第三章 程序的转换与机器级表示

程序转换概述

x86-64指令系统

C语言程序的机器级表示

复杂数据类型的分配和访问

越界访问和缓冲区溢出

浮点指令和代码

程序的转换与机器级表示

- 主要教学目标
  - 了解高级语言与汇编语言、汇编语言与机器语言之间的关系
  - 掌握有关指令格式、操作数类型、寻址方式、操作类型等内容
  - 了解高级语言源程序中的语句与机器级代码之间的对应关系
  - 了解复杂数据类型（数组、结构等）的机器级实现
- 主要教学内容
  - 介绍C语言程序与机器级指令之间的对应关系。
  - 主要包括：程序转换概述、 x86-64指令系统、C语言中控制语句和过程调用等机器级实现、复杂数据类型（数组、结构等）的机器级实现等。
  - 本章所用的机器级表示主要以汇编语言形式表示为主。

采用逆向工程方法!

程序的机器级表示

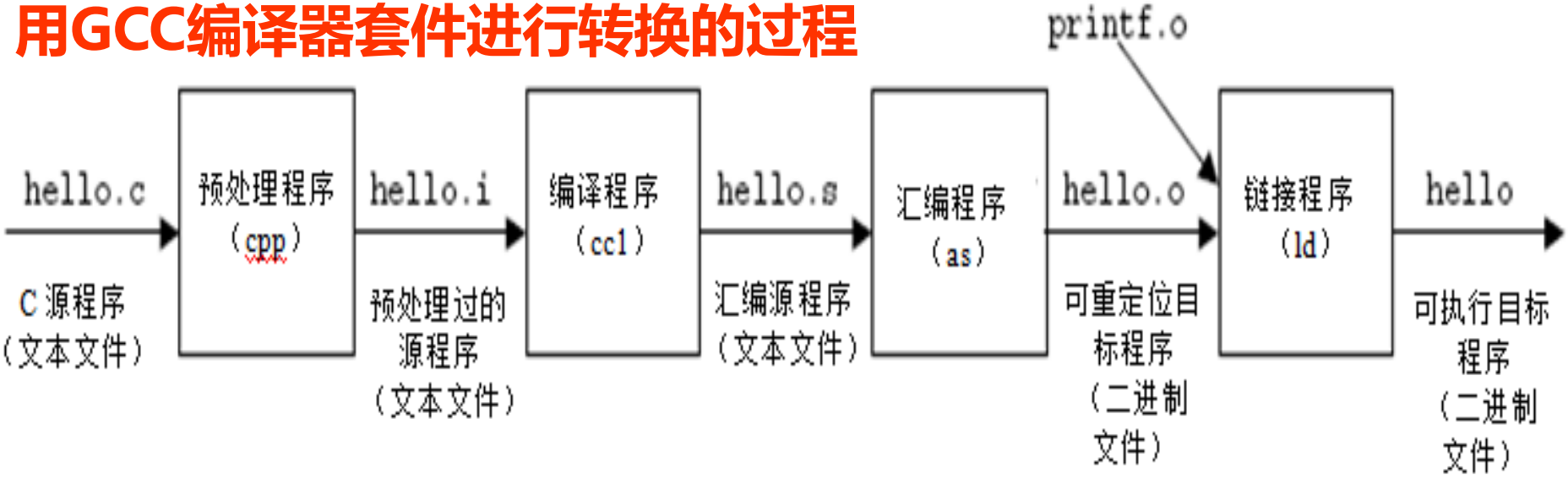
- 分以下五个部分介绍
  - 第一讲： x86-64指令系统
    - 高级语言程序转换为机器代码的过程
    - 机器指令和汇编指令
    - x86-64指令系统
  - 第二讲： C语言程序的机器级表示
    - 选择语句的机器级表示
    - 循环结构的机器级表示
    - 过程调用的机器级表示
  - 第三讲：复杂数据类型的分配和访问
    - 数组的分配和访问
    - 结构体数据的分配和访问
    - 联合体数据的分配和访问
    - 数据的对齐
  - 第四讲：内存越界引用和缓冲区溢出
  - 第五讲：浮点指令和代码

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

围绕C语言中的语句和复杂数据类型，解释其在底层机器级的实现方法

高级语言程序转换为机器代码的过程

用GCC编译器套件进行转换的过程



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

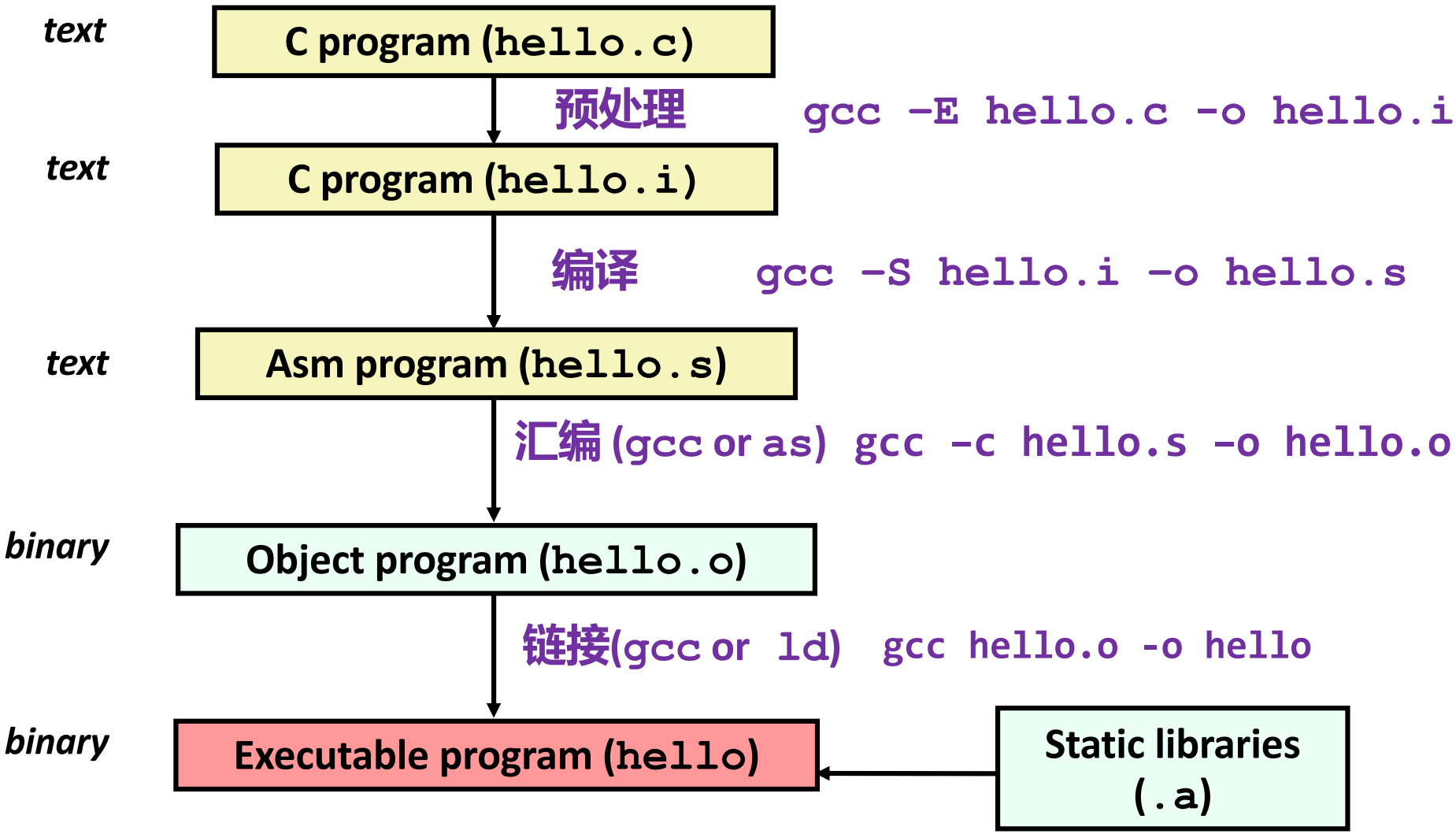
编译：将预处理后的源程序文件编译生成相应的汇编语言程序。

汇编：由汇编程序将汇编语言源程序文件转换为可重定位的机器语言目标代码文件。

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

高级语言程序转换为机器代码的过程

- Code in files **`hello.c`**
- Compile with command: **`gcc hello.c -o hello`**



C Code (mstore.c)

```
long mult2(long , long);
void multstore(
     long x,
     long y,
     long *dest)
{
   long t = mult2(x , y);
   *dest=t;
}
```

Generated x86-64 Assembly

```
multstore:
   pushq %rbx
   movq %rdx,%rbx
   call mult2
   movq %rax,(%rbx)
   popq %rbx
   ret
```

linux> *gcc -Og -S mstore .c -o mstore.s*

选项 -S 将mstore.c编译为mstore.s汇编程序

-Og 表示采用不会干扰调试的优化

linux> *gcc -Og -c mstore.c -o mstore.o*

linux> *objdump -d mstore.o* 将mstore.o反汇编

```
Disassembly of function multstore in binary file mstore.o
0000000000000000 <multstore>:
Offset   Bytes              Equivalent assembly language
 0:      53                 push   %rbx
 1:      48 89 d3           mov    %rdx,%rbx
 4:      e8 00 00 00 00     callq  9 <multstore+0x9>
 9:      48 89 03           mov    %rax,(%rbx)
 c:      5b                 pop    %rbx
 d:      c3                 retq
```

相对地址　机器指令　汇编指令

编译得到的与反汇编得到的汇编指令形式稍有差异

C Code (main.c)

```
#include <stdio.h>
void multstore(long , long , long *);
int main(){
  long d;
  multstore(2 , 3 , &d);
  printf("2 * 3 --> %ld\n", d);
  return 0;
}
long mult2(long a , long b) {
  long s= a * b;
  return s;
}
```

linux> *gcc -Og -no-pie main.c mstore.c -o prog*

两个源程序文件main.c和mstore.c，最终生成可执行文件为prog

选项 -o指出输出文件名

-no-pie表示采用固定地址

```
Disassembly of function multstore in binary file mstore.o
0000000000000000 <multstore>:
Offset   Bytes                Equivalent assembly language
 0:      53                   push   %rbx
 1:      48 89 d3             mov    %rdx,%rbx
 4:      e8 00 00 00 00       callq  9 <multstore+0x9>
 9:      48 89 03             mov    %rax,(%rbx)
 c:      5b                   pop    %rbx
 d:      c3                   retq
```

mstore.o可重定位目标文件

相对地址

objdump –d mstore.o 结果

```
0000000000400540 <multstore>:
400540:  53                   push   %rbx
400541:  48 89 d3             mov    %rdx,%rbx
400544:  e8 42 00 00 00       callq  40058b <mult2>
400549:  48 89 03             mov    %rax,(%rbx)
40054c:  5b                   pop    %rbx
40054d:  c3                   retq
40054e:  90                   nop
40054f:  90                   nop
```

prog可执行目标文件

绝对地址

objdump –d prog 结果

x86指令概述

指令：在某种计算机结构中定义的单个CPU操作，每条指令执行一个特定的操作。指令也可以理解为：通知CPU执行某种操作的“命令”。CPU全部指令的集合，称为指令集。

x86-64指令集 兼容 IA32指令集

机器指令：二进制格式的序列（一串0，1代码）。

注意：硬件只能识别，存储，运行机器指令

符号指令：字符串形式的序列（包含字符串形式的操作码以及操作数助记符）。

x86指令概述

AT&T与Intel汇编 两种代码格式

```
multstore:
   pushq %rbx
   movq %rdx,%rbx
   call mult2
   movq %rax,(%rbx)
   popq %rbx
   ret
```

```
multstore:
   push rbx
   mov  rbx,rdx
   call mult2
   mov [rbx],rax
   pop rbx
   ret
```

使用 % 表示寄存器
使用 $ 表示立即数，例如 $0x10。
使用 () 表示内存地址，例如 (%eax)。
操作数顺序是 源操作数在前，目标操作数在后（即 op src, dst）。
指令名称通常带有后缀，表示操作数大小（q 表示四字，即8字节64位）。

指令的组成：操作码 + 操作数

- **操作码 —— 告诉计算机要执行的操作是什么，如：加、减、逻辑与等。**
- **操作数 —— 执行操作过程所要操作的数，如加运算的两个加数。**

汇编代码后缀

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	四字	q	8
char *	四字	q	8

例如： `movb $0x78,%al`

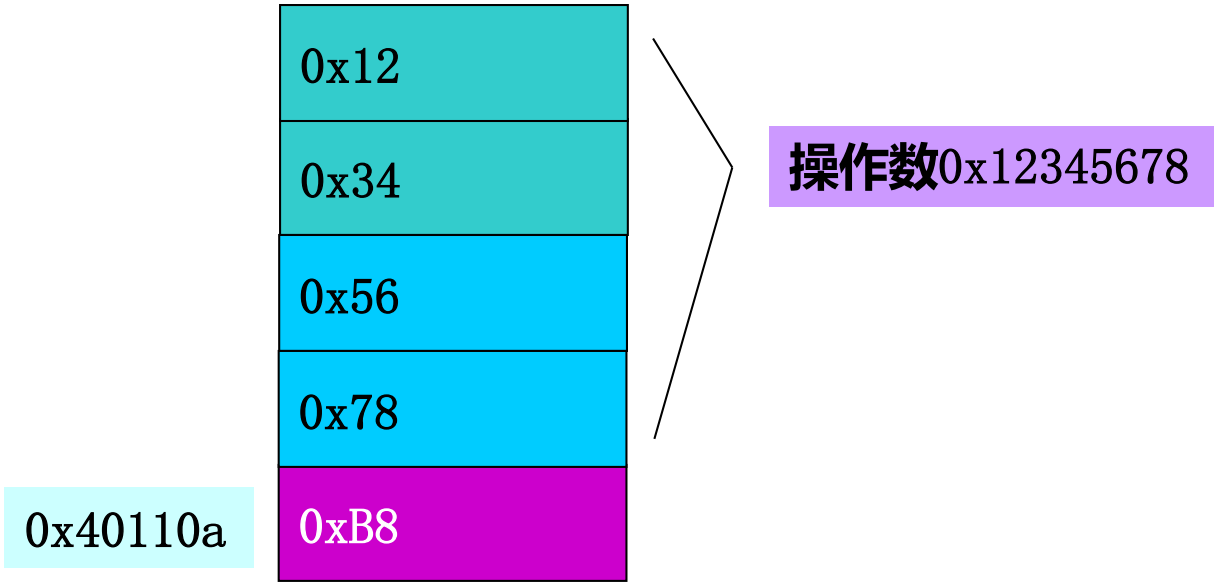
x86指令概述

指令存放

多字节操作数连续存放，顺序依据小端规则（Little Endian），即：低位字节存放在低地址单元，高位字节存放在相邻的高地址单元。

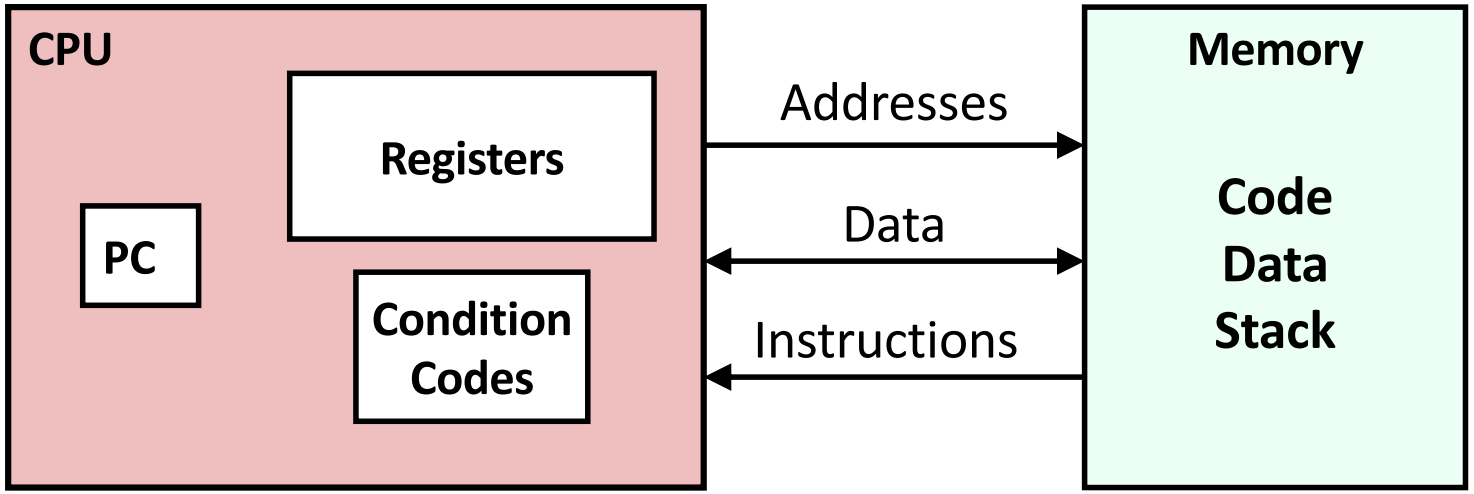
```
40110a:    b8 78 56 34 12        movl   $0x12345678,%eax
40110f:    bb 78 56 34 12        movl   $0x12345678,%ebx
```

如：40110a**单元中有一条指令** movl $0x12345678,%eax

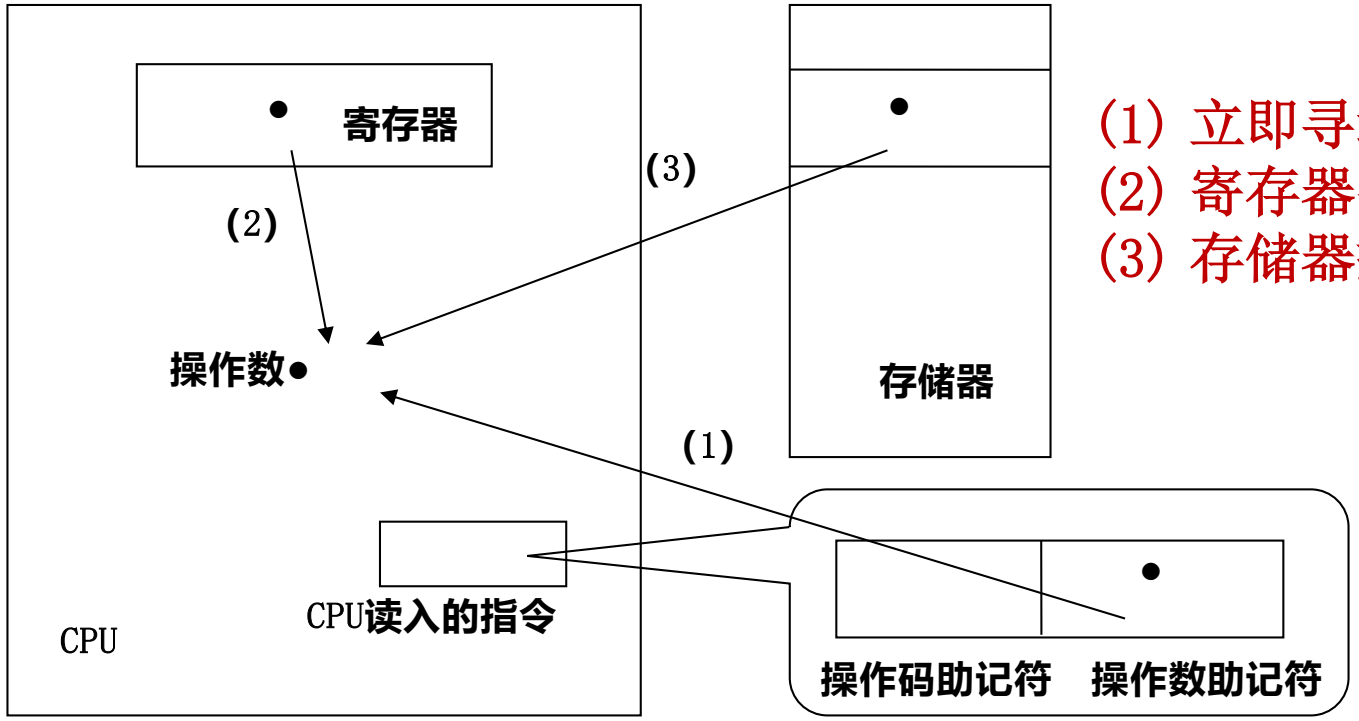


寻址方式

- 寻址方式
  - 根据指令给定信息得到操作数
- 操作数所在的位置
  - 指令中：立即寻址
  - 寄存器中：寄存器寻址
  - 存储单元中（属于存储器操作数，按字节编址）：
  - 端口中
- 存储器操作数的寻址方式与微处理器的工作模式有关
  - 两种工作模式
    - 实地址模式
    - 保护模式


CPU
Registers
PC
Condition Codes
Addresses
Data
Instructions
Memory
Code
Data
Stack

操作数对应的寻址方式



(1) 立即寻址
(2) 寄存器寻址
(3) 存储器操作数寻址

指令中需给出的信息：

操作性质（操作码）

源操作数1 或/和 源操作数2　　（立即数、寄存器名称、存储地址）

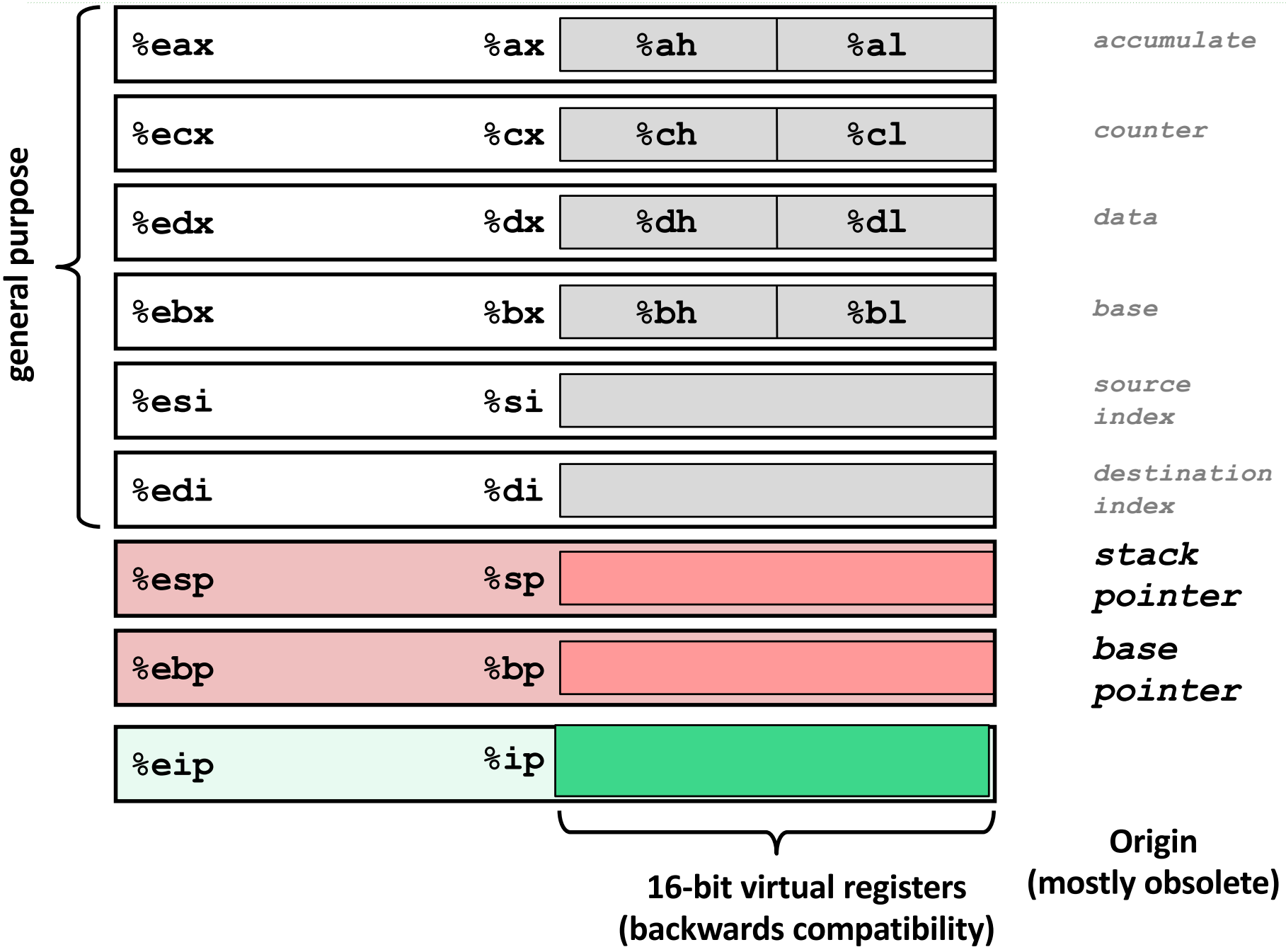
目的操作数　　（寄存器名称、存储地址）

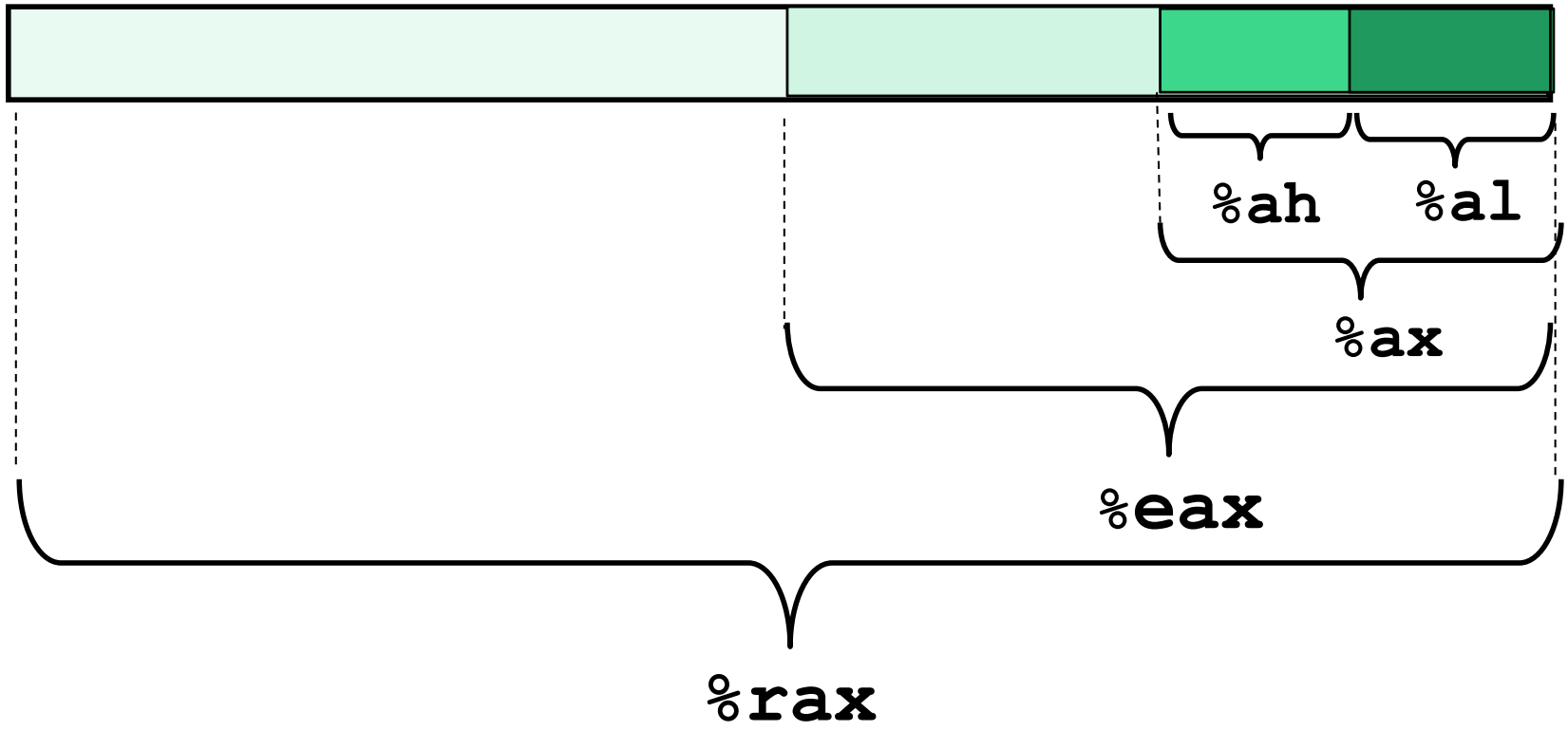
x86-64 整数寄存器

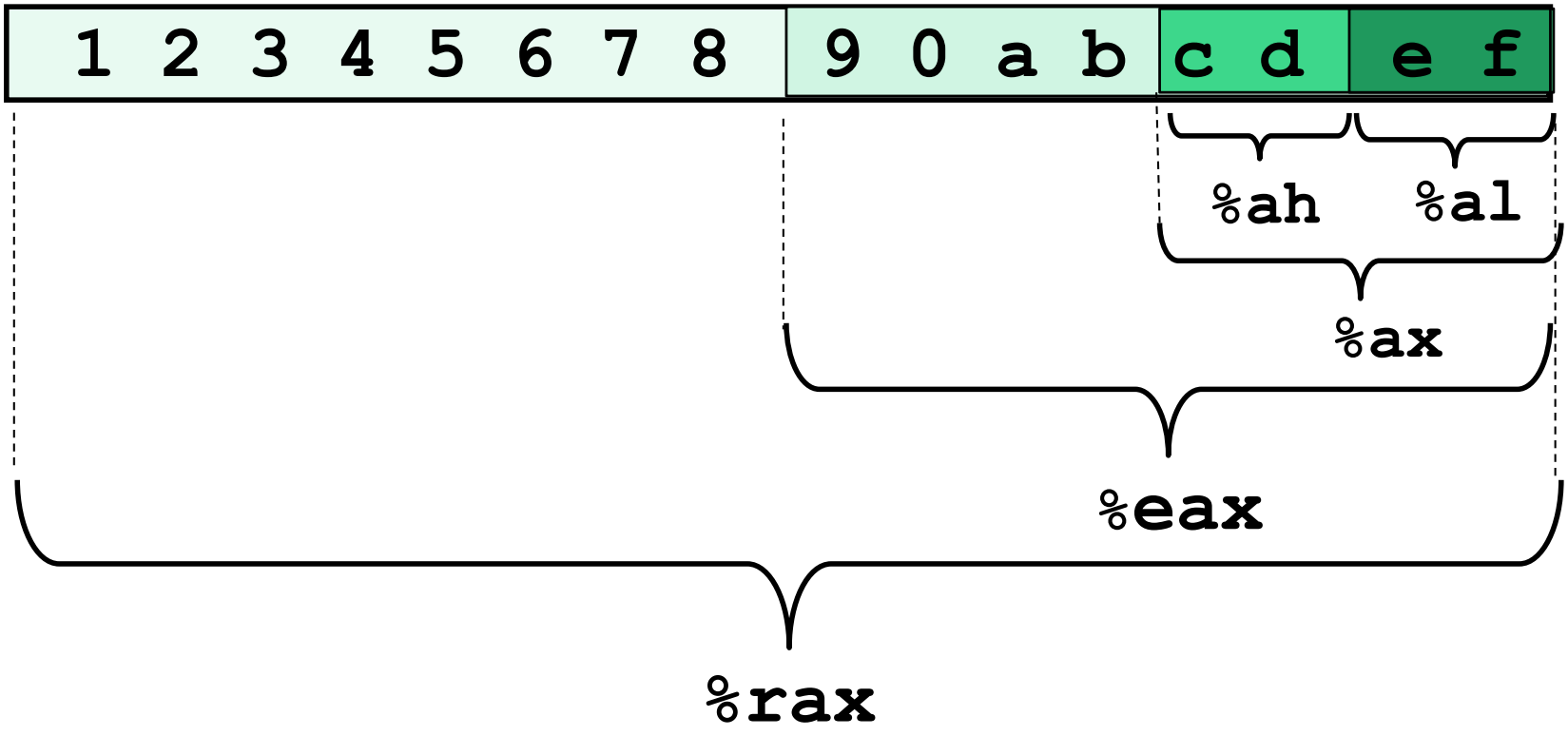
%rax	%eax
%rbx	%ebx
%rcx	%ecx
%rdx	%edx
%rsi	%esi
%rdi	%edi
%rsp	%esp
%rbp	%ebp

%r8	%r8d
%r9	%r9d
%r10	%r10d
%r11	%r11d
%r12	%r12d
%r13	%r13d
%r14	%r14d
%r15	%r15d

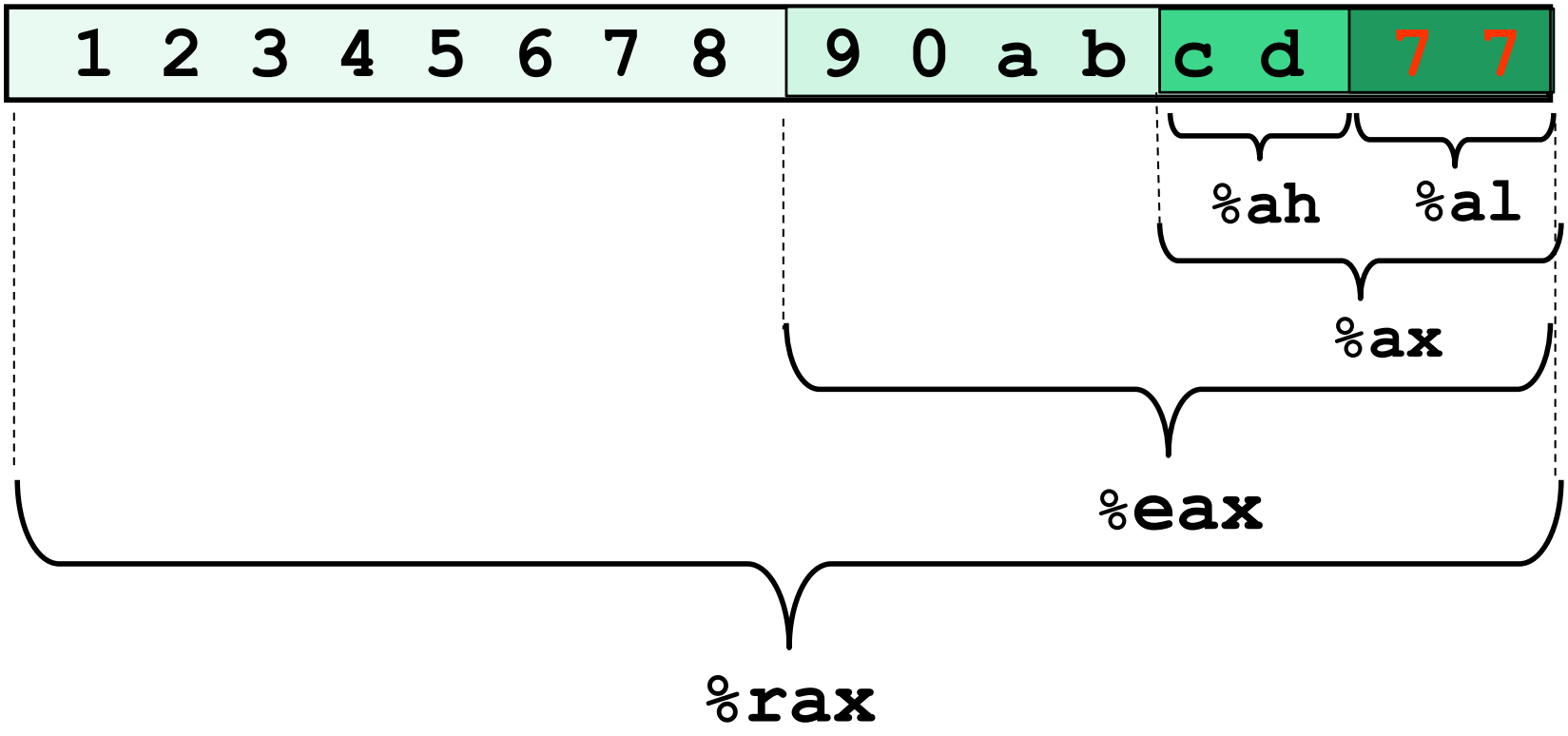
%rip	%eip


general purpose
%eax
%ax
%ah
%al
accumulate
%ecx
%cx
%ch
%cl
counter
%edx
%dx
%dh
%dl
data
%ebx
%bx
%bh
%bl
base
%esi
%si
source index
%edi
%di
destination index
%esp
%sp
stack pointer
%ebp
%bp
base pointer
%eip
%ip
16-bit virtual registers
(backwards compatibility)
Origin
(mostly obsolete)

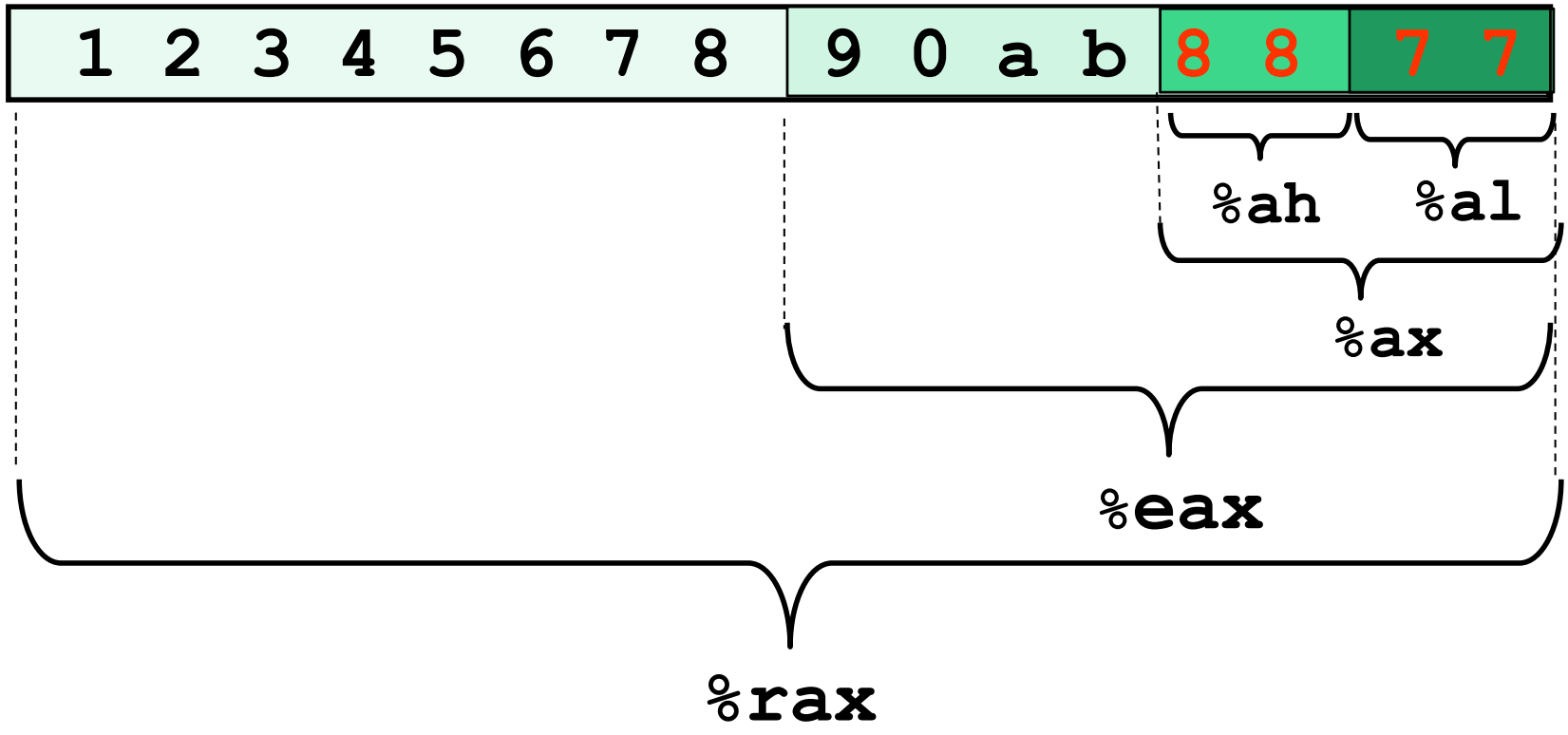

%ah
%al
%ax
%eax
%rax



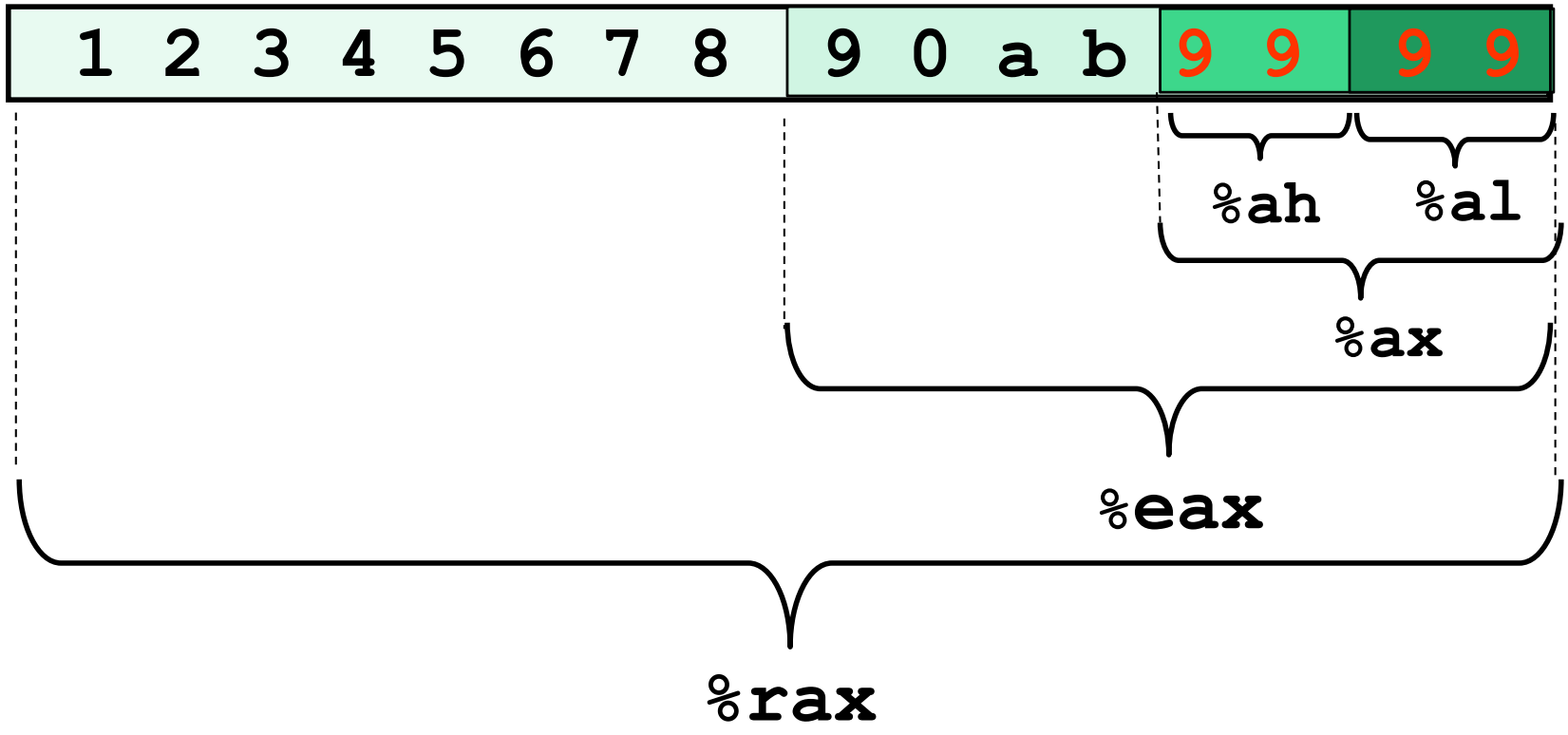

```
movabsq $0x1234567890abcdef,%rax
movb $0x77,%al
```



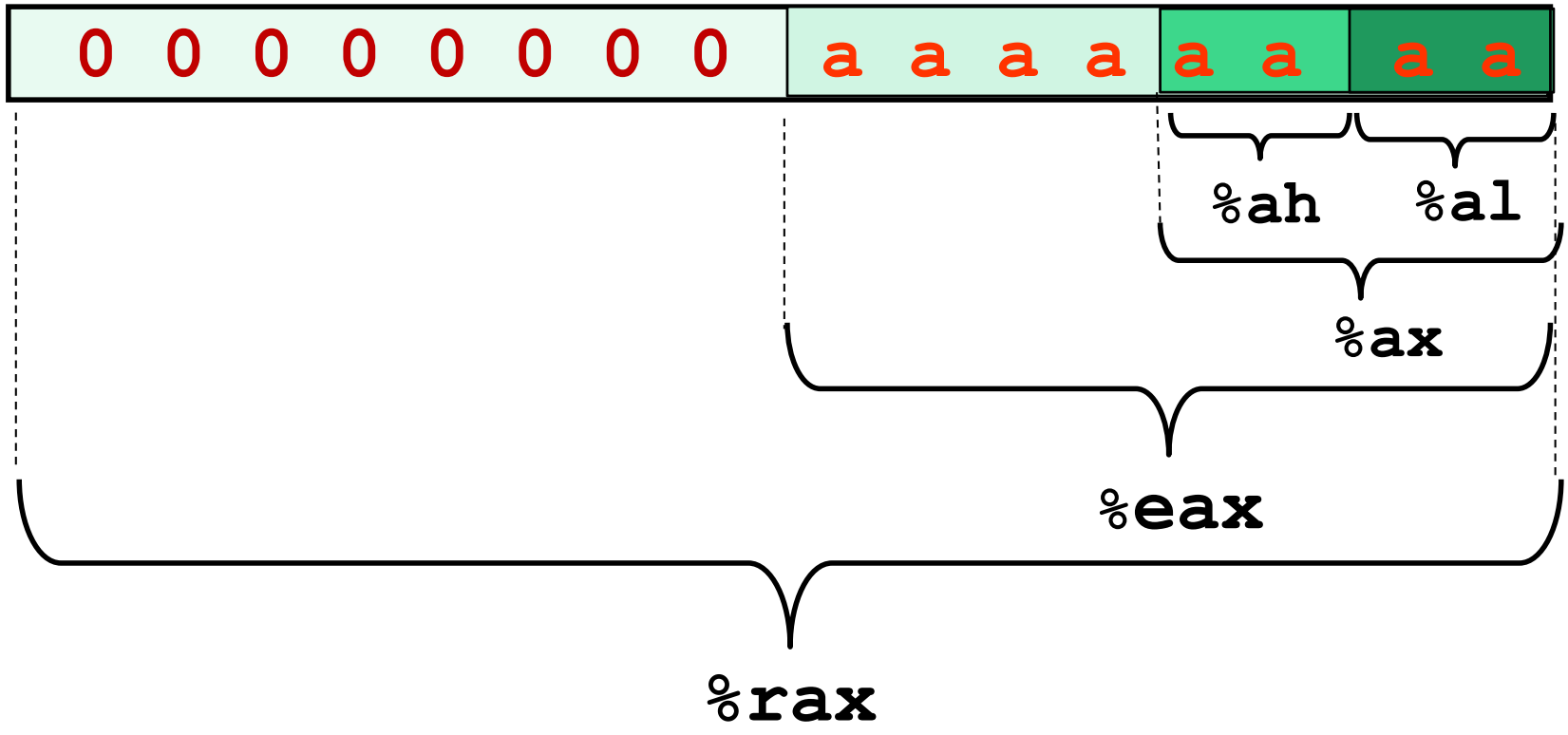

```
movabsq $0x1234567890abcdef,%rax
movb $0x77,%al
movb $0x88,%ah
```




```
movabsq $0x1234567890abcdef,%rax
movb $0x77,%al
movb $0x88,%ah
movw $0x9999,%ax
```




```
movabsq $0x1234567890abcdef,%rax
movb $0x77,%al
movb $0x88,%ah
movw $0x9999,%ax
movl $0xaaaaaaaa,%eax
```



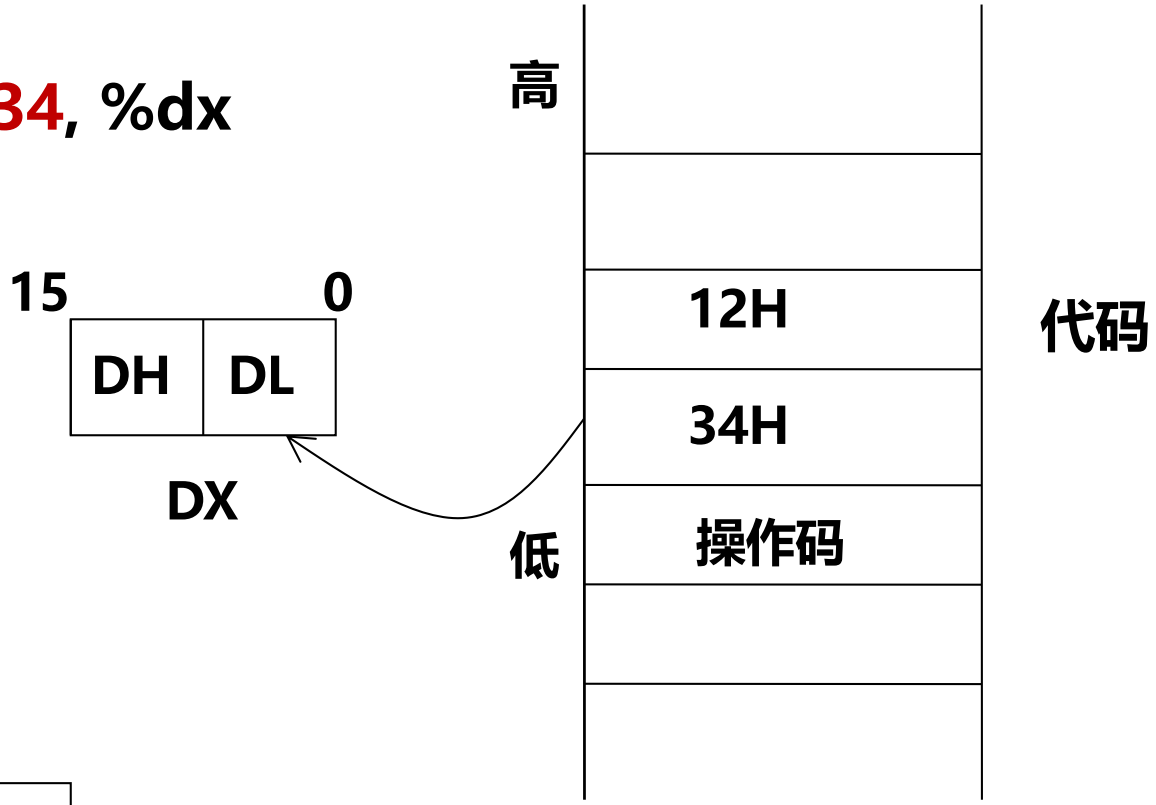

```
movabsq $0x1234567890abcdef,%rax
movb $0x77,%al
movb $0x88,%ah
movw $0x9999,%ax
movl $0xaaaaaaaa,%eax
```

寻址方式

立即寻址-寻址方式所提供的操作数直接放在指令中，紧跟在操作码的后面，与操作码一起放在代码区域中。

立即数的书写方式： $ 后面跟一个用标准C表示法表示的整数，例如：$-577或$0x12。

例：movw $0x1234, %dx



立即寻址方式示意图

寻址方式

寄存器寻址-操作数在CPU的某个寄存器中

书写方式： % 后面跟一个寄存器名称

例：movw $0x1234, %dx
movw %dx, %ax

3. 存储器操作数寻址方式-操作数在存储器中

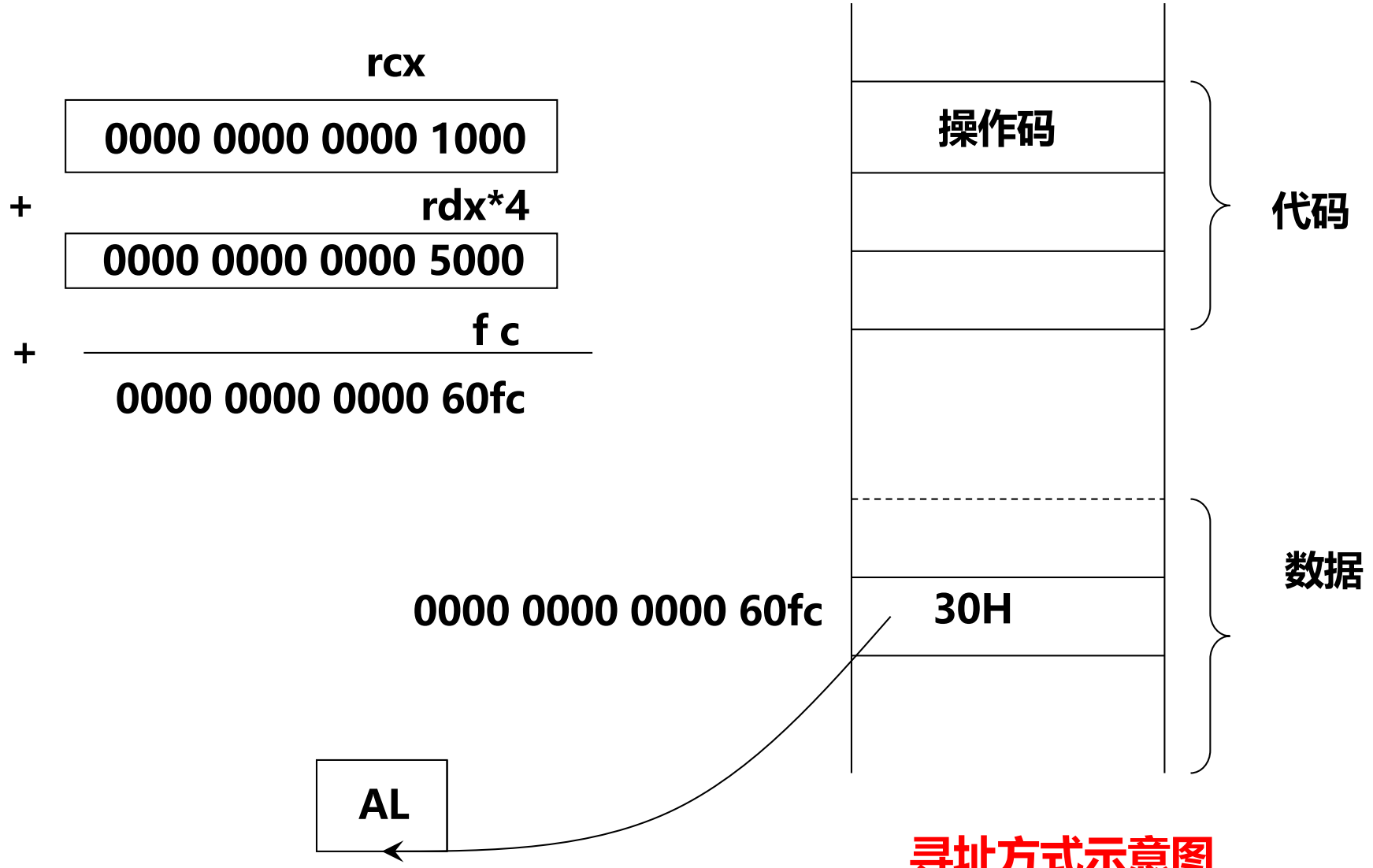
给出操作数所存放内存单元的有效地址。

有效地址通用表达式: $Imm(r_b, r_i, s)$

四个组成部分:
立即数偏移: Imm
基址寄存器: r_b
变址寄存器: r_i
比例因子: s 必须是1、2、4或者8

有效地址 $= Imm + R[r_b] + R[r_i] \bullet s$

例： movb 0xfc(%rcx,%rdx,4), %al



寻址方式示意图

表达式	操作数	
Imm	M[*Imm*]	绝对寻址
(r_a)	M[R[r_a]]	间接寻址
Imm(r_b)	M[*Imm*+R[r_b]]	（基址 + 偏移量）寻址
(r_b, r_i)	M[R[r_b]+R[r_i]]	变址寻址
Imm(r_b, r_i)	M[*Imm*+R[r_b]+R[r_i]]	变址寻址
(,r_i, s)	M[R[r_i] · s]	比例变址寻址
Imm(,r_i,s)	M[*Imm*+R[r_i] · s]	比例变址寻址
(r_b, r_i, s)	M[R[r_b]+R[r_i] · s]	比例变址寻址
Imm(r_b, r_i, s)	M[*Imm*+R[r_b]+R[r_i] · s]	比例变址寻址

有效地址计算

%rdx	0xf000
%rcx	0x0100

Expression	Address Computation	Address
0x8(%rdx)	0xf000 + 0x8	0xf008
(%rdx,%rcx)	0xf000 + 0x100	0xf100
(%rdx,%rcx,4)	0xf000 + 4*0x100	0xf400
0x80(,%rdx,2)	2*0xf000 + 0x80	0x1e080

寻址方式举例

地址	值
0x100	0xff
0x104	0xab
0x108	0x13
0x10c	0x11

寄存器	值
rax	0x100
rcx	0x1
rdx	0x3

操作数	值	操作数	值
%rax			
0x104		260(%rcx,%rdx)	
$0x108		0xFC(,%rcx,4)	
(%rax)		(%rax,%rcx,4)	
4(%rax)		9(%rax,%rdx)	

寻址方式举例

地址	值
0x100	0xff
0x104	0xab
0x108	0x13
0x10c	0x11

寄存器	值
rax	0x100
rcx	0x1
rdx	0x3

操作数	值	操作数	值
%rax	0x100		
0x104	0xab	260(%rcx,%rdx)	0x13
$0x108	0x108	0xFC(,%rcx,4)	0xff
(%rax)	0xff	(%rax,%rcx,4)	0xab
4(%rax)	0xab	9(%rax,%rdx)	0x11

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

a[i]的地址如何计算?

104+i×4

i=99时，104+99×4=500

b[i][j]的地址如何计算?

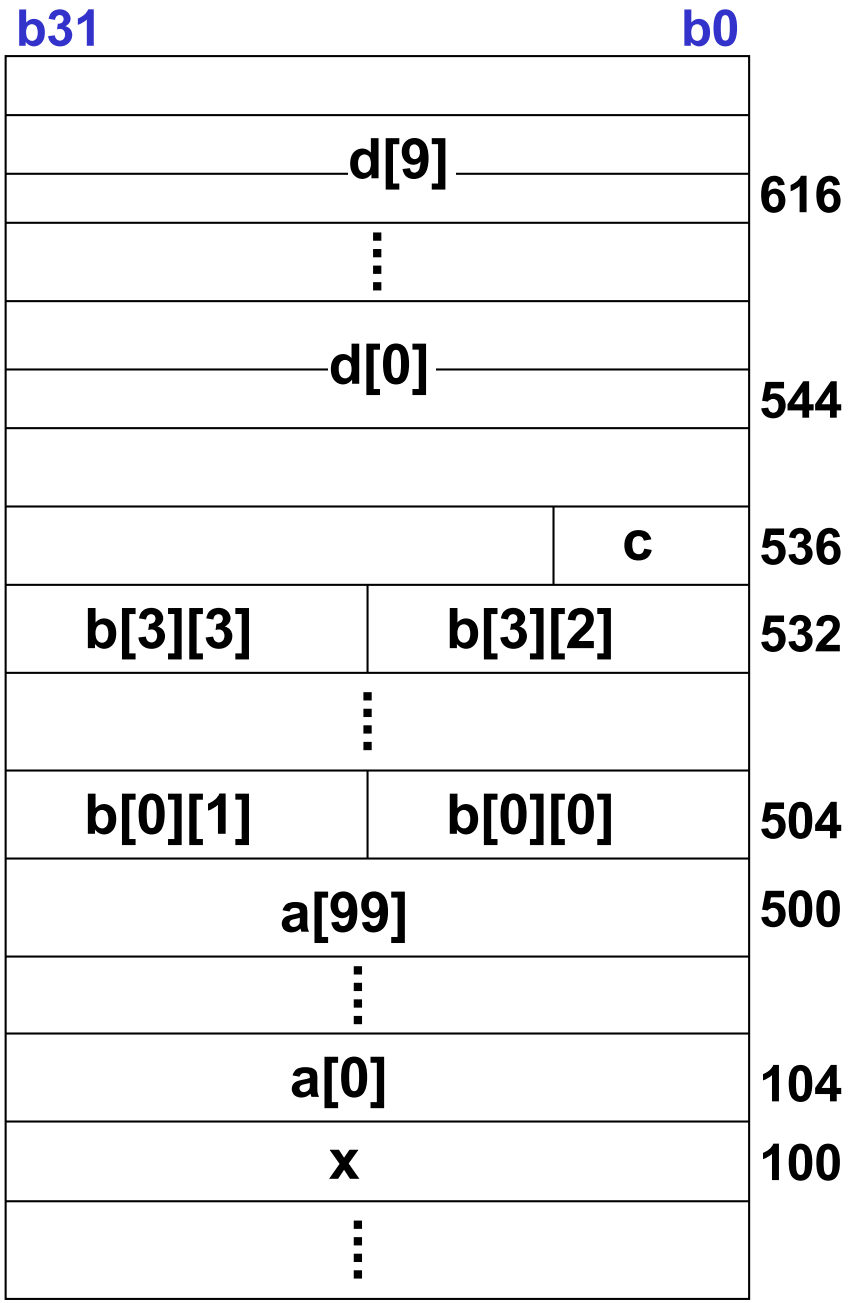
504+i×8+j×2

i=3、j=2时，504+24+4=532

d[i]的地址如何计算?

544+i×8

i=9时，544+9×8=616



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

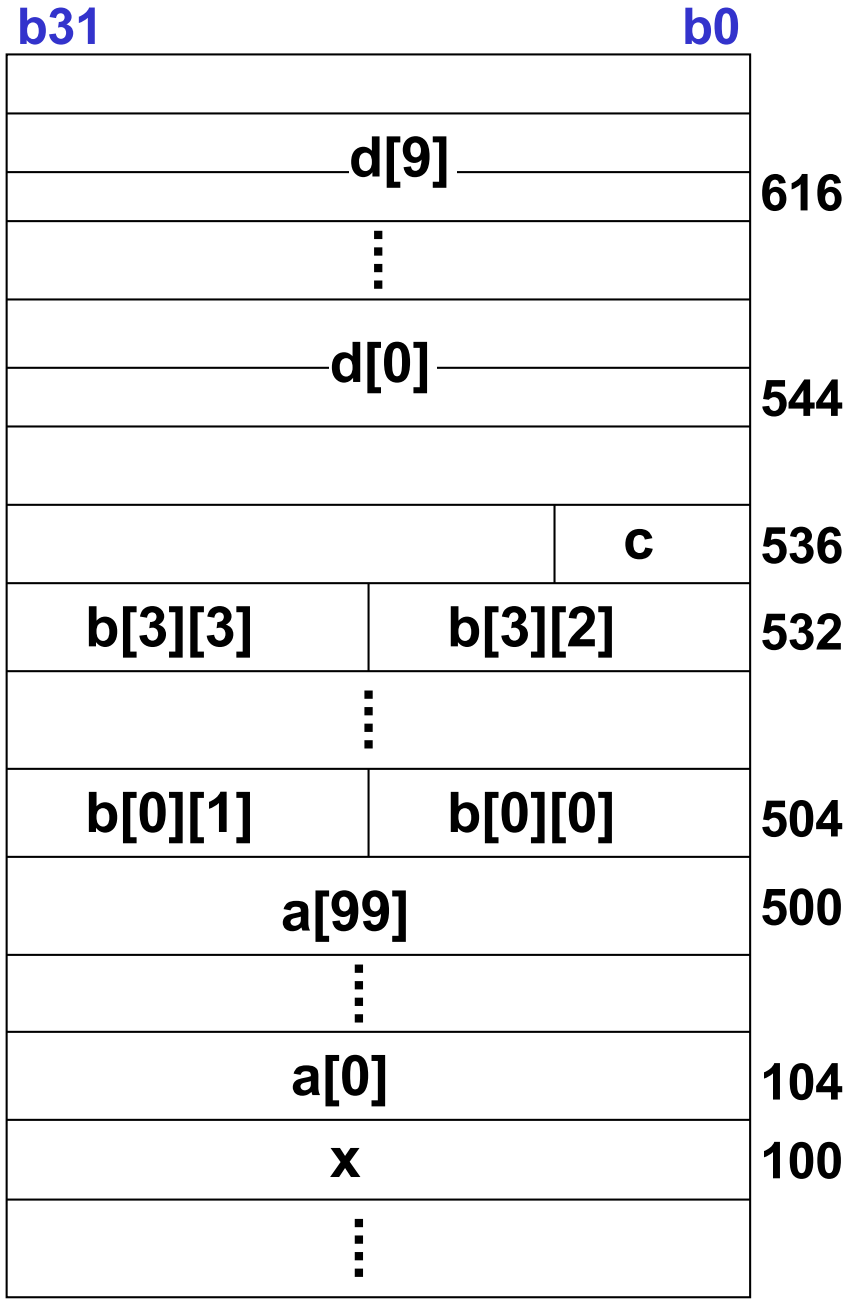
各变量应采用什么寻址方式?

X、C：**绝对寻址**

```
movl 100, %eax
movb 536, %al
```

X、C：**间接寻址**

```
movq  $100, %rcx
movl (%rcx), %eax
```



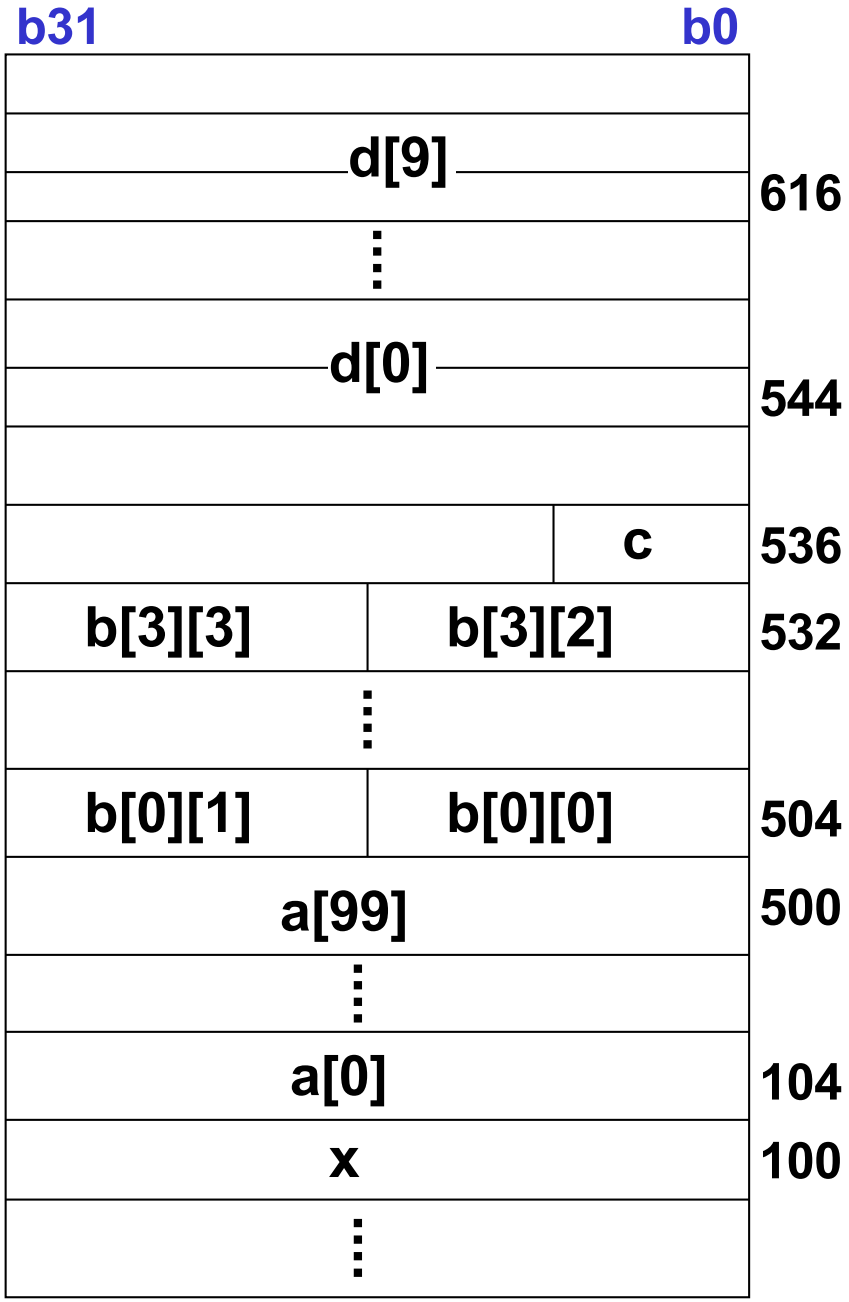
```
int x;
float a[100];
short b[4][4];
char c;
double d[10];
```

各变量应采用什么寻址方式?

a[i]: 104+i×4
比例变址+位移

d[i]: 544+i×8
比例变址+位移

```
movl 104(,%rsi,4), %eax
movq 544(,%rsi,8), %rax
```



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

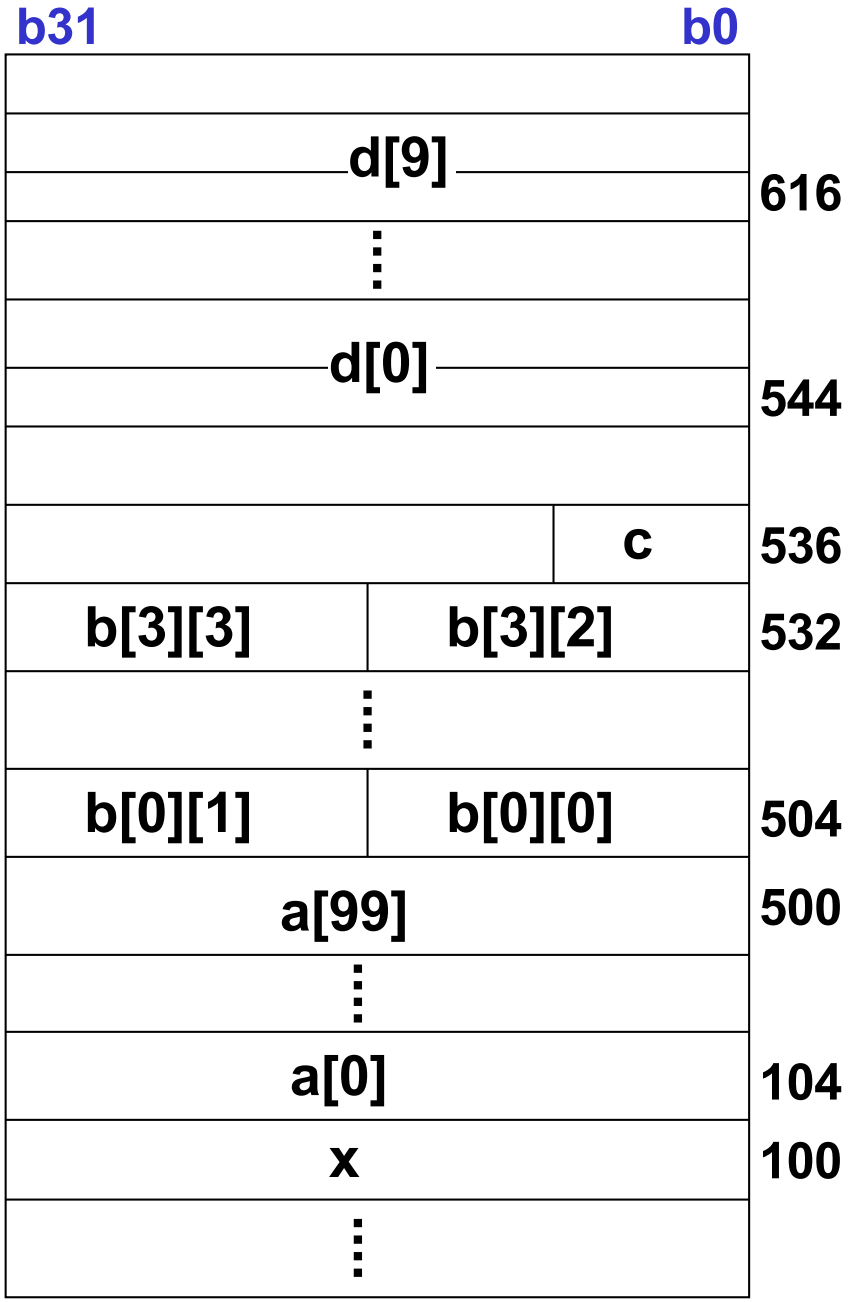
各变量应采用什么寻址方式?

b[i][j]: 504+i×8+j×2,
基址+比例变址+位移

将b[i][j]取到AX中的指令可以是:

"movw 504(%rbp,%rsi,2), %ax"

其中, i×8在rbp中, j在rsi中,
2为比例因子



x86-64常用指令类型

(1)通用数据传送指令

MOV：一般传送，包括movb、movw、movl、movq、movabsq等

MOVS：符号扩展传送，如movsbw、movswl等

MOVZ：零扩展传送，如movzwl、movzbl等

	movq	movabsq
效果	传送4字	传送绝对4字
可否操作内存	可，movq %rax, (%rbx)	否，只能用于寄存器
源/目标	寄存器、内存、32位立即数	64位立即数，64位寄存器
立即数	32位（符号扩展为64位）	64位（完整）
	大于0x7fffffff报错或截断	

mov传送指令

	Source	Dest	Src,Dest	C Analog
movq	*Imm*	*Reg*	`movq $0x4,%rax`	`temp = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	*Reg*	`movq %rax,%rdx`	`temp2 = temp1;`
		Mem	`movq %rax,(%rdx)`	`*p = temp;`
	Mem	*Reg*	`movq (%rax),%rdx`	`temp = *p;`

两个操作数不能同为内存操作数

例:

```
movabsq $0x0011223344556677, %rax
rax = 0011223344556677
movb $-1, %al
rax = 0011223344556FF
movw $-1, %ax
rax = 001122334455FFFF
movl $-1, %eax
rax = 00000000FFFFFFFF
movq $-1, %rax
rax = FFFFFFFFFFFFFFFF
```

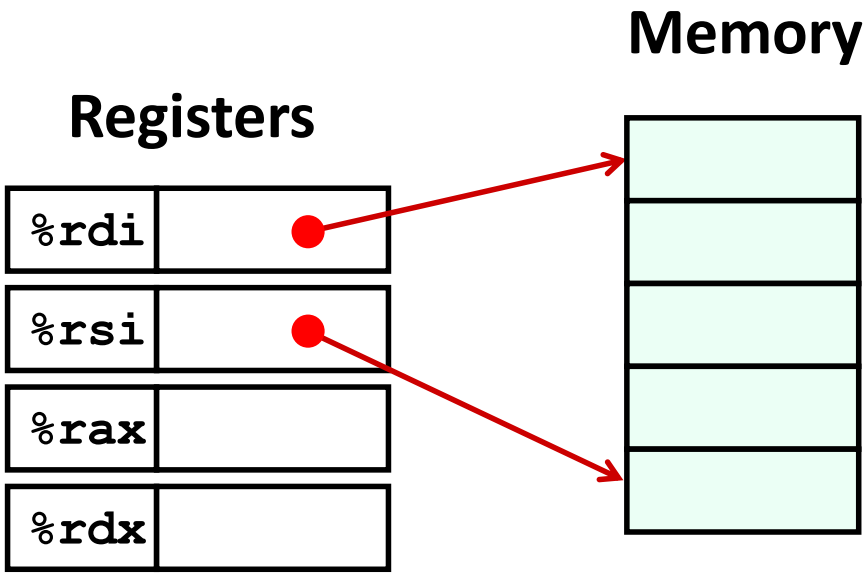
例：Swap()

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

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

例：Swap()

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



Register	Value
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

例：Swap()

Registers

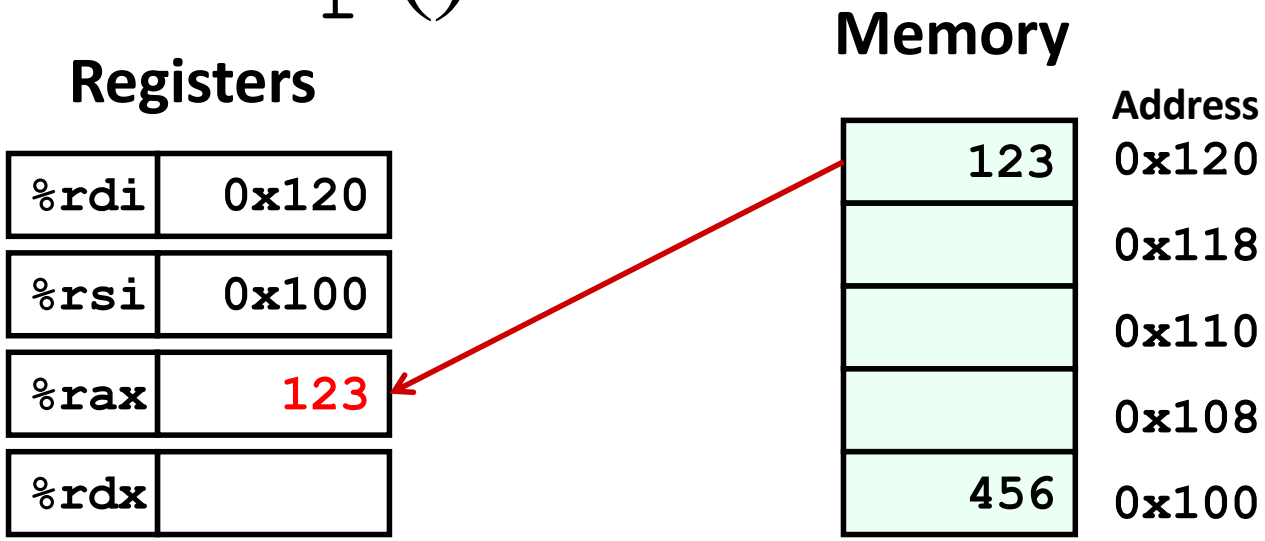
%rdi	0x120
%rsi	0x100
%rax	
%rdx	

Memory

	Address
123	0x120
	0x118
	0x110
	0x108
456	0x100

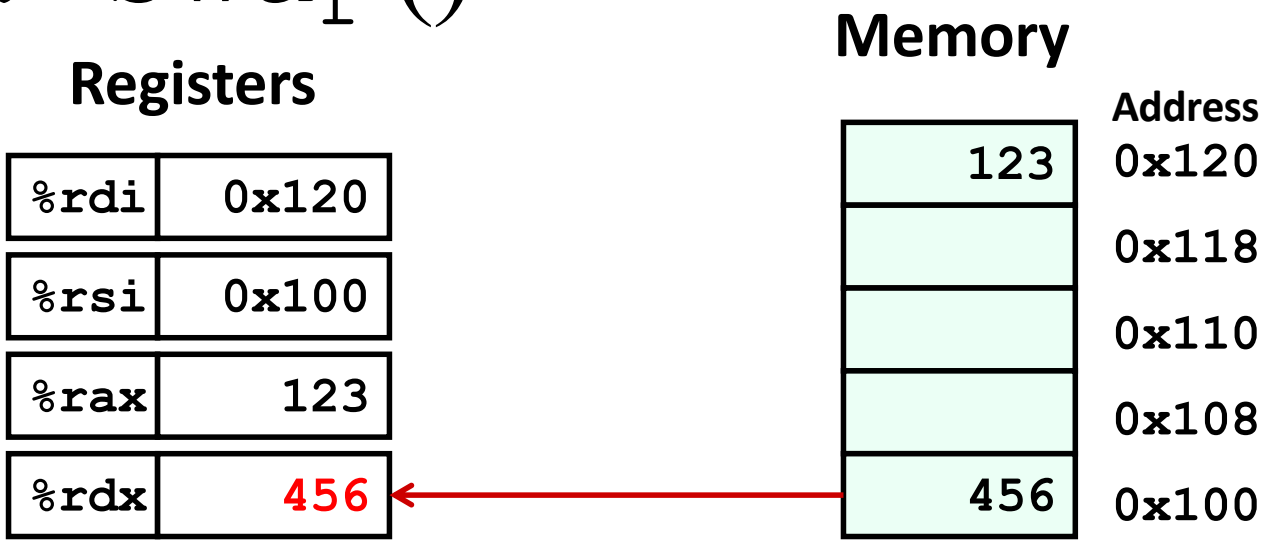
```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

例：Swap()



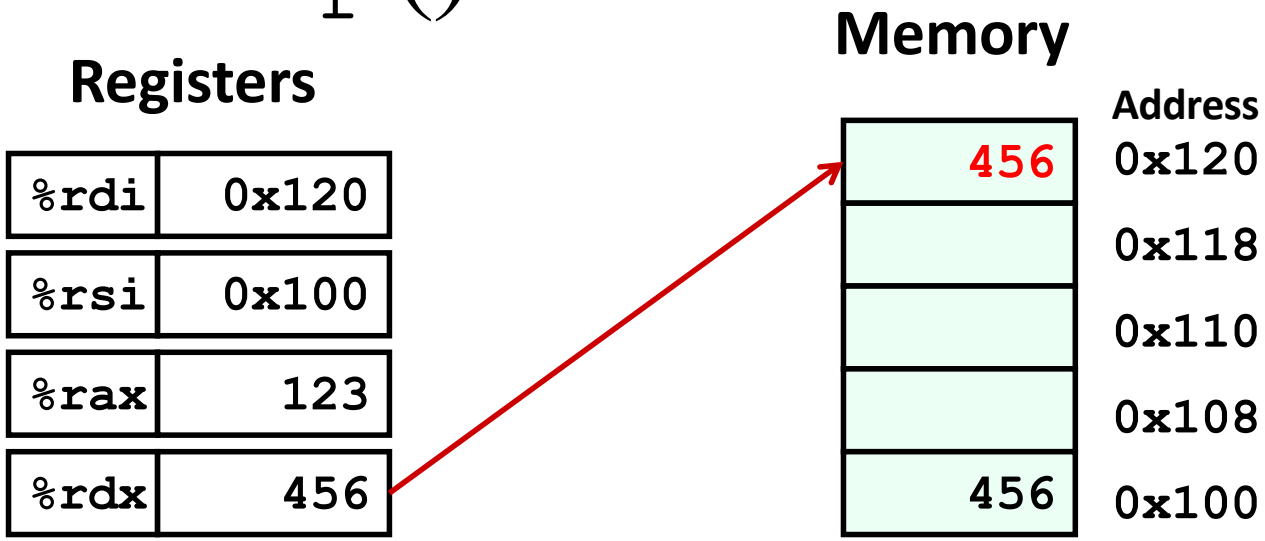

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

例：Swap()



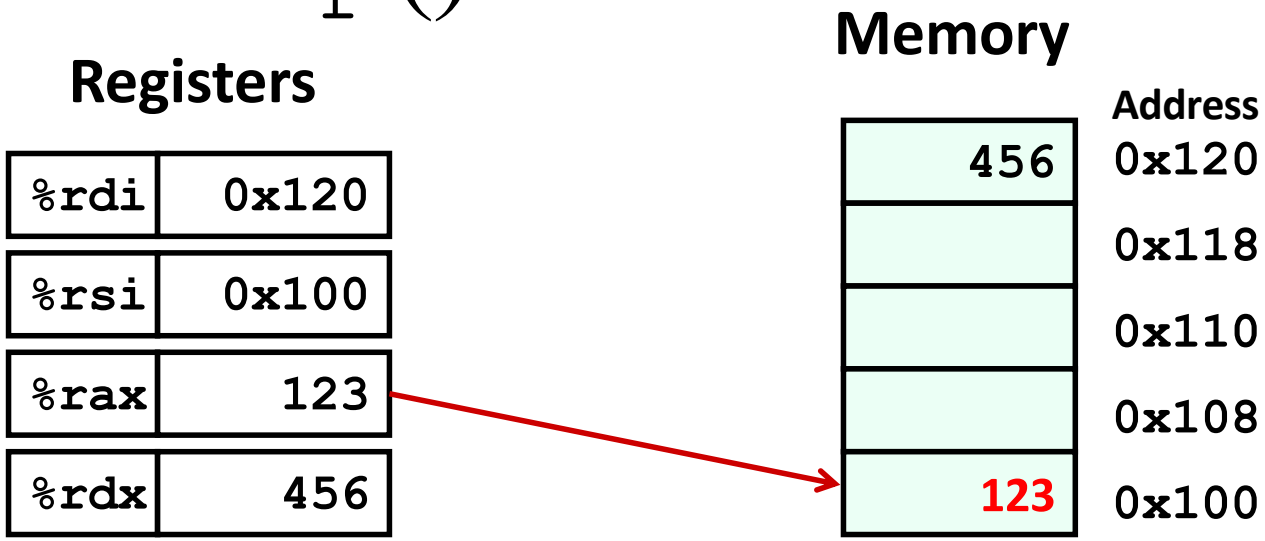

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

例：Swap()




```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

例：Swap()




```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

MOVS：符号扩展传送

指令	效果	描述
MOVZ S，R	R←零扩展(S)	以零扩展进行传送
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
movzbq		将做了零扩展的字节传送到四字
movzwq		将做了零扩展的字传送到四字

MOVZ：零扩展传送

指令	效果	描述
MOVS S，R	R←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
movsbq		将做了符号扩展的字节传送到四字
movswq		将做了符号扩展的字传送到四字
movslq		将做了符号扩展的双字传送到四字
cltq	%rax ←符号扩展(%eax)	把%eax 符号扩展到%rax

```
movabsq $0x0011223344556677, %rax
movb    $0xAA, %dl
movb    %dl, %al
movsbq  %dl, %rax
movzbq  %dl, %rax
```

rax = 0011223344556677
dl = AA
rax = 0011223344556AA
rax = FFFFFFFFFFFFFFAA
rax = 00000000000AA

```
long *sp;
long *dp;
*dp=*sp;
```

位相等 ⟹

```
movq (%rsi),%rax
movq %rax, (%rdi)
```

```
char *sp;
int *dp;
*dp=(int)*sp;
```

位扩展 ⟹

```
movsbl (%rsi),%eax
movl %eax, (%rdi)
```

```
unsigned char *sp;
int *dp;
*dp=(int)*sp;
```

位扩展 ⟹

```
movzbl (%rsi),%eax
movl %eax, (%rdi)
```

```
int *sp;
char *dp;
*dp=(char)*sp;
```

位截断 ⟹

```
movl (%rsi),%eax
movb %al, (%rdi)
```

- **`lea` Src, Dst**
  1. 计算有效地址

     E.g., translation of **`p = &x[i];`**
  2. 算术运算表达式 形式为： x + k*y
     - k = 1, 2, 4, or 8
     - E.g., leaq 7(%rdx,%rdx,4), %rax [%rax]=5*[%rdx]+7

       ~~M[5*[%rdx]+7]~~

```
long m12(long x)
{
  return x*12;
}
```

Converted to ASM by compiler:

```
leaq (%rdi,%rdi,2), %rax # return <- x+x*2
salq $2, %rax            # return <- return<<2
```

堆栈操作指令

堆栈——按照后进先出的原则组织的一段内存区域。

堆栈指针rsp的初值决定了堆栈的大小，rsp始终指向堆栈的顶部，即始终指向最后推入堆栈的信息所在的单元。

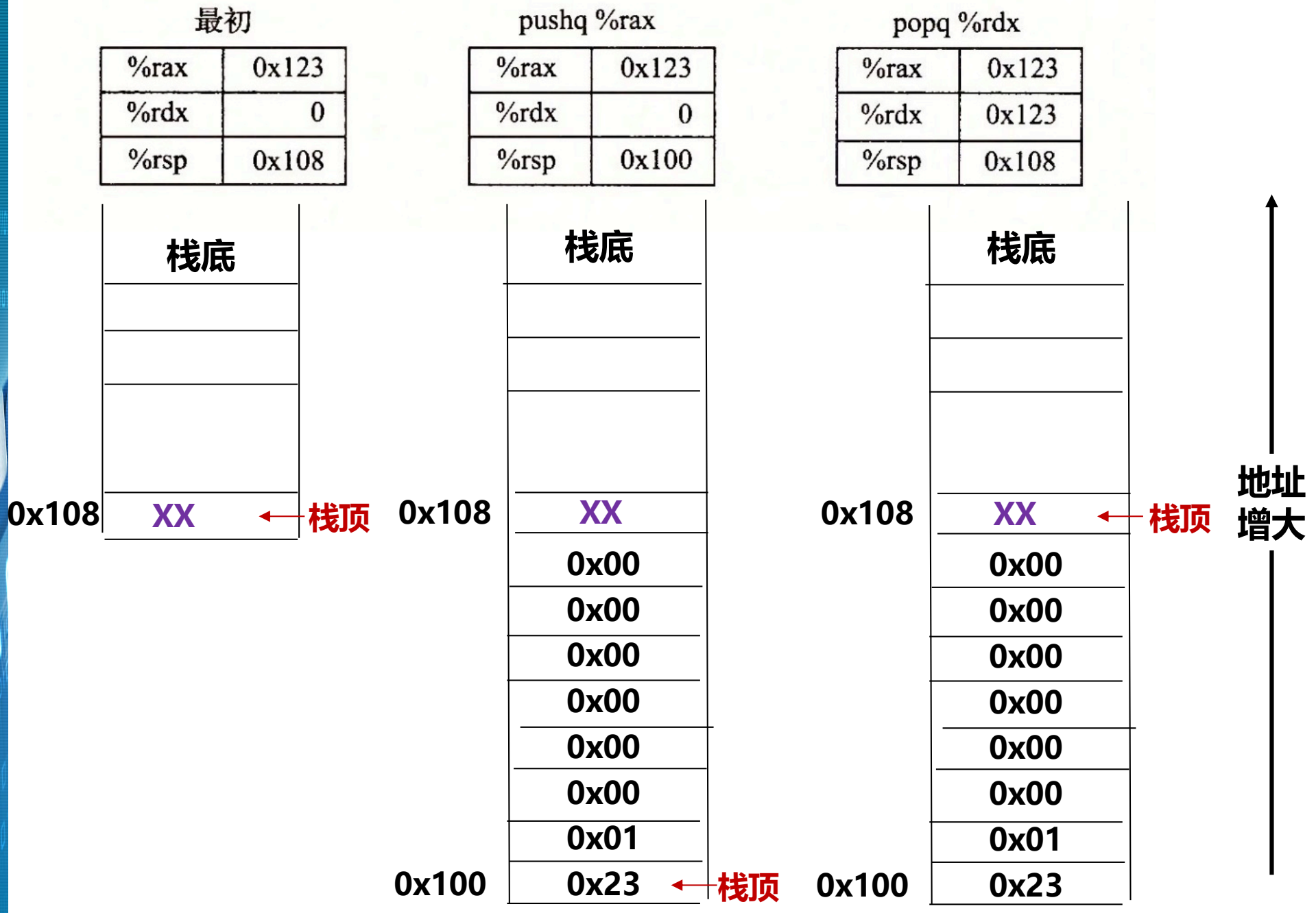
栈顶：栈区的低地址

栈底：栈区的高地址

PUSH/POP：入栈/出栈指令

pushw, pushl, pushq, popw, popl, popq等

“入栈”和“出栈”操作


最初
%rax 0x123
%rdx 0
%rsp 0x108
栈底
0x108 XX 栈顶
pushq %rax
%rax 0x123
%rdx 0
%rsp 0x100
栈底
0x108 XX 栈顶
0x00
0x00
0x00
0x00
0x00
0x00
0x00
0x01
0x100 0x23 栈顶
popq %rdx
%rax 0x123
%rdx 0x123
%rsp 0x108
栈底
0x108 XX 栈顶
0x00
0x00
0x00
0x00
0x00
0x00
0x01
0x100 0x23
地址
增大

“入栈”和“出栈”操作

```
movq    $0x123, %rax
movq    $0x567, %rbx
pushq   %rax
pushq   %rbx
popq    %rax
popq    %rbx
```

上述程序段执行后，rax=0x 567 ，rbx=0x 123

常用指令类型

算术运算指令（影响标志位）

- 加 / 减运算（影响标志、不区分无/带符号）
  - add：加，包括addb、addw、addl、addq
  - sub：减，包括subb、subw、subl 、subq
- 增1 / 减1运算（影响除CF以外的标志、不区分无/带符号）
  - inc：加，包括incb、incw、incl、incq
  - dec：减，包括decb、decw、decl、decq
- 取负运算（影响标志、若对0取负，则结果为0且CF=0, 否则CF=1）
  - neg：取负，包括negb、negw、negl、negq　　-n=0-n
- 乘 / 除运算（区分无/带符号）
  - mul / imul：无符号乘 / 带符号乘
  - div/ idiv：无符号除 / 带符号除

整数加法指令举例

- 假设R[ax]=FFFA H，R[bx]=FFF0 H，则执行以下指令后

addw %bx, %ax

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

解：功能：R[ax]←R[ax]+R[bx]，指令执行后的结果如下

R[ax]=FFFAH+FFF0H=FFEAH ，BX中内容不变

CF=1，OF=0，ZF=0，SF=1

若是无符号整数运算，则CF=1说明结果溢出

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

FFEA的真值为65535-21=65514≠65530+65520，溢出

若是带符号整数运算，则OF=0说明结果没有溢出

验证：FFFA的真值为-6，FFF0的真值为-16

FFEA的真值为-22=-6+(-16)，结果正确，无溢出

整数乘除指令

- 乘法指令：可给出一个、两个或三个操作数
  - 若给出一个操作数SRC，则另一个源操作数隐含在AL/AX/EAX中，结果存放在AX（16位）或DX:AX（32位）或EDX:EAX（64位）中。DX:AX表示32位乘积的高、低16位分别在DX和AX中。 n位×n位=2n位

```
movw $100,%ax
movw  $23,%bx
mulw  %bx
```

  - 若指令中给出两个操作数SRC和DST，则将SRC和DST相乘，结果在DST中。n位×n位=n位(截断)

```
imulw $23,%ax
```

  - 若指令中给出三个操作数SRC1、SRC2和DST，则将SRC1和SRC2相乘，结果在DST中。n位×n位=n位(截断)

-

```
imulw %bx,%ax,%cx
```

除法指令：只明显指出除数

- 若为8位，则16位被除数在AX寄存器中，商送回AL，余数在AH
- 若为16位，则32位被除数在DX:AX寄存器中，商送回AX，余数在DX
- 若为32位，则被除数在EDX:EAX寄存器中，商送EAX，余数在EDX

```
movw $0x12,%dx
movw $0x23,%ax
movw  0x$80,%bx
divw  %bx
```

整数乘法指令举例

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

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

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

R[ax]=B4H × 11H（无符号整数180和17相乘）

R[ax]=0BF4H，真值为3060=180×17

“imulb %bl”功能为 R[ax]←R[al]×R[bl]

R[ax]=B4H × 11H（带符号整数-76和17相乘）

R[ax]=0BF4H？ 则真值为-76×17≠3060

R[al]=F4H, R[ah]=? AH中的值不一样！

R[ax]=FAF4H, 真值为-1292=-76×17

无符号乘：

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

AH=?　AL=?

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

假设下面的值存放在指定的内存地址和寄存器中：

地址	值
0x100	0xFF
0x108	0xAB
0x110	0x13
0x118	0x11

寄存器	值
%rax	0x100
%rcx	0x1
%rdx	0x3

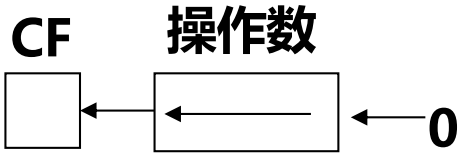
指令	目的	值
addq %rcx,(%rax)	内存地址:0x100	0x100
subq %rdx,8(%rax)	内存地址: 0x108	0xA8
imulq $16,(%rax,%rdx,8)	内存地址: 0x118	0x110
incq 16(%rax)	内存地址: 0x110	0x14
decq %rcx	%rcx	0x0
subq %rdx,%rax	%rax	0xfd

常用指令类型

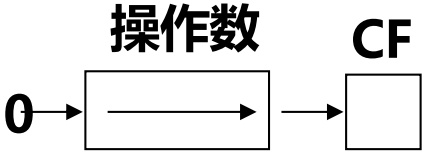
按位运算指令

- 按位布尔运算（仅NOT不影响标志，其他指令OF=CF=0，而ZF和SF根据结果设置：若全0，则ZF=1；若最高位为1，则SF=1 ）
  - not：反，包括 notb、notw、notl、notq　　C语言的~
  - and：与，包括 andb、andw、andl、andq　　C语言的&
  - or：或，包括 orb、orw、orl、orq　　C语言的|
  - xor：异或，包括 xorb、xorw、xorl、xorq　　C语言的^
- 移位运算（左/右移时，最高/最低位送CF）　C语言的<<和>>
  - shl/shr：逻辑左/右移，包括 shlb、shrw、shrl、shlq
  - sal/sar：算术左/右移，左移判溢出，右移高位补符
    （移位前、后符号位发生变化，则OF=1 ）
  - rol/ror: 循环左/右移，包括 rolb、rorw、roll 、rolq
  - rcl/rcr: 带循环左/右移，将CF作为操作数一部分循环移位

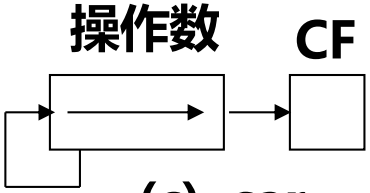
以上内容不要死记硬背，遇到具体指令时能查阅到并理解即可。



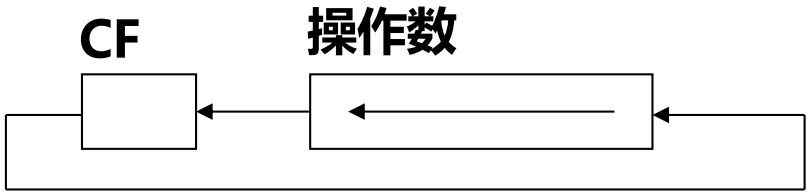
(a) sal(shl)



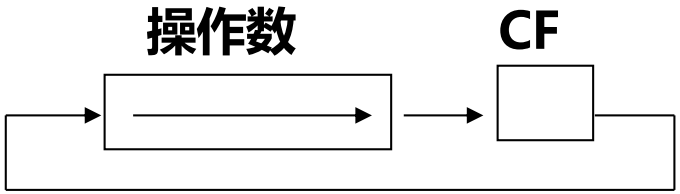
(b) shr



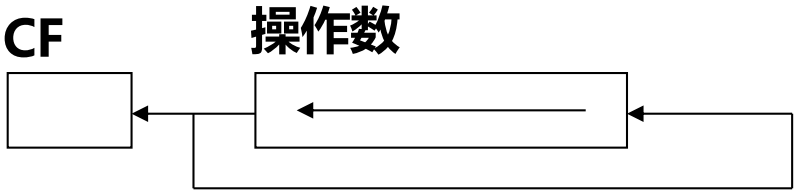
(c) sar



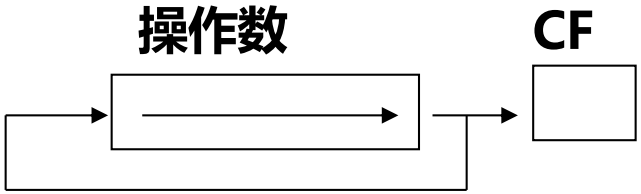
(d) rcl



(e) rcr



(f) rol



(g) ror

例：将AL寄存器的低四位置0，其余位保持不变。

andb $0xf0,%al

```
 10101010
&11110000
----------
 1010 0000
```

例：将BX寄存器的最高两位和最低两位置1，其余位保持不变。

orw $0xc003,%bx

```
 1010101010101010
|1100000000000011
----------------
 1110101010101011
```

例：将AH寄存器的所有位取反。

```
notb %ah
```

```
~ 1010 1010
-----------
  0101 0101
```

例：将BX寄存器的最高两位和最低两位取反，其余位保持不变。

```
xorw $0xc003,%bx
```

```
  1010101010101010
⊕ 1100000000000011
------------------
  0110101010101001
```

移位指令举例


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

```
                                    push   %ebp
                                    mov    %esp,%ebp
                                    and    $0xfffffff0,%esp
                                    sub    $0x20,%esp
00 00 00                            movl   $0x80000000,0x1c(%esp)

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

整数运算指令举例


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

```
arith:
  leaq    (%rdi,%rsi), %rax    # t1
  andq    %rdx, %rax           # t2
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx             # t4
  leaq    4(%rdi,%rdx), %rcx   # t5
  imulq   %rcx, %rax           # rval
  ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z, t4
%rax	t1, t2, rval
%rcx	t5

按位运算指令举例

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

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

解：$2和$1分别表示立即数2和1 。

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

逆向工程：从汇编指令推出高级语言程序代码（的实际功能）