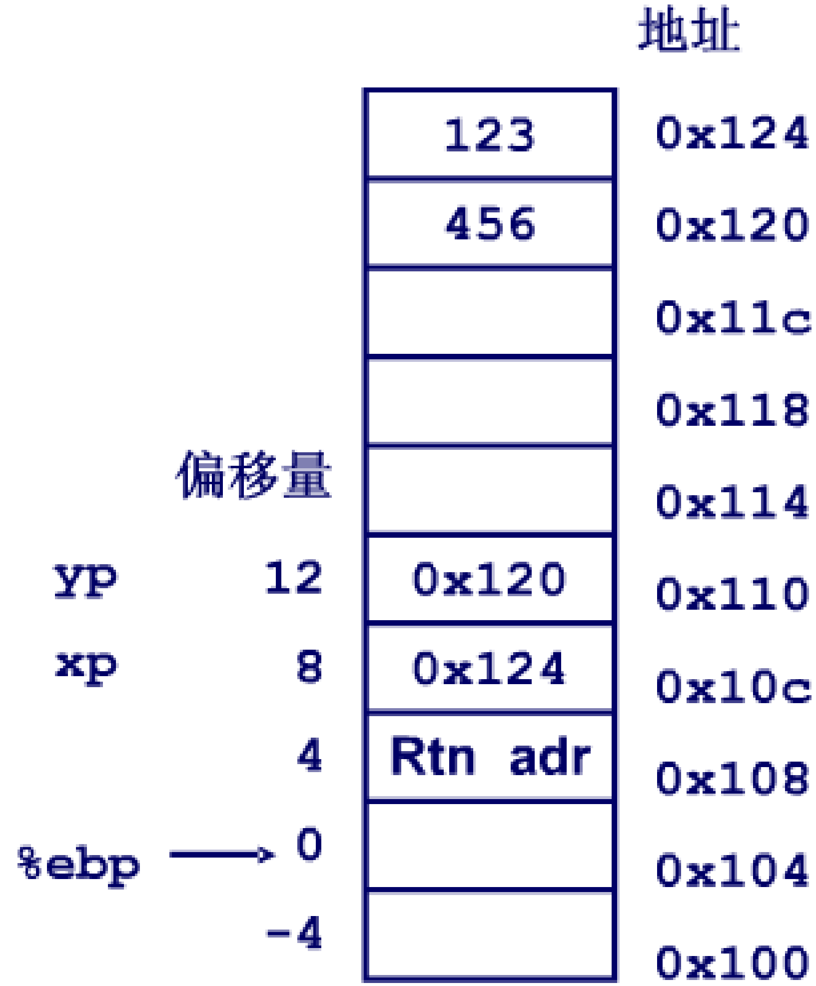
实例分析

```
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
```

%eax	
%edx	
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

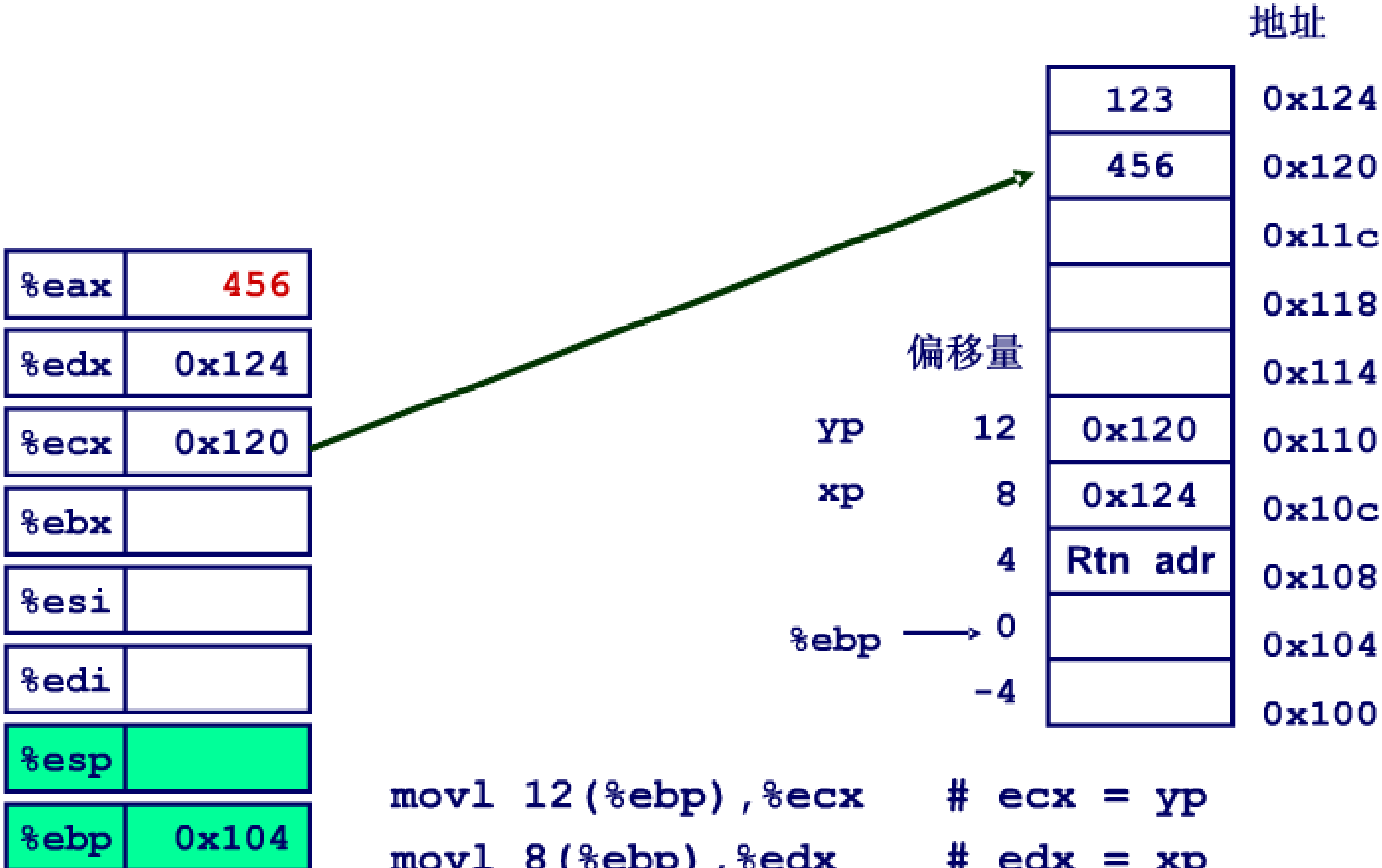
	偏移量		地址
		123	0x124
		456	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp →	0		0x104
	-4		0x100

```
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
```

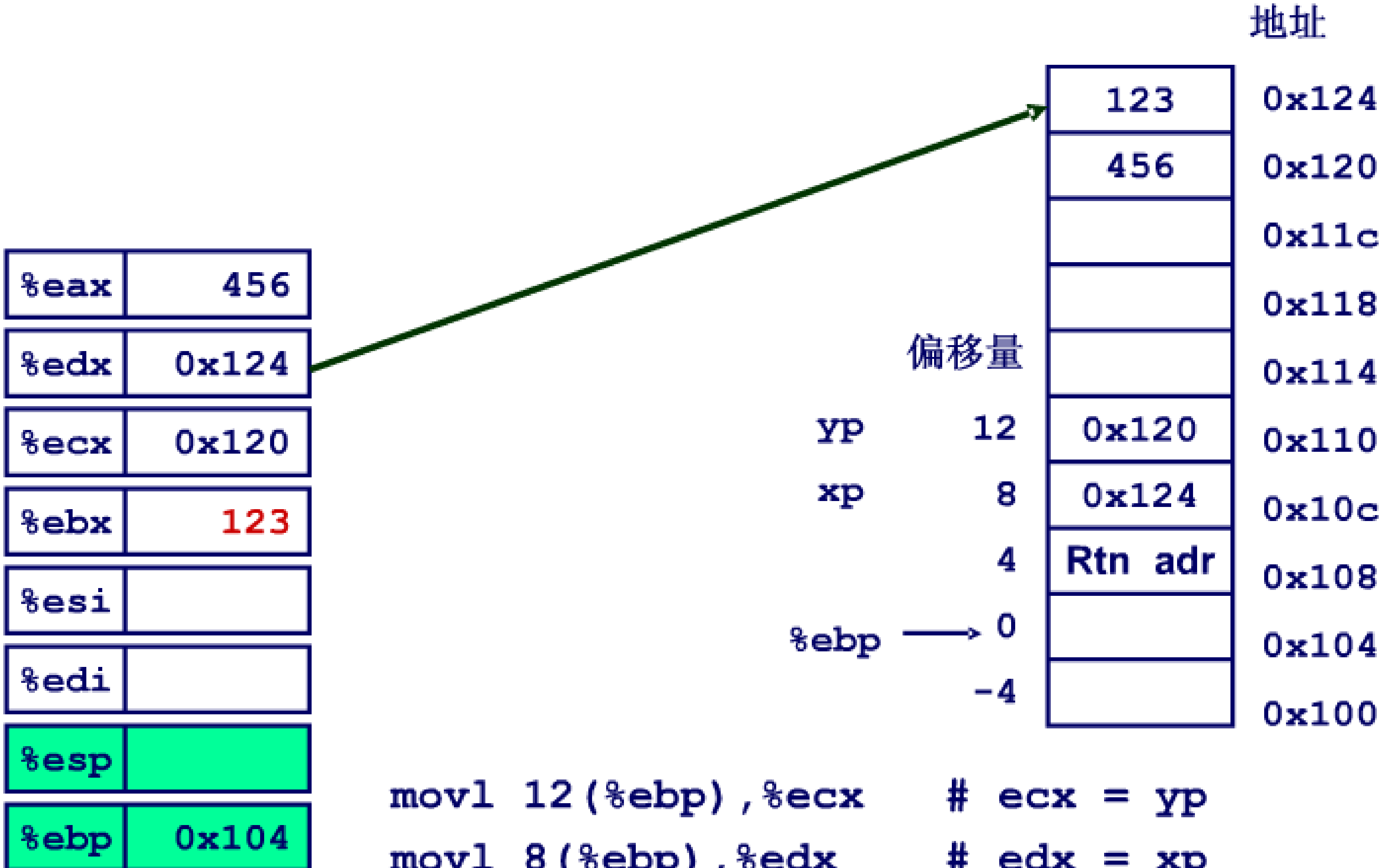
%eax	
%edx	0x124
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

	偏移量		地址
		123	0x124
		456	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp →	0		0x104
	-4		0x100

```
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
```

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

	偏移量		地址
		456	0x124
		456	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp →	0		0x104
	-4		0x100

```
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
```

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

	偏移量		地址
		456	0x124
		123	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp →	0		0x104
	-4		0x100

```
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	$D \leftarrow S$	传送
movb		传送字节	
movw		传送字	
movl		传送双字	
MOVS	S, D	$D \leftarrow$ 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	$D \leftarrow$ 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*

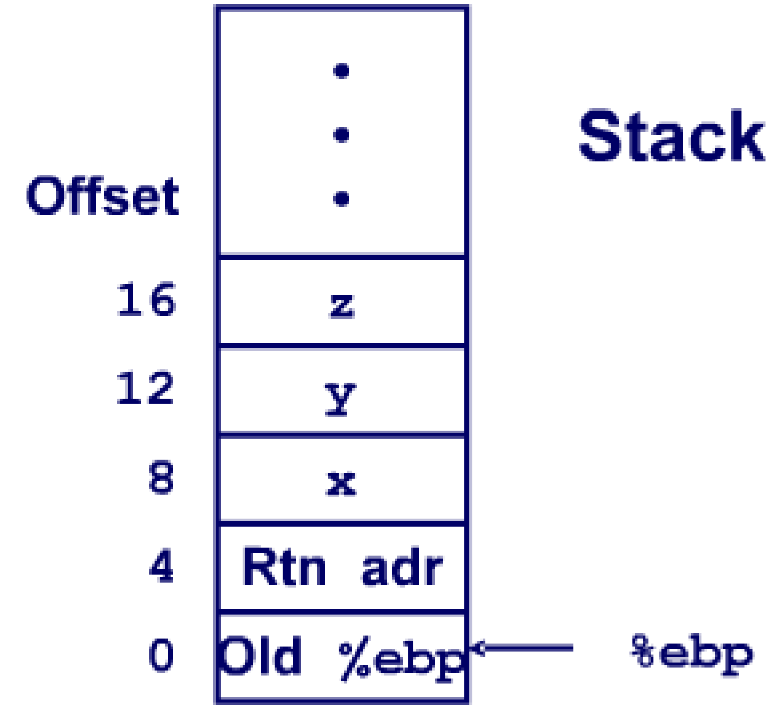
指令格式	计算
单操作数指令	
`incl` *Dest*	*Dest* = *Dest* + 1
`decl` *Dest*	*Dest* = *Dest* - 1
`negl` *Dest*	*Dest* = - *Dest*
`notl` *Dest*	*Dest* = ~ *Dest*

将leal指令用于计算（实例1）

```c
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;
}
```

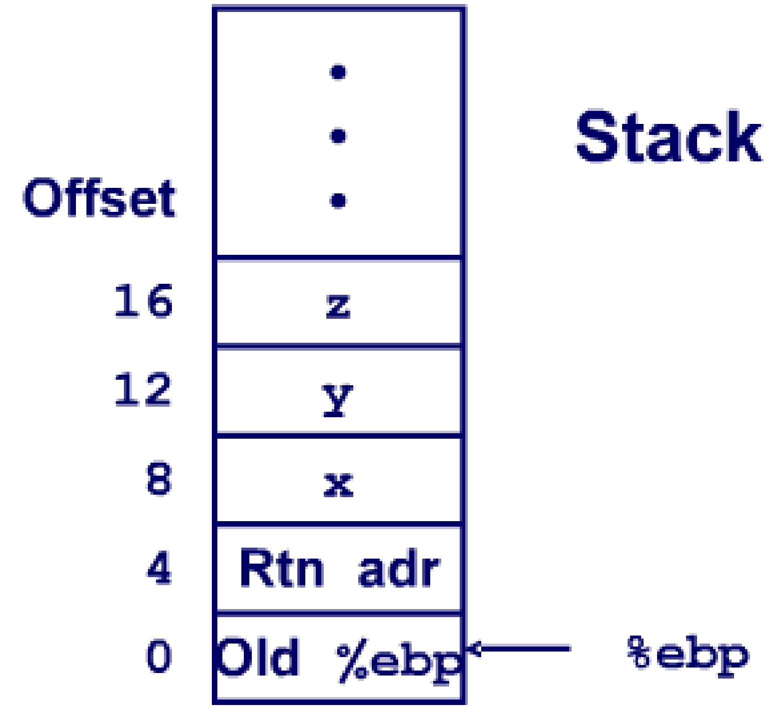
```
arith:
    pushl %ebp                      } Set
    movl %esp,%ebp                  } Up

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

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

```
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;
}
```



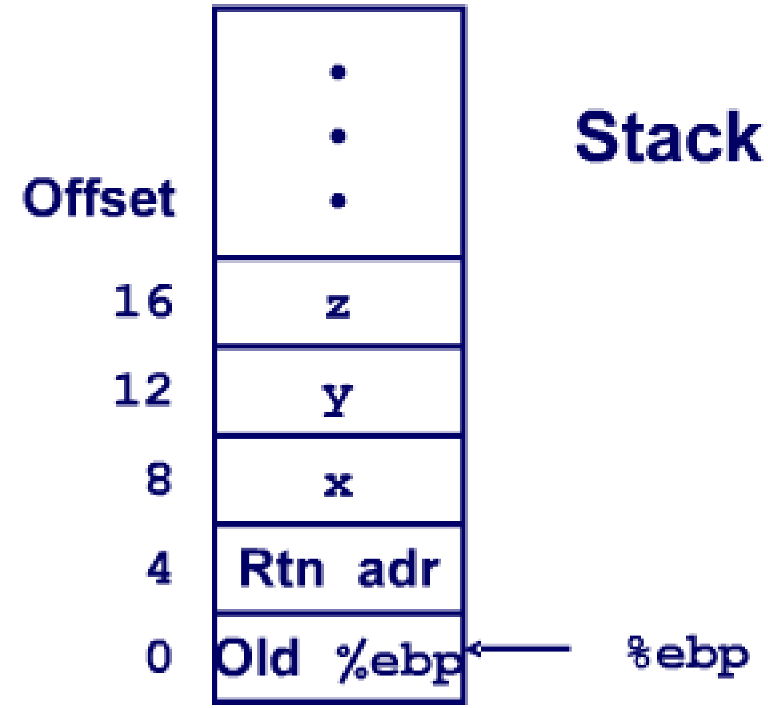

```
movl 8(%ebp),%eax          # eax = x
movl 12(%ebp),%edx         # edx = y
leal (%edx,%eax),%ecx      # ecx = x+y   (t1)
leal (%edx,%edx,2),%edx    # edx = 3*y
sall $4,%edx               # edx = 48*y (t4)
addl 16(%ebp),%ecx         # ecx = z+t1 (t2)
leal 4(%edx,%eax),%eax     # eax = 4+t4+x (t5)
imull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

```c
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;
}
```



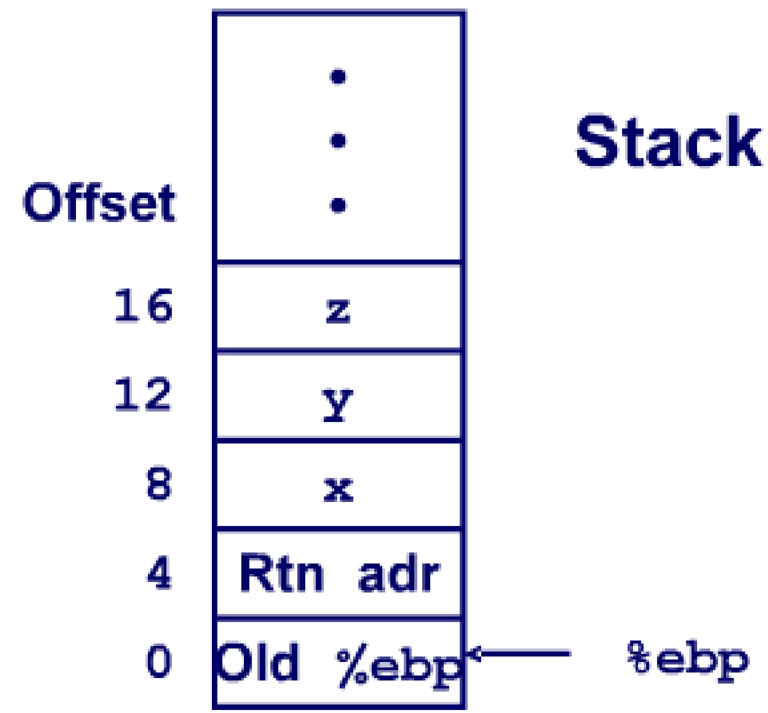

```
movl 8(%ebp),%eax        # eax = x
movl 12(%ebp),%edx       # edx = y
leal (%edx,%eax),%ecx    # ecx = x+y   (t1)
leal (%edx,%edx,2),%edx  # edx = 3*y
sall $4,%edx             # edx = 48*y (t4)
addl 16(%ebp),%ecx       # ecx = z+t1 (t2)
leal 4(%edx,%eax),%eax   # eax = 4+t4+x (t5)
imull %ecx,%eax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

```c
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)
```

```c
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)
```

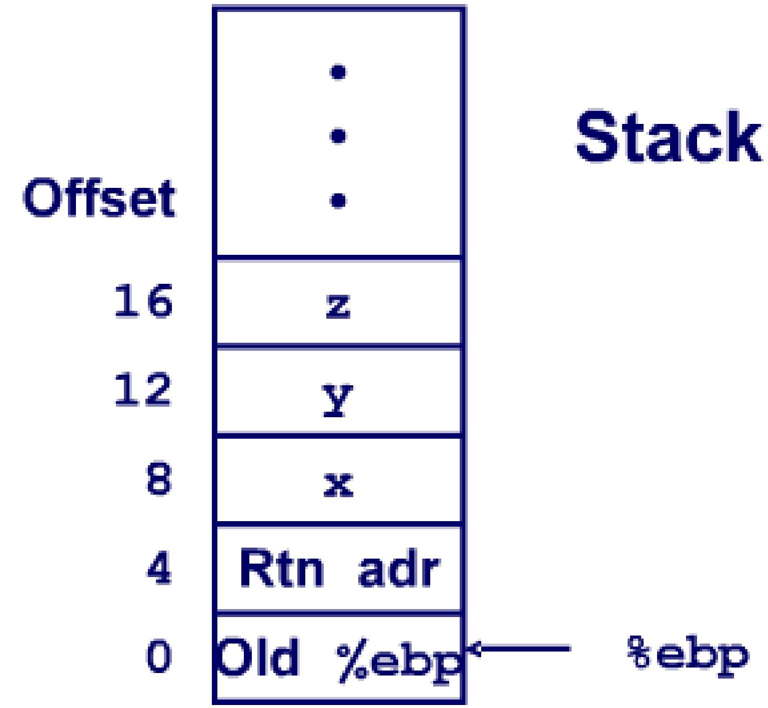
```c
int arith
  (int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```

Stack

Offset	
	• • •
16	z
12	y
8	x
4	Rtn adr
0	Old %ebp ← %ebp

```
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)
```

```c
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            } Set
    movl %esp,%ebp        } Up

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

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

```
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
```

```c
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              } Set Up
    movl %esp,%ebp

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

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

```
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
```

```c
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              } Set
    movl %esp,%ebp          } Up

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

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

```
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
```

```c
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              } Set
    movl %esp,%ebp          } Up

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

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

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

```
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

%rax	%eax	%r8	%r8d
%rdx	%edx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rbx	%ebx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

- 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

```c
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...

```c
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
    movl    16(%ebp), %edx
    subl    12(%ebp), %edx
    movl    %edx, %eax
    sall    $15, %eax
    sarl    $15, %eax
    xorl    8(%ebp), %edx
    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;
}
```