程序的机器级表示 I：基础
Machine-Level Programming

章节要求

- 程序的编译过程
- 汇编特点：操作数，数据类型，运算
- X86-64的寄存器
- 多种指令要求熟记
- 内存寻址模式
- 经典例题

程序的机器级表示 I: 基础

- C, 汇编, 机器代码
- 汇编基础: 寄存器、操作数、数据传送、栈操作
- 算术和逻辑运算
- 比较和跳转

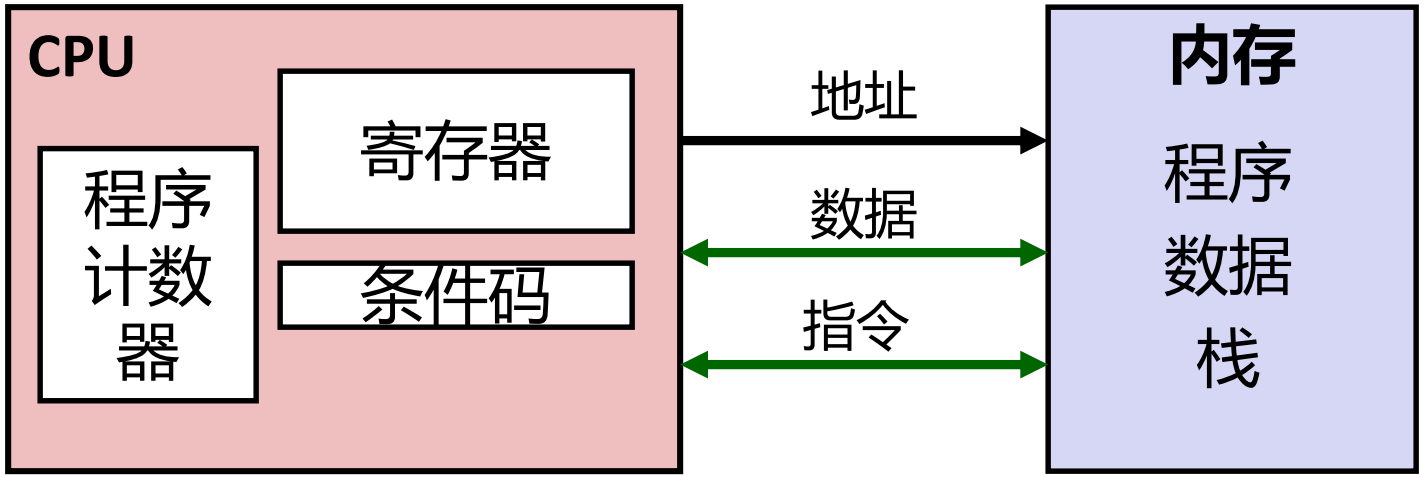
程序设计语言的特点

- **高级语言**
  - 抽象(Abstraction)
    - 编程效率高
    - 可靠
  - 拼写检查(Type checking)
  - 可在不同的机器上编译后运行
- **汇编语言**
  - 管理内存
  - 使用低级（底层）指令完成运算
  - 高度依赖机器

为什么要学习汇编?

- **为何要理解汇编代码**
  - 理解编译器的优化能力
  - 分析代码中潜在的低效性
  - 有时需要知道程序的运行时行为（数据）
- **为何要理解编译系统如何工作**
  - 优化程序性能
  - 理解链接时错误
  - 避免安全漏洞——缓冲区溢出
- **从写汇编代码到理解汇编代码**
  - 不同的技能：转换、源代码与汇编代码的关系
  - 逆向工程(Reverse engineering)
    - 直接从成品分析，获知产品的设计原理/过程

汇编/机器代码视图

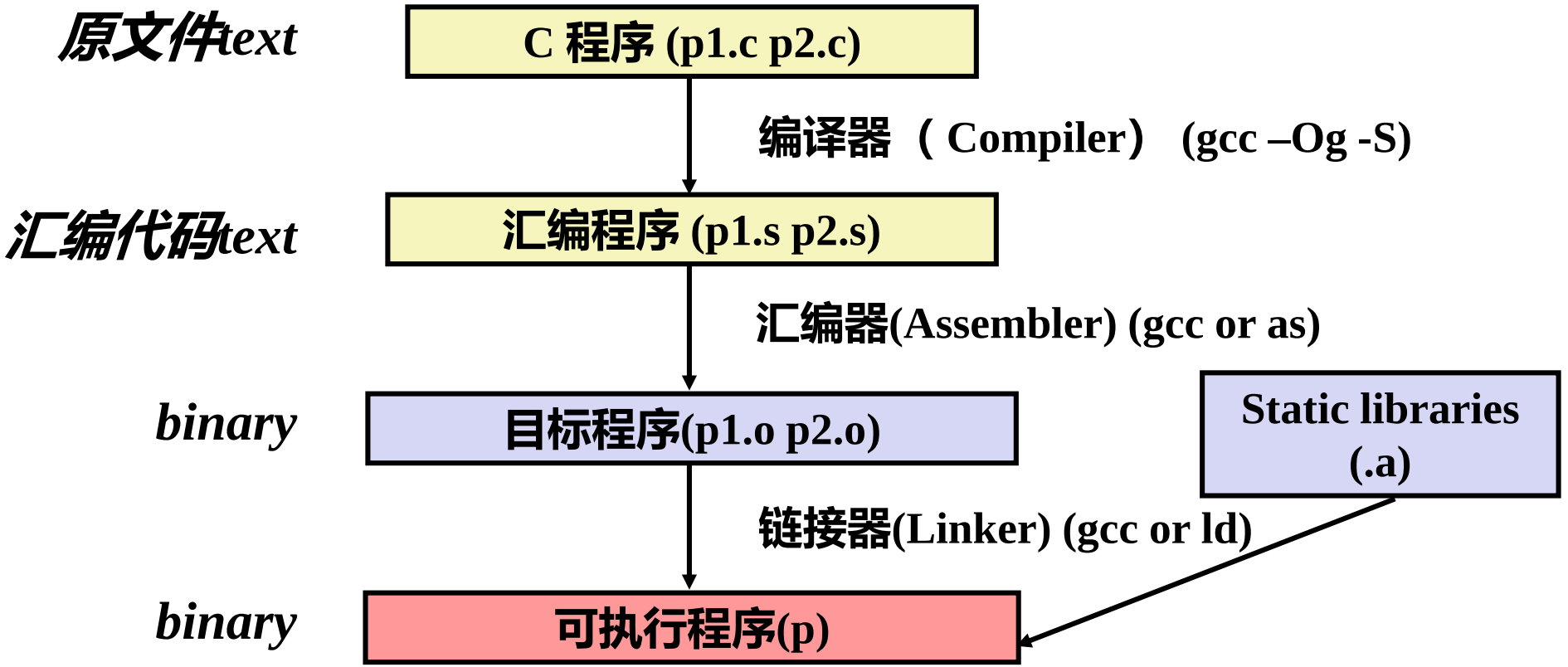


程序员可视的状态

- **程序计数器(Program counter，PC)**
  - 下一条指令的地址
  - 名字 EIP(IA32)、RIP (x86-64)
- **寄存器文件(Register file)**
  - 大量使用的程序数据
- **条件码(Condition codes)**
  - 存储最近的算术或逻辑运算的状态信息
  - 用于条件分支
- **内存(Memory)**
  - 可按字节寻址的数组
  - 程序和数据
  - 栈(Stack，用于过程的实现)

将 C 变为目标代码(Object Code)

- 程序文件: `p1.c p2.c`
- 编译命令: `gcc -Og p1.c p2.c -o p`
  - 使用基础优化项(`-Og`) [新版本GCC]
  - 生成二进制结果文件`p`



编译成汇编

C 代码

```
long plus(long x, long y);
void sumstore(long x, long y,
              long *dest)
{
    long t = plus(x, y);
    *dest = t;
}
```

生成的 x86-64汇编代码

```
sumstore:
    pushq   %rbx
    movq    %rdx, %rbx
    call    plus
    movq    %rax, (%rbx)
    popq    %rbx
    ret
```

使用的命令：

```
gcc -Og -S sum.c
```

生成文件： `sum.s`

gcc版本和选项的不同，生成的结果也会不同

备注：gcc的-Og选项提供了一些基本的优化，提高了代码的运行效率，但不会破坏代码的结构和变量的可见性，调试时仍然可以跟踪变量和代码逻辑。

目标代码

sumstore的机器码

```
0000000000400595 <sumstore>:
  400595:  53                  push   %rbx
  400596:  48 89 d3            mov    %rdx,%rbx
  400599:  e8 f2 ff ff ff      callq  400590 <plus>
  40059e:  48 89 03            mov    %rax,(%rbx)
  4005a1:  5b                  pop    %rbx
  4005a2:  c3                  retq
```

- **共14字节**
- **每个指令占1, 3, 或 5字节**
- **开始地址：0x0400595**

- **汇编器**
  - 将 .s 翻译成 .o
  - 指令的二进制编码
  - 几乎完整的可执行代码映像
  - 缺少不同文件代码之间的联系
- **链接器（第七章）**
  - 解析文件之间的引用
  - 与静态运行库相结合
    - 例如,malloc, printf的运行库
  - 动态链接库
    - 程序开始执行时，再进行链接

C 程序的构成

- **变量(Variable)**
  - 可定义并使用不同的数据类型
- **运算(Operation)**
  - 赋值、算术表达式计算
- **控制**
  - 循环
  - 过程（函数）的调用/返回

代码例子

```
//C code
int accum = 0;   //全局变量
int sum(int x, int y)
{
  int t = x+y;    //局部变量
  accum += t;
  return t;
}
```

代码例子（续） IA32汇编代码用栈保存参数

```
//C code
int accum = 0;
int sum(int x, int y)
{
  int t = x+y;
  accum += t;
  return t;
}
```

栈

y
x
返回地址
ebp

编译命令

gcc –O2 -S code.c

汇编文件 code.s

```
_sum:
        pushl %ebp
        movl %esp,%ebp
        movl 12(%ebp),%eax
        addl 8(%ebp),%eax
        addl %eax, accum
        movl %ebp,%esp
        popl %ebp
rbp     ret
```

指令

在IA32位汇编中：
ebp：存放调用函数前（旧的）ebp的值。
ebp+4：存放call指令调用函数时压入堆栈的返回地址。
ebp+8：存放函数传参的第一个参数地址
ebp+12: 存放函数传参的第二个参数地址。

32位汇编ebp、ebp+4、ebp+8

在IA32位汇编中：

- ebp：存放调用函数前（旧的）ebp的值。
- ebp+4：存放call指令调用函数时压入堆栈的返回地址。
- ebp+8：存放函数传参的第一个参数地址
- ebp+C：存放函数传参的第二个参数地址。
- ebp+10: .....

注意：不同于32位机器，X64的函数传参更先进，采用了寄存器的方式，分别是rdi，rsi，rdx，rcx，r8，r9，如果这六个参数寄存器不够用再考虑用栈存函数参数。

从C代码到汇编代码 IA32汇编代码用栈保存参数

- **汇编指令**
  - 执行一个具体明确的操作
- **两个有符号整型数相加**
  - C 代码:

    int t = x+y;
  - 汇编代码:

    addl 8(%ebp),%eax
    - 将两个4字节整型数相加
    - 类似C表达式 y += x

汇编特点：操作数

- **高级语言的操作数**
  - 常量、变量，例如：x = y + 4
- **汇编代码的操作数**
  - x: 寄存器　　%eax
  - y: 内存　　M[%ebp+8]--间接寻址
  - 4: 立即数　　$4
- **寄存器的特点**
  - 寄存器访问速度快
  - 数量少
  - 很多现代指令只能使用寄存器

汇编特点: 数据类型

- **整型数：1、2、 4 或8字节**
  - 数值
  - 汇编代码后缀：b、w、l、q
  - 地址（无类型指针）
- **浮点数：4(单精度)，8（双精度），or 10 bytes（扩展双精度）**
- **程序(Code):指令序列的字节编码串**
- **没有数组、结构体等聚合类型(aggregate types)**
  - 本质就是内存中连续分配的字节。

汇编特点: 运算

- **用寄存器、内存数据完成算术功能**
- **在内存和寄存器之间传送（拷贝）数据**
  - 从内存载入数据到寄存器
  - 将寄存器数据保存到内存
- **转移控制**
  - 无条件跳转到函数或从函数返回
  - 条件分支

机器指令示例

```
*dest = t;
```

```
movq %rax, (%rbx)
```

```
0x40059e:  48 89 03
```

- C 代码
  - 将数值t存到 dest指定的地方
- 汇编代码
  - 传送 8字节(Quad words)数值到内存
  - 操作数:
    - t:　寄存器　%rax
    - dest:　寄存器　%rbx
    - *dest:　内存　M[%rbx]
- 目标代码
  - 3字节的指令
  - 保存在地址0x40059e处

目标代码的反汇编

反汇编结果

```
0000000000400595 <sumstore>:
  400595:  53                  push   %rbx
  400596:  48 89 d3            mov    %rdx,%rbx
  400599:  e8 f2 ff ff ff      callq  400590 <plus>
  40059e:  48 89 03            mov    %rax,(%rbx)
  4005a1:  5b                  pop    %rbx
  4005a2:  c3                  retq
```

- **反汇编器/反汇编程序(Disassembler)**

  `objdump -d sum`

  - 检查目标代码的有用工具
  - 分析指令的位模式
  - 生成近似的汇编代码表述/译文
  - 可处理a.out (完整可执行文件)或 .o 文件

反汇编的另一种方法

Object

```
0x0400595:
    0x53
    0x48
    0x89
    0xd3
    0xe8
    0xf2
    0xff
    0xff
    0xff
    0x48
    0x89
    0x03
    0x5b
    0xc3
```

反汇编结果

```
Dump of assembler code for function sumstore:
 0x0000000000400595 <+0>: push   %rbx
 0x0000000000400596 <+1>: mov    %rdx,%rbx
 0x0000000000400599 <+4>: callq  0x400590 <plus>
 0x000000000040059e <+9>: mov    %rax,(%rbx)
 0x00000000004005a1 <+12>:pop    %rbx
 0x00000000004005a2 <+13>:retq
```

- 在调试器 gdb中反汇编sumstore
  - `gdb sum`
  - `disassemble sumstore`
  - `x/14xb sumstore`
  - 查看sumstore开始的14字节内容

什么可以被反汇编?

微软的终端用户许可协议中，
明确禁止逆向工程

```
% objdump -d WINWORD.EXE

WINWORD.EXE:    file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:  55              push   %ebp
30001001:  8b ec           mov    %esp,%ebp
30001003:  6a ff           push   $0xffffffff
30001005:  68 90 10 00 30  push   $0x30001090
3000100a:  68 91 dc 4c 30  push   $0x304cdc91
```

- 任何可执行代码
- 反汇编程序检查字节，并重构汇编资源

机器级程序设计I: 基础

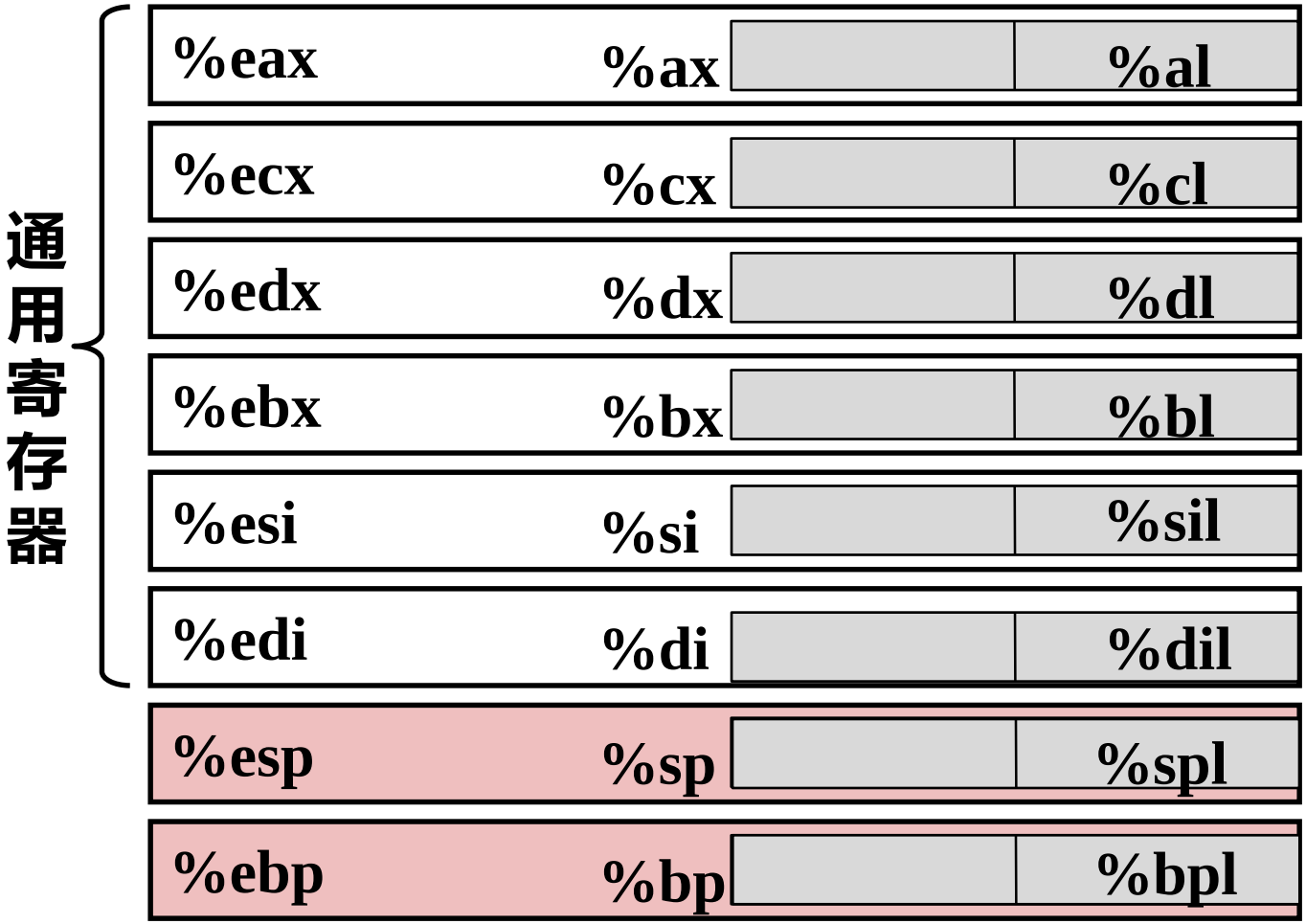
- C, 汇编, 机器代码
- **汇编基础: 寄存器、操作数、数据传送、栈操作**
- 算术和逻辑运算

x86-64 的整数寄存器

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

- 可使用低1、2、4字节

低四字节的寄存器


通用寄存器
%eax
%ax
%al
%ecx
%cx
%cl
%edx
%dx
%dl
%ebx
%bx
%bl
%esi
%si
%sil
%edi
%di
%dil
%esp
%sp
%spl
%ebp
%bp
%bpl

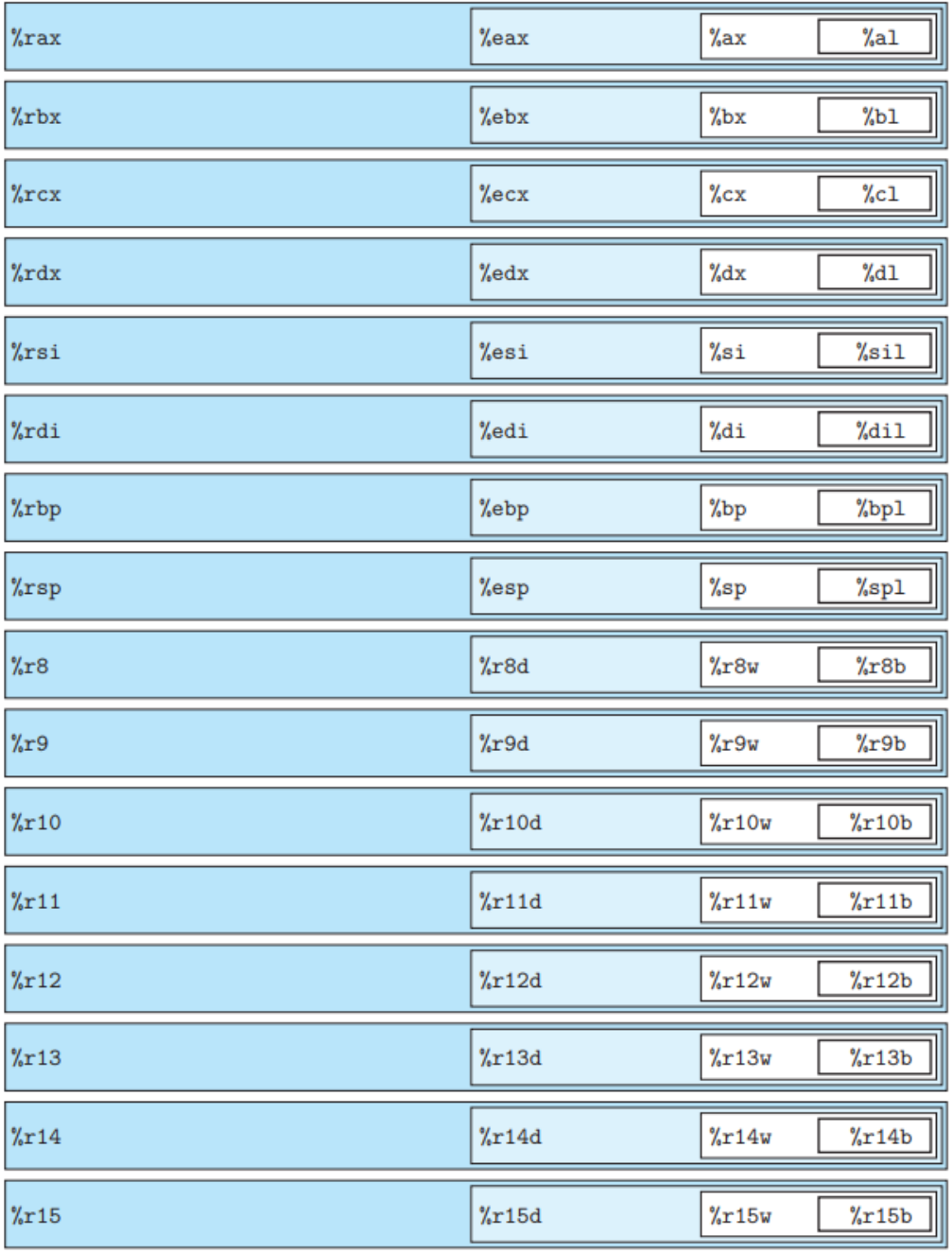
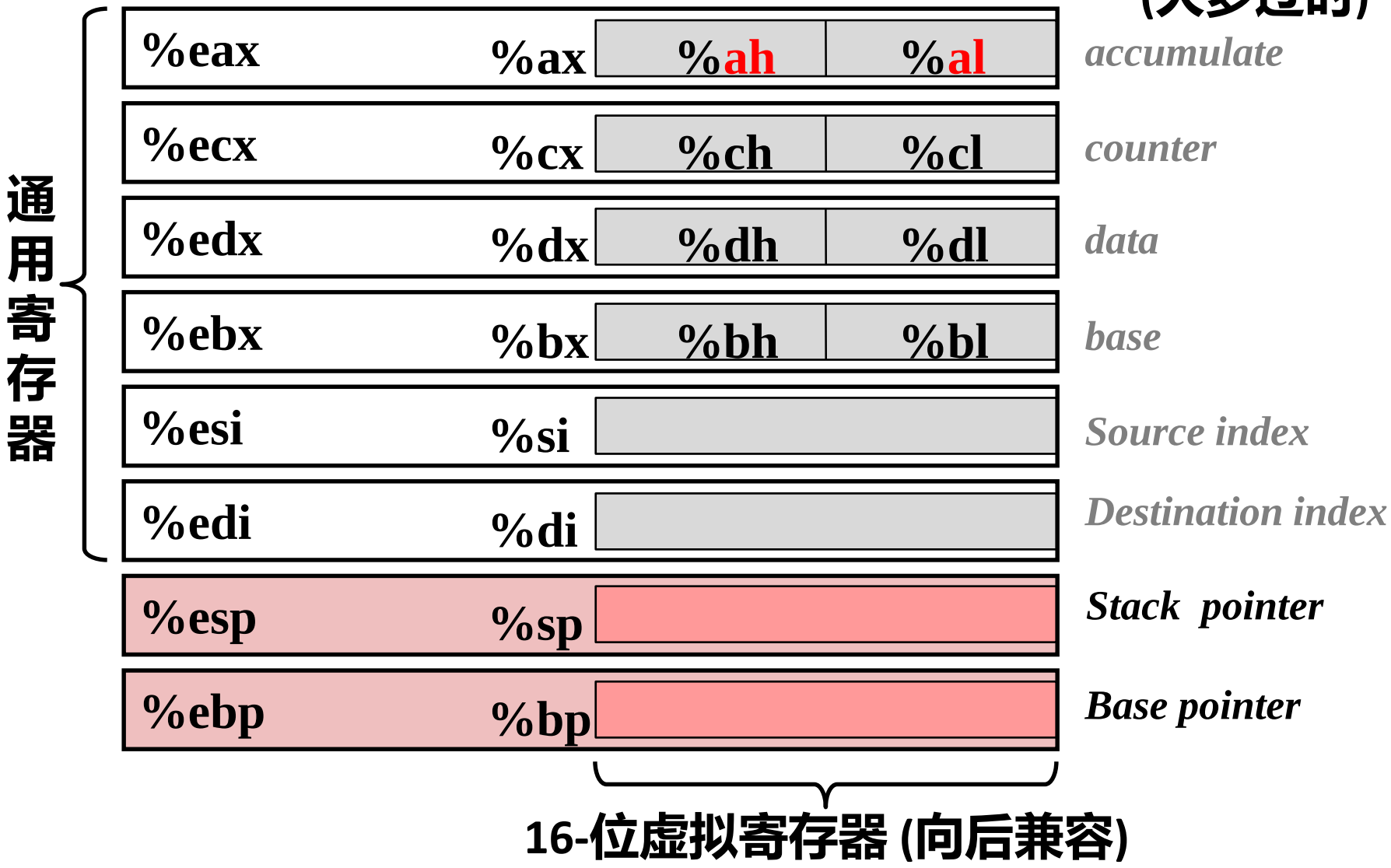


Figure 3.2 **Integer registers.** The low-order portions of all 16 registers can be accessed as byte, word (16-bit), double word (32-bit), and quad word (64-bit) quantities.

常见应用场景

- %rax：返回值
- %rbp：被调用者保存，常用于保存栈帧
- %rsp：栈指针，一般不能用作其他用途

历史: IA32的寄存器


来源
(大多过时)
通用寄存器
%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
16-位虚拟寄存器 (向后兼容)

AT&T汇编格式

- **操作数类型和表示**
  - ***立即数(Immediate)：*** 整型常数，以**$开头**
    - 例子: $0x400, $-533， $123
    - 类似 C的常数，但编码是1、 2 或 4 字节
  - ***寄存器(Register)：*** 加前缀%

    如：%eax, %ebx, %rcx, %r13
  - ***内存(Memory):*** 指定内存地址开始的连续字节，地址的指定方式有多种

    如：(%rax),4(%rsp,%rax),4(%rax,%rbx,2)
- **操作数顺序**
  - 多操作数指令，通常左边是src操作数，右边是dst操作数

    即运算结果向右传输 

AT&T汇编格式

- **操作数长度标识**
  - **整数操作数**

    b : 1字节、w :2 字节、l :4 字节、q :8字节
  - **浮点型操作数**

    s : 单精度浮点数、　　l :双精度浮点数
- **指令带操作数长度标识（如需要）**

数据传送

- **传送指令**

  mov*x*　*src, dst*

  ***x*：空白或b,w,l,q，分别对应1/2/4/8字节操作数**

- **操作数类型(三大类)**
  - ***立即数(Immediate)*：** 整型常数
  - ***寄存器(Register)*：** 16个整数寄存器之一
    - **不能用%rsp（系统保留）**
    - 其他特殊指令专用寄存器
  - ***内存(Memory)*:** 多种寻址模式

```
movb  $1,  %al
movw $1,  %ax
movl  $1,  %eax
movq  $1,  %rax
movq  $1,  %r8
```

mov 的操作数组合

	src	dst	源操作数，目的操作数	C 语言模拟
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;

单条指令不能进行从内存到内存的数据传送

RFLAGS寄存器的状态标志（条件码）

1. OF（Overflow Flag，溢出标志）

含义：表示有符号整数运算是否发生了溢出。
作用：当有符号整数的运算结果超出了当前数据类型所能表示的范围时，OF 被置为 1。例如，对于 8 位有符号整数，范围是 -128 到 127。如果计算结果超出了这个范围，OF 就会被置为 1。
应用场景：在进行有符号数的加减运算时，用于判断结果是否溢出。

2. ZF（Zero Flag，零标志）

含义：表示运算结果是否为 0。
作用：如果运算结果为 0，则 ZF 被置为 1；否则 ZF 为 0。
应用场景：常用于条件分支判断，例如在循环或条件语句中，根据 ZF 的值来决定是否继续执行。

3. SF（Sign Flag，符号标志）

含义：表示运算结果的符号。
作用：如果运算结果为负数（最高位为 1），则 SF 被置为 1；否则 SF 为 0。
应用场景：在处理有符号数时，用于判断结果的正负。例如，在比较两个有符号数的大小时，SF 的值可以帮助确定结果的符号。

4. CF（Carry Flag，进位标志）

含义：表示无符号整数运算是否产生了进位或借位。
作用：如果无符号整数的加法运算产生了进位，或者减法运算产生了借位，则 CF 被置为 1；否则 CF 为 0。
应用场景：在进行无符号数的加减运算时，用于判断是否需要进位或借位。例如，在多字节的加法运算中，CF 可以用于判断是否需要将进位传递到更高位。

5. PF（Parity Flag，奇偶标志）

含义：表示运算结果的最低字节中“1”的个数的奇偶性。

作用：如果“1”的个数为偶数，则 PF 被置为 1；否则 PF 为 0。

应用场景：主要用于奇偶校验。在数据传输或存储中，可以通过 PF 来检查数据的奇偶性，从而发现数据传输过程中可能出现的错误。

6. AF（Auxiliary Carry Flag，辅助进位标志）

含义：表示低 4 位（即低半字节）的进位或借位情况。

作用：如果低 4 位的运算产生了进位或借位，则 AF 被置为 1；否则 AF 为 0。

应用场景：主要用于二进制编码的十进制（BCD）运算。在 BCD 运算中，需要对低 4 位和高 4 位分别处理，AF 可以帮助判断低 4 位的进位情况

重要标志位总结

- **OF**：有符号数溢出标志
- **ZF**：零标志
- **SF**：符号标志
- **CF**：进位标志
- **PF**：奇偶标志
- **AF**：辅助进位标志

条件码(隐含赋值: Compare指令)

- **Compare指令对条件码的隐含赋值**
  - **cmpq** *Src1*, *Src2*

  cmpq *Src1*，*Src2* 计算*Src2 – Src1* 但不改变目的操作数，仅用结果设置条件码

- **cmpq *b*，*a* 注意顺序：*a-b*对标志位的影响**
  - **CF=1** 如果最高有效位有进位(无符号数比较)
  - **ZF=1** 如 a== b
  - **SF=1** 如 (a-b) < 0 (结果看做有符号数)
  - **OF=1** 如补码 (有符号数)溢出

```
(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)
```

数据传送

- **条件传送指令**

  cmovcc src, dst

  cc：表示条件

  src源: reg16, reg32, reg64

  dst目的: reg/mem16, reg/mem32, reg/mem64

  不支持单字节的条件传送

- ***在条件传送指令之前的指令（减法）会导致EFLAGS中的状态标志位发生变化。***
- ***利用EFLAGS中的CF、OF、PF、SF、ZF实现条件判断***
- ***经常和CMP b, a 命令配合使用***

什么是溢出

- 处理器内部以补码表示有符号数
- 8位表达的整数范围是：＋127～－128
- 16位表达的范围是：＋32767～－32768
- 如果运算结果超出这个范围，就产生了溢出
- 有溢出，说明有符号数的运算结果不正确

3AH＋7CH＝B6H，就是58＋124＝182，
已经超出－128～＋127范围，产生溢出，故OF＝1；
另一方面，补码B6H表达真值是-74，
显然运算结果也不正确

溢出和进位（重要）

- 溢出标志OF和进位标志CF是两个意义不同的标志
- 进位标志表示**无符号数**运算结果是否超出范围，运算结果仍然正确；
- 溢出标志表示**有符号数**运算结果是否超出范围，运算结果已经不正确。

溢出和进位的对比

例1：3AH + 7CH = B6H

无符号数运算：58 + 124 = 182

范围内，无进位

有符号数运算：58 + 124 = 182

范围外，有溢出

例2：AAH + 7CH＝（1）26H

无符号数运算：170＋124＝294

范围外，有进位

有符号数运算：－86＋124＝28

范围内，无溢出

数据传送（教材P147）

- ***无符号数的条件传送***
  - *用a、b、e、n、c分别表示：大于、小于、等、否、进位*
  - *CPU用当前的状态标志CF、ZF、PF实现判别*

CMOVA/CMOVNBE	大于/不小于且不等于	**CF=0 & ZF=0**
CMOVAE/CMOVNB	大于或者等于/不小于	**CF = 0**
CMOVNC	无进位	**CF = 0**
CMOVB/CMOVNAE	小于/不大于且不等于	**CF = 1**
CMOVC	进位	**CF = 1**
CMOVBE/CMOVNA	小于或者等于/不大于	**CF=1 | ZF=1**

数据传送

- ***无符号数的条件传送***

CMOVE/CMOVZ	等于/零	ZF = 1
CMOVNE/CMOVNZ	不等于/不为零	ZF = 0
CMOVP/CMOVPE	奇偶校验	PF = 1

例子：

cmova %ebx,%eax

cmoval %ebx, %eax 不需要指定长度，编译器可以通过目的寄存器的名字识别操作数长度。

数据传送

- ***有符号数的条件传送***
  - *用g、l、e、n、o分别表示：大于、小于、等、否、溢出*
  - *CPU用SF、ZF、OF实现判别*

CMOVG/CMOVNLE	大于/不小于等于	**ZF=0 & (SF^OF)=0**
CMOVGE/CMOVNL	大于等于/不小于	**SF^OF = 0**
CMOVL/CMOVNGE	小于/不大于等于	**SF^OF = 1**
CMOVLE/CMOVNG	小于等于/不大于	**SF^OF=1 | ZF = 1**
CMOVO	溢出	**OF = 1**
CMOVNO	末溢出	**OF = 0**
CMOVS	负数	**SF = 1**
CMOVNS	非负数	**SF = 0**

```
cmovge   %r8,   %r9
cmovgeq  %r9, %r10
cmovgl   %r8d,%r10d
cmovll   %r8d,%r10d
```

指令	同义名	传送条件	描述
cmove S, R	cmovz	ZF	相等/零
cmovne S, R	cmovnz	~ZF	不相等/非零
cmovs S, R		SF	负数
cmovns S, R		~SF	非负数
cmovg S, R	cmovnle	~(SF ^ OF) & ~ZF	大于（有符号>）
cmovge S, R	cmovnl	~(SF ^ OF)	大于或等于（有符号>=）
cmovl S, R	cmovnge	SF ^ OF	小于（有符号<）
cmovle S, R	cmovng	(SF ^ OF) | ZF	小于或等于（有符号<=）
cmova S, R	cmovnbe	~CF & ~ZF	超过（无符号>）
cmovae S, R	cmovnb	~CF	超过或相等（无符号>=）
cmovb S, R	cmovnae	CF	低于（无符号<）
cmovbe S, R	cmovna	CF | ZF	低于或相等（无符号<=）

图 3-18 条件传送指令。当传送条件满足时，指令把源值 S 复制到目的 R。有些指令是“同义名”，即同一条机器指令的不同名字

例如：

C: long cread(long *xp){

return (xp?*xp:0);

}

汇编：

```
cread:
movq   (%rdi),%rax
testq    %rdi,%rdi          #检查是负数/0/正数
movl   $0,%edx
cmove    %rdx,%rax
ret
```

注意：cmov系列指令是寄存器到寄存器传数，见教材P147

数据传送

- **扩展传送指令**
  - 符号扩展的传送

    movsbl S, D

    SignedExtend(S)➔D
  - 零扩展的传送

    movzbl S, D

    ZeroExtend(S) ➔D

  其中bl代表将单字节扩展到四字节
- ***初始值：%rax=0xfa4, %rbx=0x7654321012345678***

  movsbl %al, %ebx # %rbx=0xff ff ff a4

  movzbl %al, %ebx # %rbx=0xa4

数据传送的例子

扩展的数据传送：

初始值：%dl=8d　　%eax =0x98765432

movb　　%dl, %al　　%eax=0x9876548d

movsbl　%dl, %eax　　%eax=0xffffff8d

movzbl　%dl, %eax　　%eax=0x0000008d

数据传送的操作数：

movl　$0x4050, %eax　　immediate　register

movl　%ebp, %esp　　register　register

movl　(%edx, %ecx), %eax　memory　register

movl　$-17, (%esp)　　immediate　memory

movl　%eax, -12(%ebp)　　register　memory

分的大小必须与指令最后一个字符('b'，'w'，'l'或'q')指定的大小匹配。大多数情况中，MOV 指令只会更新目的操作数指定的那些寄存器字节或内存位置。唯一的例外是 movl 指令以寄存器作为目的时，它会把该寄存器的高位 4 字节设置为 0。造成这个例外的原因是 x86-64 采用的惯例，即任何为寄存器生成 32 位值的指令都会把该寄存器的高位部分置成 0。

数据传送的例子

备注：美元符号$放在常数前面代表常量，放变量前面表示变量的地址
$varx：表示变量 varx 的地址
varx：表示变量 varx 的值

varx:

```
.int 124,-2345 #124地址是varx，-2345地址是varx+4

movl  $-1, %eax          # %eax = 0xffffffff  = -1
movq $varx, %rax     # %rax = 0x4000ac
movl  varx, %ebx       # %rbx = 0x7c = 124
movq  varx+4, %rcx   #%rcx =0xfffffffffffff6d7=-2345
movl (%rax), %edx   # %rdx = 0x7c = 124
```

特别注意：

```
movl  $-1, %rax          # %rax = 0x00000000ffffffff
```

理解一个问题

分的大小必须与指令最后一个字符('b'，'w'，'l'或'q')指定的大小匹配。大多数情况中，MOV 指令只会更新目的操作数指定的那些寄存器字节或内存位置。唯一的例外是 movl 指令以寄存器作为目的时，它会把该寄存器的高位 4 字节设置为 0。造成这个例外的原因是 x86-64 采用的惯例，即任何为寄存器生成 32 位值的指令都会把该寄存器的高位部分置成 0。

- **数据传送如何改变目的寄存器**
  - movabsq $0x0011223344556677 %rax #绝对四字 %rax = 0x0011223344556677
  - movb $-1,%al #%rax = 0011223344556FF
  - movw $-1,%ax #%rax = 001122334455FFFF
  - movl $-1,%eax #%rax = 00000000FFFFFFFF
  - movq $-1,%rax #%rax = FFFFFFFFFFFFFFFF
- **可以看出movl的特殊**
- **这是因为规定movl：生成四字节值并以寄存器为目的指令会把高四字节置为0.**
- **为什么会有movabsq：处理64位立即数，常规的mov指令只能处理32位立即数。**

简单的内存寻址模式

- **寄存器间接寻址（常用）**

  形式：　(R)　　含义：Mem[Reg[R]]

  - 寄存器R指定内存地址
  - 比较：C语言的指针解引用

  ```
  movq (%rcx),%rax
  ```

- **相对寻址**

  形式：　D(R)　　含义：Mem[Reg[R]+D]

  - 寄存器R指定内存区域的开始地址
  - D: 常数位移量 “displacement”， 1, 2, 4, 8字节指定偏移值(offset)

  ```
  movq 8(%rbp),%rdx
  ```

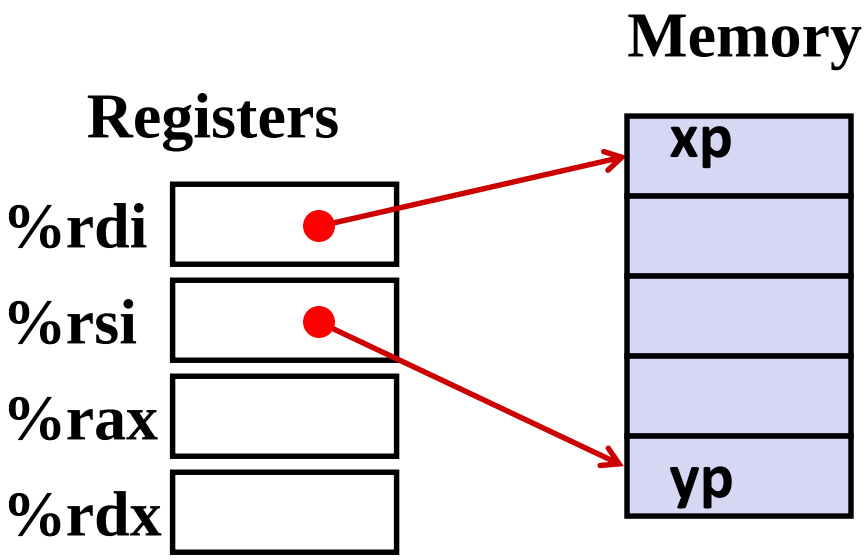
寻址模式例子

```
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
```

理解Swap()

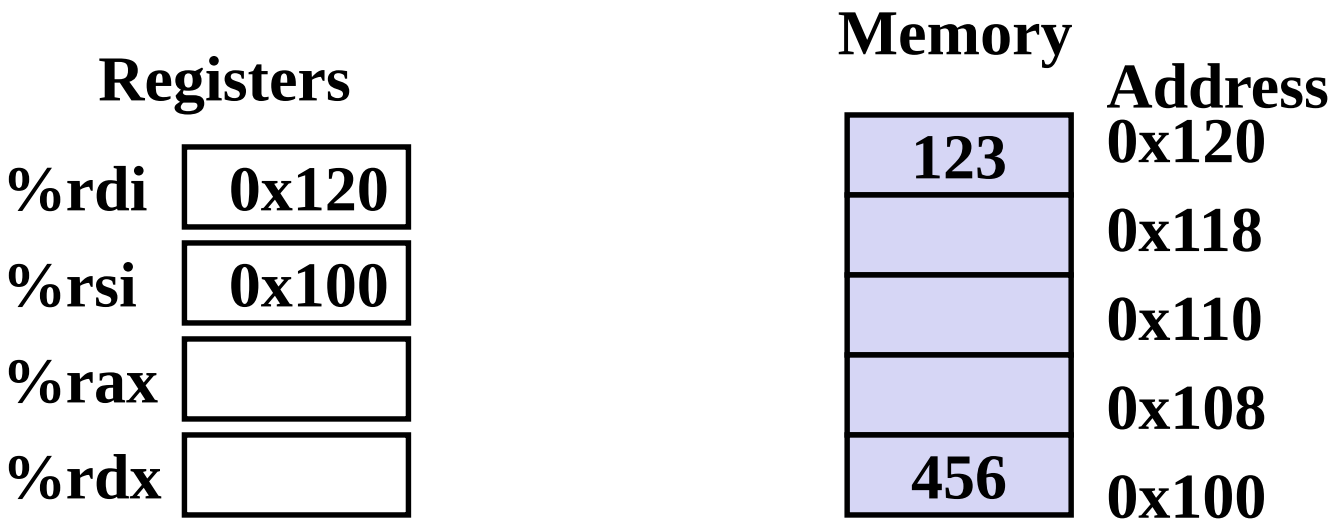
```
void swap
   (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



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

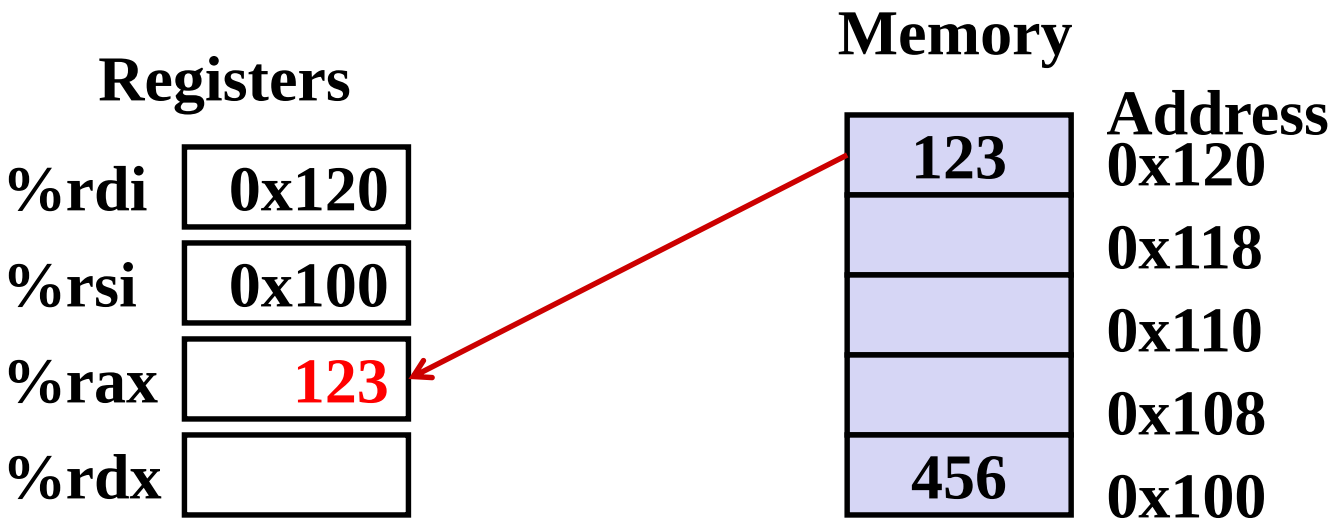
```
swap:
   movq    (%rdi), %rax  # t0 = *xp
   movq    (%rsi), %rdx  # t1 = *yp
   movq    %rdx, (%rdi)  # *xp = t1
   movq    %rax, (%rsi)  # *yp = t0
   ret
```

理解Swap()



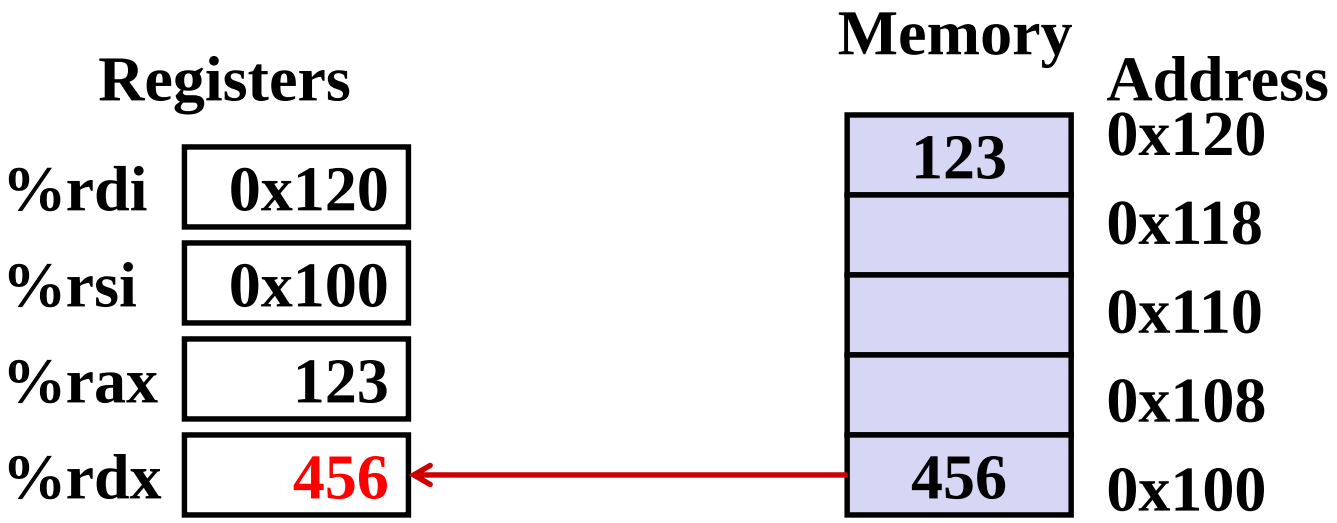

```
swap:
   movq    (%rdi), %rax  # t0 = *xp
   movq    (%rsi), %rdx  # t1 = *yp
   movq    %rdx, (%rdi)  # *xp = t1
   movq    %rax, (%rsi)  # *yp = t0
   ret
```

理解Swap()



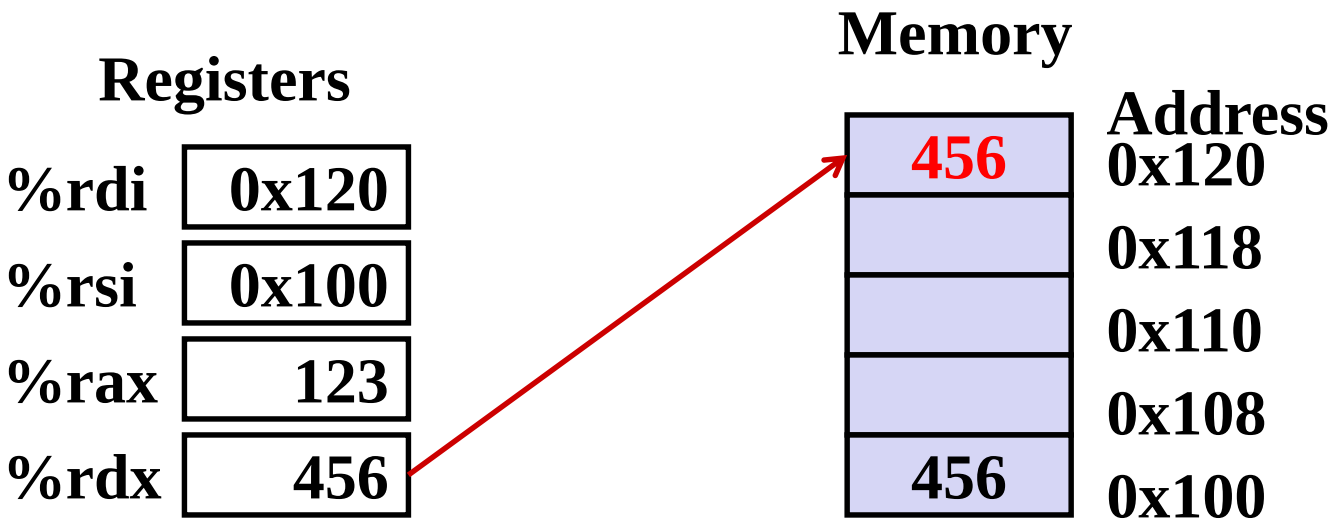

```
swap:
   movq    (%rdi), %rax  # t0 = *xp
   movq    (%rsi), %rdx  # t1 = *yp
   movq    %rdx, (%rdi)  # *xp = t1
   movq    %rax, (%rsi)  # *yp = t0
   ret
```

理解Swap()



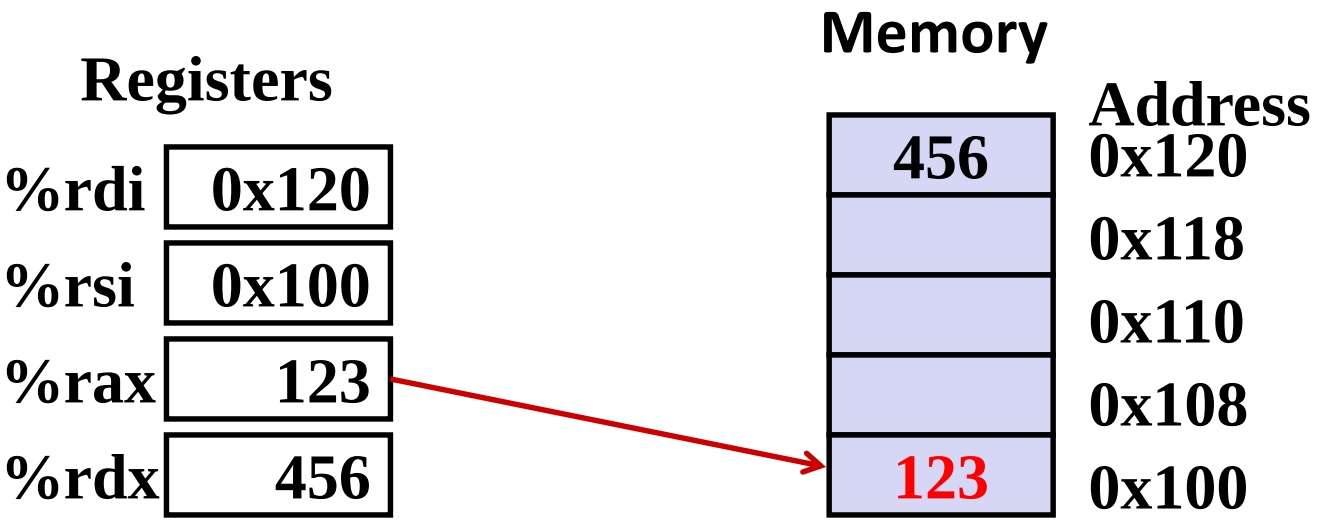

```
swap:
   movq    (%rdi), %rax  # t0 = *xp
   movq    (%rsi), %rdx  # t1 = *yp
   movq    %rdx, (%rdi)  # *xp = t1
   movq    %rax, (%rsi)  # *yp = t0
   ret
```

理解Swap()




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

理解Swap()




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

完整的内存寻址模式

- **最一般形式:　D(Rb,Ri,S)**

  含义：Mem[Reg[Rb]+S*Reg[Ri]+ D]

  索引化的寻址方式

  - D——常量，表示位移量(displacement): 1, 2, or 4 字节
  - Rb——基址寄存器(Base register)：所有16位整数寄存器
  - Ri——变址寄存器(Index register): 不可用`%rsp`
  - S ——比例因子(Scale): 1, 2, 4, or 8 (*why these numbers?*)

- **特殊情况**

(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]]
D(,Ri,S)	Mem[D+Reg[Ri]*S]

地址计算例子：

%rdx	0xf000
%rcx	0x0100

表达式	地址计算	地址
0x8(%rdx)	0xf000 + 0x8	0xf008
(%rdx,%rcx)	0xf000 + 0x100	0xf100
(%rdx,%rcx,4)	0xf000 + 4*0x100	0xf400
0x80(,%rdx,2)	0x80 + 2*0xf000	0x1e080

地址	值
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

寄存器	值
%eax	0x100
%ecx	0x1
%edx	0x3

操作数	值
%eax	0x100
(%eax)	0xFF
$0x108	0x108
0x108	0x13
260(%ecx,%edx)	0x13
(%eax,%edx,4)	0x11

$符号也可以用于表示常量

栈操作

- 除了直接的数据传送操作，还可以将数据压入程序栈中，或从栈中弹出数据。
- 栈是一种数据结构，遵循后进先出的原则。
- 注意程序栈是自栈底向下扩展。
- %rsp保存栈顶元素的地址。

栈操作

- 入栈指令 pushq S

效果：R[%rsp] <-- R[%rsp] - 8

M[%rsp] <-- S

将S压入栈中

- 出栈指令 popq D

效果：D <-- M[%rsp]

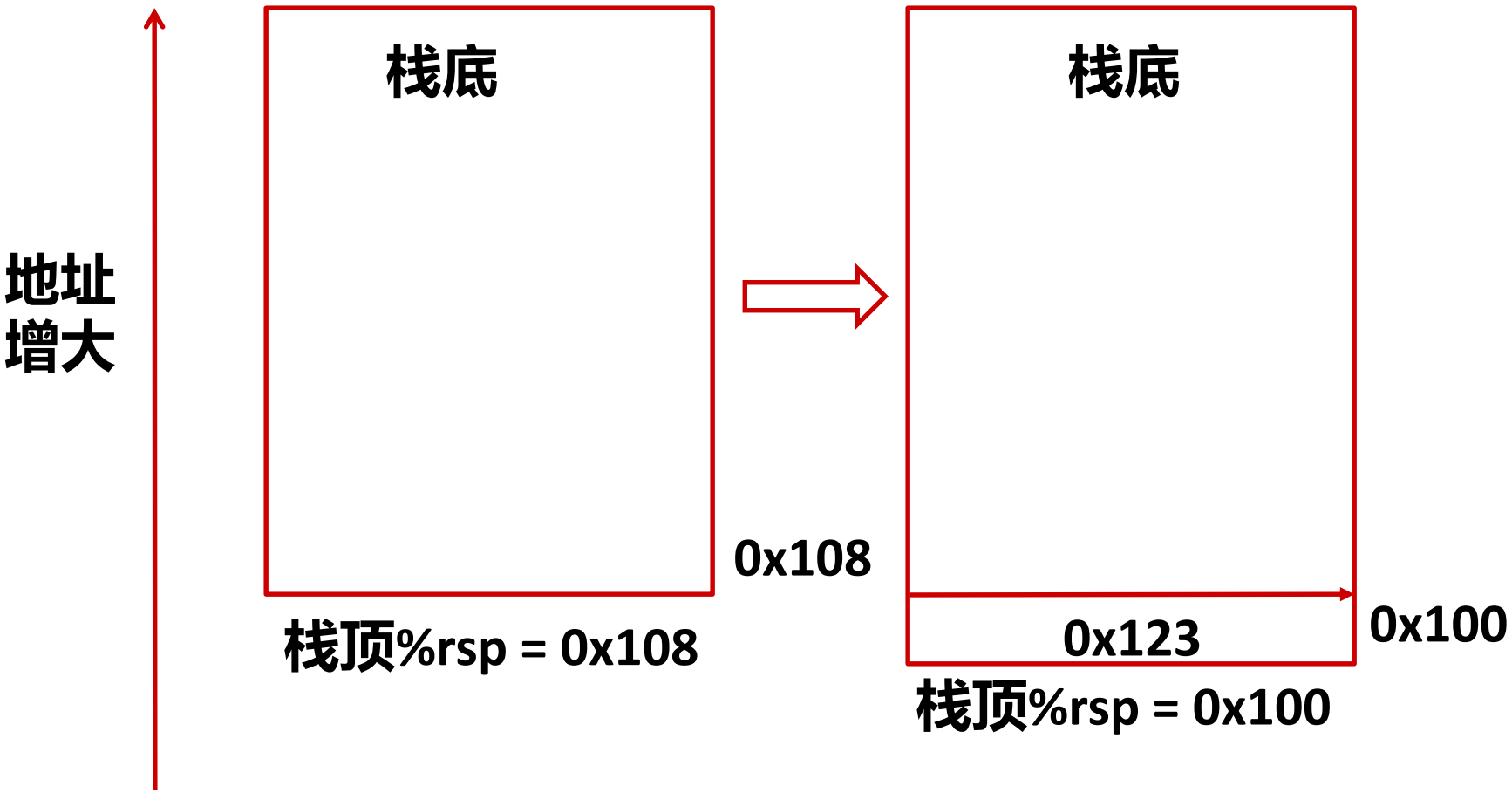
R[%rsp] <-- R[%rsp] + 8

将栈顶元素出栈并保存至D。

栈操作

- 入栈示例

%rax = 0x123
pushq %rax



机器级程序设计I: 基础

- Intel CPU及架构的发展史
- IA32处理器体系结构
- C, 汇编, 机器代码
- 汇编基础: 寄存器、操作数、数据传送、栈操作
- **算术和逻辑运算**

取地址指令（load effective address）

- `leaq` ***Src*, *Dst***
  - *Src* 源地址模式表达式。
  - 将表达式对应的地址保存到*Dst*中。
  - 注意不涉及内存引用，也不会改变标志位。
- **用法**
  - **不涉及内存引用，计算地址**
    - 例如, 翻译语句 `p = &x[i];`
  - 计算形如x + k*y的算术表达式
    - k = 1, 2, 4, or 8
- **Example**

C代码

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

编译器生成

```
leaq (%rdi,%rdi,2), %rax    # t ← x+x*2 (即3x)
salq $2, %rax               # return t<<2；即(3x*4)
```

算术运算指令

- 注意参数顺序!
- 有/无符号数整数之间没有差别
- 影响的标志：CF、ZF、SF、OF（PF，AF）

- 2操作数指令:

格式		运算
addq	Src,Dest	#Dest = Dest + Src
subq	Src,Dest	# Dest = Dest – Src
imulq	Src,Dest	# Dest = Dest * Src
salq	Src,Dest	# Dest = Dest << Src算术左移，同shlq 逻辑左移效果相同，编译器常用左移和加法替代乘法
sarq	Src,Dest	# Dest = Dest >> Src算术右移(高位补符号位)
shrq	Src,Dest	# Dest = Dest >> Src逻辑右移(高位补0)
xorq	Src,Dest	# Dest = Dest ^ Src 按位异或
andq	Src,Dest	# Dest = Dest & Src 按位与
orq	Src,Dest	# Dest = Dest | Src 按位或

算术运算指令

- **单操作数指令**

```
incq    Dest        # Dest = Dest + 1
decq    Dest        # Dest = Dest – 1
negq    Dest        # Dest = – Dest (取负/补，即相反数的补码)
notq    Dest        # Dest = ~Dest (取非/反，即按位取反)
imulq/mulq   Dest           # 特殊的乘运算，
```

Dest*R[%rax]

结果的高位保存在%rdx中，低位（真正的结果）保存在%rax中。前缀i代表有符号乘，去掉i代表无符号乘

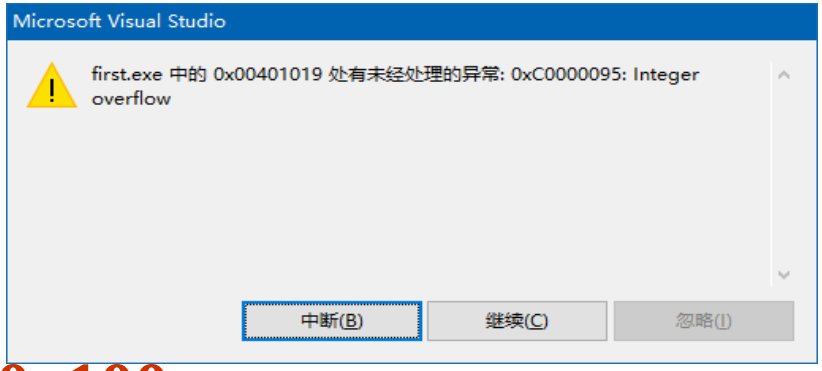
```
idivq/divq      Dest        # 特殊的除运算，
```

R[%rdx]:R[%rax]/Dest

结果的商保存在%rax中，余数保存在%rdx中。前缀i代表有符号除，去掉i代表无符号除

除法溢出

- DIV/IDIV执行后，所有算术状态标志均不确定！
- 除法的商太大，目的操作数无法容纳→除法溢出；
- **除法溢出→ CPU触发中断，终止程序运行。**




```
movw $0x1000，%ax
movb $0x10,%bl
divb %bl # AL 无法容纳结果0x100
```

```
(gdb) s

Program received signal SIGFPE, Arithmetic exception.
_start () at try64.s:35
3: /x $rbx = 0x10
2: /x $rax = 0x1000
```

除0运算,除0错误。

如何防止?

用更多位数的除法、检查除数确定不为0

算术表达式例子

```
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
  addq    %rdx, %rax
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx
  leaq    4(%rdi,%rdx), %rcx
  imulq   %rcx, %rax
  ret
```

- leaq: 取地址
- salq: 移位
- imulq: 乘,仅用了一次

算术表达式例子

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

寄存器	用途
%rdi	参数x
%rsi	参数y
%rdx	参数z
%rax	t1, t2, rval
%rdx	t4
%rcx	t5

机器级程序设计I: 基础

- C, 汇编, 机器代码
- 汇编基础: 寄存器、操作数、数据传送
- 算术和逻辑运算
- **比较和跳转**

布尔和比较指令

- **布尔指令**
  - AND、OR、XOR、NOT
  - TEST
- **比较指令**
  - CMP
- **条件跳转指令**

  JCond
- **CPU的状态标志：ZF、SF、CF、OF（PF、AF）**

2.1 布尔指令——AND

- **AND指令**
  - AND指令在每对操作数的对应数据位之间执行布尔位"与"操作，并将结果存放在目的操作数中：

    AND 源操作数，目的操作数

    AND reg/mem/imm, reg

    AND reg/imm, mem

总是使得CF=0、OF=0
依据目的操作数的值修改SF、ZF和PF的值

2.1 布尔指令——AND

- **具体应用举例——字符大小写转换**
  - 分析：
    - 'a'　(61h)　= 0110 0001b
    - 'A'（41h ）　= 0100 0001b

```
and   $0X20, arrayElem
      ；保留字符元素的第6位，
      ；以确定其是大写还是小写
```

2.1 布尔指令——OR

- OR指令在每对操作数的对应数据位之间执行布尔位“或”操作，并将结果存放在目的操作数中：

  OR 源操作数, 目的操作数

  OR reg/mem/imm, reg

  OR reg/imm, mem

总是使得CF=0、OF=0
依据目的操作数的值修改SF、ZF和PF的值

2.1 布尔指令——XOR

- XOR指令在每对操作数的对应数据位之间执行布尔位“异或”操作，并将结果存放在目的操作数中：

```
XOR  源操作数,目的操作数
XOR   reg/mem/imm, reg
XOR   reg/imm, mem
```

总是使得CF=0、OF=0
依据目的操作数的值修改SF、ZF和PF的值

2.1 布尔指令——XOR

- **利用XOR指令的特性实现简单的数据加密**
  - 特性：对数值进行两次“异或”操作后其操作效果将被抵消；

$$(X \oplus Y) \oplus Y = X$$

2.1 布尔指令——NOT

- NOT指令将一个操作数的所有数据位取反：

  NOT reg

  NOT mem

NOT指令不修改任何状态标志

```
movb      $0xf0, %al
notb   %al
#al = 0x0f=00001111b
```

2.1 布尔指令——TEST

- TEST指令：执行隐含的“与”操作，并相应设置标志位（ZF，SF）。
- TEST指令不修改目的操作数；
- 指令格式和AND指令相同；
- 测试操作数某位或某几位是否被设置时特别有用！还被应用于验证操作数的符号。
- 当所有测试位都为0时，ZF=1

```
movb    $0x0fe, %al
testb   $0x2e,  %al
```

条件码(隐含赋值: Compare指令)

- **Compare指令对条件码的隐含赋值**
  - **cmpq** *Src1*, *Src2*

  cmpq *Src1*，*Src2* 计算*Src2 – Src1*（**右边减左边**），但不改变目的操作数，仅用结果设置条件码

- **cmpq *b*，*a* 注意顺序：*a-b*对标志位的影响**
  - **CF=1** 如果最高有效位有进位(无符号数比较)
  - **ZF=1** 如 a== b
  - **SF=1** 如 (a-b) < 0 (结果看做有符号数)
  - **OF=1** 如补码 (有符号数)溢出

```
(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)
```

2.2 比较指令—— CMP

- CMP指令执行隐含的减法操作:目的操作数-源操作数（注意顺序），并设置标志位，但不保存减法的结果，两个操作数都不会被修改：

CMP 源操作数,目的操作数

格式与SUB相同，修改OF、SF、ZF、CF、AF和PF

可以理解为比较大小。

2.2 比较指令—— CMP

- CMP指令后，操作数大小判别方法（利用标志位）

无符号数大小判别

CMP的结果	ZF	CF
目的 < 源	0	1
目的 > 源	0	0
目的 = 源	1	0

有符号数大小判别

CMP的结果	标志
目的 < 源	SF≠OF
目的 > 源	SF = OF
目的 = 源	ZF = 1

语法： CMP 源操作数，目的操作数

无条件跳转指令

- jmp 直接跳转到操作数位置。
- 如：

```
movq $0,%rax
jmp .L1
movq (%rax),%rdx #不会执行
.L1:...
```

条件跳转指令

- **跳转依据**
  - 操作数之间是否相等
  - 根据比较结果
    - 无符号操作数
    - 有符号操作数

2.4 条件跳转指令

- 根据相等比较的跳转指令
  - JE/JNE
  - JE表示等于则跳转
  - JNE表示不等于则跳转

例3：机器状态监测与重置

```
movb    status, %al
testb   $0x8c, %al
cmpb    $0x8c,  %al
je ResetMachine
```

2.4 条件跳转指令

- **无符号数比较**
  - JA/JNA(>/<=)
  - JAE/JNAE(>=/<)
  - JB/JNB(</>=)
  - JBE/JNBE(<=/>)

```
.data
        v1:  .short 1
        v2:  .short 2
        v3:  .short 3
.text
        movw        v1,  %ax
        cmpw        v2,  %ax
        jbe     L1
        movw        v2,  %ax
L1:     cmpw        v3,  %ax
        jbe     L2
        movw        v3, %ax
L2:
```

2.4 条件跳转指令

- **有符号数比较**
  - JG/JNG(>/<=)
  - JGE/JNGE(>=/<)
  - JL/JNL(</>=)
  - JLE/JNLE(<=/>)

机器级编程I: 小结

- **Intel CPU及架构的发展史**
  - 进化设计导致许多怪癖和假象
- **IA32处理器体系结构**
- **C, 汇编, 机器代码**
  - 可视状态的新形式: 程序计数器、 寄存器 ...
  - 编译器必须将高级语言的声明、表达式、过程(函数)翻译成低级(底层)的指令序列
- **汇编基础： 寄存器、 操作数、 数据传送**
  - x86-64的传送指令涵盖了广泛的数据传送形式
- **算术运算**
  - C 编译器将使用不同的指令组合完成计算

经典例题

1.下列叙述正确的是（ ）

A.一条mov指令不可以使用两个内存操作数

B.在一条指令执行期间，CPU不会两次访问内存

C.CPU不总是执行CS::RIP所指向的指令，例如遇到call、ret指令时

D.X86-64指令"movl $1,%eax"不会改变%rax的高32位

答案：A 考点：各种指令的用法，见教材P122

B:执行期间如取指，写回。

C:必须要依靠程序计数器指向要跳转的地址才能完成跳转。

D:movl对64位的高四字节置0。

本章主要以选择、填空或分析题考查

经典例题

2.阅读的sum函数反汇编结果中编号①-⑤的代码，解释每行指令的功能和作用

```
00000000004004e7 <sum>:
 4004e7:  55              push  %rbp      #①
 4004e8:  48 89 e5        mov   %rsp,%rbp #②
 4004eb:  c7 45 fc 00 00 00 00 movl  $0x0,-0x4(%rbp) #③
 4004f2:  eb 1e           jmp  400512 <sum+0x2b>
 4004f4:  8b 45 fc          mov  -0x4(%rbp),%eax
 4004f7:  48 98           cltq
 4004f9:  8b 14 85 30 10 60 00 mov  0x601030(,%rax,4),%edx
 400500:  8b 05 3e 0b 20 00    mov  0x200b3e(%rip),%eax #601044 <val>
 400506:  01 d0           add  %edx,%eax
 400508:  89 05 36 0b 20 00    mov  %eax,0x200b36(%rip) #601044 <val>
 40050e:  83 45 fc 01        addl  $0x1,-0x4(%rbp)
 400512:  83 7d fc 03        cmpl  $0x3,-0x4(%rbp)#④
 400516:  7e dc           jle   4004f4 <sum+0xd>#⑤
 400518:  8b 05 26 0b 20 00    mov  0x200b26(%rip),%eax # 601044 <val>
 40051e:  5d              pop  %rbp
 40051f:  c3              retq
```

①入栈指令，将rbp入栈

②传送指令，将栈顶指针rsp的值传送给rbp

③传送指令，向%rbp-4的内存位置传送数值0（局部变量i赋初值0）

④比较指令：%rbp-4的内存数值（局部变量i的值）与3进行比较 （i<4吗）

⑤条件跳转指令，小于等于则跳转（跳转到4004f4处） （i<4则循环）

经典例题

2.阅读的sum函数反汇编结果中编号①-⑤的代码，解释每行指令的功能和作用

```
00000000004004e7 <sum>:
 4004e7:      55              push  %rbp      #①
 4004e8:      48 89 e5        mov   %rsp,%rbp #②
 4004eb:      c7 45 fc 00 00 00 00 movl  $0x0,-0x4(%rbp) #③
 4004f2:      eb 1e           jmp  400512 <sum+0x2b>
 4004f4:      8b 45 fc          mov  -0x4(%rbp),%eax
 4004f7:      48 98           cltq
 4004f9:      8b 14 85 30 10 60 00 mov  0x601030(,%rax,4),%edx
 400500:      8b 05 3e 0b 20 00    mov  0x200b3e(%rip),%eax #601044 <val>
 400506:      01 d0           add  %edx,%eax
 400508:      89 05 36 0b 20 00    mov  %eax,0x200b36(%rip) #601044 <val>
 40050e:      83 45 fc 01        addl  $0x1,-0x4(%rbp)
 400512:      83 7d fc 03        cmpl  $0x3,-0x4(%rbp)#④
 400516:      7e dc           jle   4004f4 <sum+0xd>#⑤
 400518:      8b 05 26 0b 20 00    mov  0x200b26(%rip),%eax # 601044 <val>
 40051e:      5d              pop  %rbp
 40051f:      c3              retq
```

代码分析

1. **栈帧设置**
   - `4004e7: push %rbp` ：将旧的基指针寄存器（ `rbp` ）压入栈，保存当前函数调用的上下文。
   - `4004e8: mov %rsp, %rbp` ：将栈指针（ `rsp` ）的值移动到基指针寄存器（ `rbp` ），建立新的栈帧。
2. **初始化变量**
   - `4004eb: movl $0x0, -0x4(%rbp)` ：在栈上分配一个局部变量，初始化为 0。这个变量可以被视为循环计数器 `i` 。
3. **跳转到循环条件检查**
   - `4004f2: jmp 400512 <sum+0x2b>` ：跳转到循环条件检查的位置。
4. **循环体**
   - `4004f4: mov -0x4(%rbp), %eax` ：将局部变量 `i` 的值加载到寄存器 `eax` 。
   - `4004f7: cltq` ：将 `eax` 的值扩展为 64 位，存储到 `rax` 。
   - `4004f9: mov 0x601030(,%rax,4), %edx` ：通过 `i` 计算数组的第 `i` 个元素的地址，并将其值加载到寄存器 `edx` 。
   - `400500: mov 0x200b3e(%rip), %eax` ：加载全局变量 `val` 的值到寄存器 `eax` 。
   - `400506: add %edx, %eax` ：将数组元素的值加到 `val` 的值上。
   - `400508: mov %eax, 0x200b36(%rip)` ：将结果存回全局变量 `val` 。
   - `40050e: addl $0x1, -0x4(%rbp)` ：将局部变量 `i` 的值加 1。
5. **循环条件检查**
   - `400512: cmpl $0x3, -0x4(%rbp)` ：比较局部变量 `i` 是否小于等于 3。
   - `400516: jle 4004f4 <sum+0xd>` ：如果 `i` 小于等于 3，跳转回循环体的开始位置。
6. **循环结束**
   - `400518: mov 0x200b26(%rip), %eax` ：加载全局变量 `val` 的值到寄存器 `eax` 。
   - `40051e: pop %rbp` ：恢复旧的基指针寄存器。
   - `40051f: retq` ：返回到调用者。

函数的功能

从上述代码可以看出，函数 `sum` 的功能是：

1. 初始化一个局部变量 `i` 为 0。
2. 使用一个循环，从数组中逐个取出元素，将其值加到全局变量 `val` 上。
3. 循环执行 4 次（ `i` 从 0 到 3）。
4. 最终返回全局变量 `val` 的值。

C 语言源代码

C

```c
#include <stdio.h>

// 假设有一个全局变量 val
int val = 0;

// 假设有一个全局数组 arr
int arr[] = {1, 2, 3, 4};

int sum() {
    int i = 0;  // 局部变量 i，用于循环计数
    for (i = 0; i < 4; i++) {
        val += arr[i];  // 将数组元素的值加到全局变量 val 上
    }
    return val;  // 返回全局变量 val 的值
}

int main() {
    printf("Sum: %d\n", sum());  // 调用 sum 函数并打印结果
    return 0;
}
```

Enjoy!