

程序的机器级表示 I: 基础

Machine-Level Programming

教师：郑贵滨

计算机科学与技术学院

哈尔滨工业大学

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
- 算术和逻辑运算

课程内容

■ IA32

- 传统x86
- ...> gcc -m32 hello.c

■ x86-64

- 标准
- ...> gcc hello.c
- ...> gcc -m64 hello.c

程序设计语言的特点

■ 高级语言

- 抽象 (Abstraction)
 - 编程效率高
 - 可靠
- 类型检查
- 与手写代码同样高效
- 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？

■ 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）。

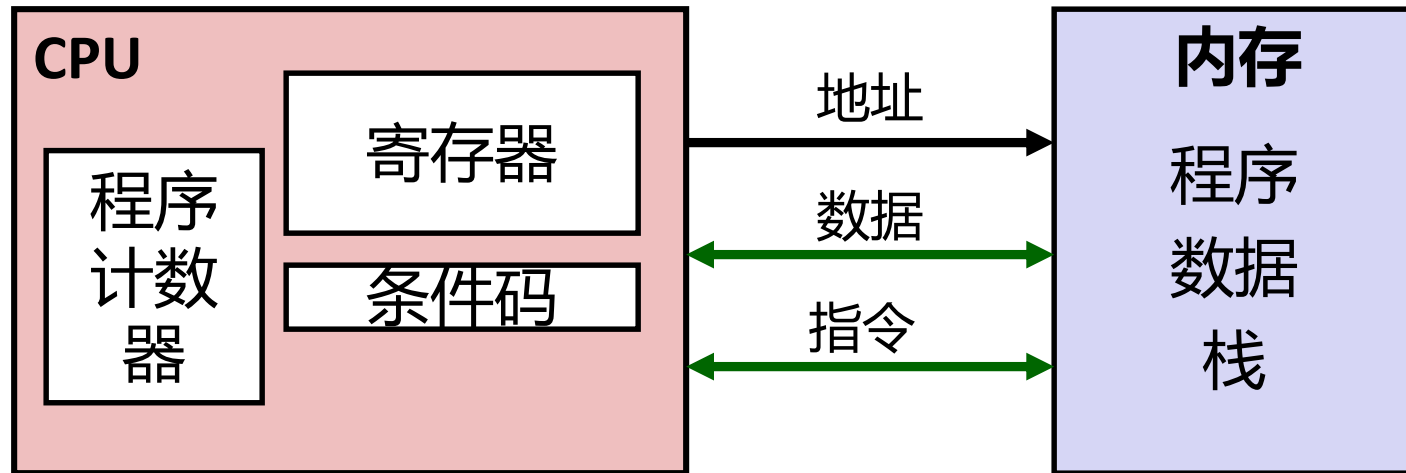
■ 为何要理解编译系统如何工作

- 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

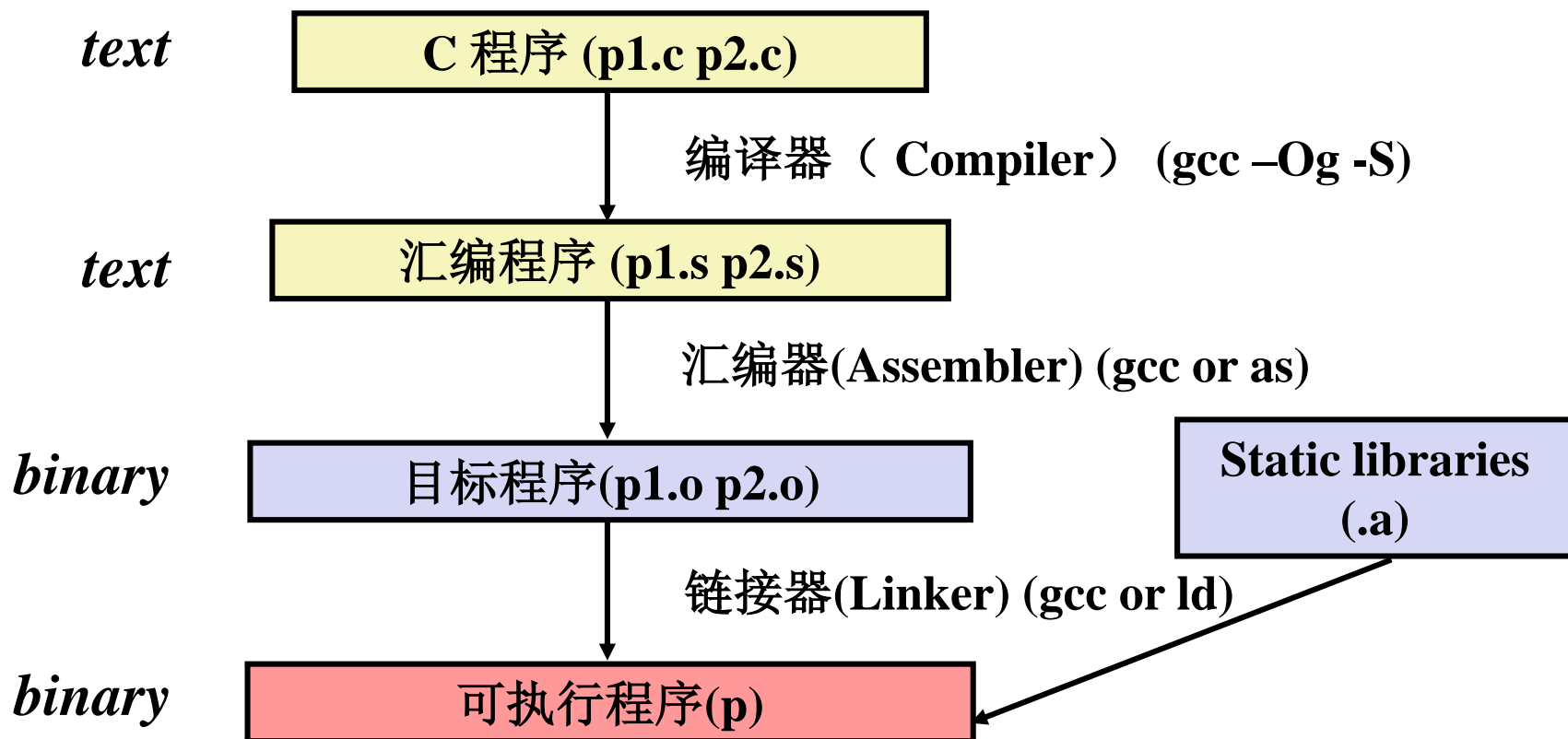


程序员可视的状态

- **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



编译成汇编

C 代码 (sum.c)

```
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

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

代码例子

//C code

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

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
    ret
```

指令



从C代码到汇编代码

■ 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 $x += y$

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]
- 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
 - 地址（无类型指针）
- **浮点数: 4, 8, or 10 bytes**
- **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)**
 - 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

目标代码

sumstore的代码

0x0400595:

0x53

0x48

0x89

0xd3

0xe8

0xf2

0xff

0xff

0xff

0x48

0x89

0x03

0x5b

0xc3

- 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

■ 连接器

- 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, 在进行链接

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
- 算术和逻辑运算

x86-64 的整数寄存器

%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

- 可使用低1、2、4字节

历史: IA32的寄存器

来源
(大多过时)

通用寄存器

%eax	%ax	%ah	%al	<i>accumulate</i>
%ecx	%cx	%ch	%cl	<i>counter</i>
%edx	%dx	%dh	%dl	<i>data</i>
%ebx	%bx	%bh	%bl	<i>base</i>
%esi	%si			<i>Source index</i>
%edi	%di			<i>Destination index</i>
%esp	%sp			<i>Stack pointer</i>
%ebp	%bp			<i>Base pointer</i>

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

■ 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

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

`mov b $1, $\%al$`

`mov w $1, $\%ax$`

`mov l $1, $\%eax$`

`mov q $1, $\%rax$`

`mov q $1, $\%r8$`

$\%rax$

$\%rbx$

$\%rcx$

$\%rdx$

$\%rsi$

$\%rdi$

$\%rsp$

$\%rbp$

$\%rN$

mov 的操作数组合

	源	目的	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

数据传送

■ 条件传送指令

cmovcc src, dst

cc: 表示条件

src: *r16, r32, r64*

dst: *r/m16, r/m32, r/m64*

■ 利用EFLAGS中的CF、OF、PF、SF、ZF实现条件判断

数据传送

■ 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 CF 、 ZF 、 PF 实现判别

CMOVA/CMOVNBE	大于/小于或者不等于	(CF或者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	CF = 1
CMOVC	进位	CF = 1
CMOVBE/CMOVNA	小于或者等于/不大于	(CF或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

cmova! %ebx, %eax

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 and SF=OF)
CMOVGE/CMOVNLE	大于等于/不小于	(SF异域OF) = 0
CMOVL/CMOVNGE	小于/不大于等于	(SF民域OF) = 1
CMOVLE/CMOVNG	小于等于/不大于	((SF异域OF)或ZF) = 1
CMOVO	溢出	OF = 1
CMOVNO	未溢出	OF = 0
CMOVS	带符号(负)	SF = 1
CMOVNS	无符号(非负)	SF = 0

```

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

数据传送

■ 扩展传送指令

■ 符号扩展的传送

`movsbl / movsbq S, D`

`SignedExtend(S) → D`

■ 零扩展的传送

`movzbl / movzbq S, D`

`ZeroExtend(S) → D`

`mov $0xfa4, %rax # %rax=0xfa4`

`movabsq 0x8877665544332211, %rbx # %rbx=0x8877665544332211`

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

`# %rbx=?, 0xff ff ff a4`

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

★: 如指令将4字节值存到32位寄存器, 会把寄存器的高4字节置0

数据传送的例子

初始值: %dh=0x8d %eax = 0x98765432

movb %dh, %al %eax=0x9876548d

movsbl %dh, %eax %eax=0xffffffff8d

movzbl %dh, %eax %eax=0x0000008d

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

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, or 4 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;  
}
```

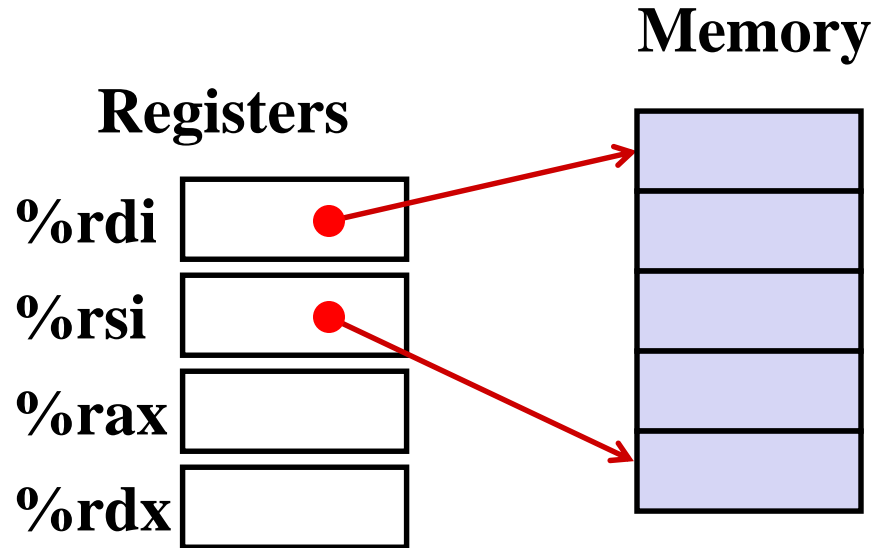
```
swap:  
  movq    (%rdi), %rax  
  movq    (%rsi), %rdx  
  movq    %rdx, (%rdi)  
  movq    %rax, (%rsi)  
  ret
```

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()

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

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

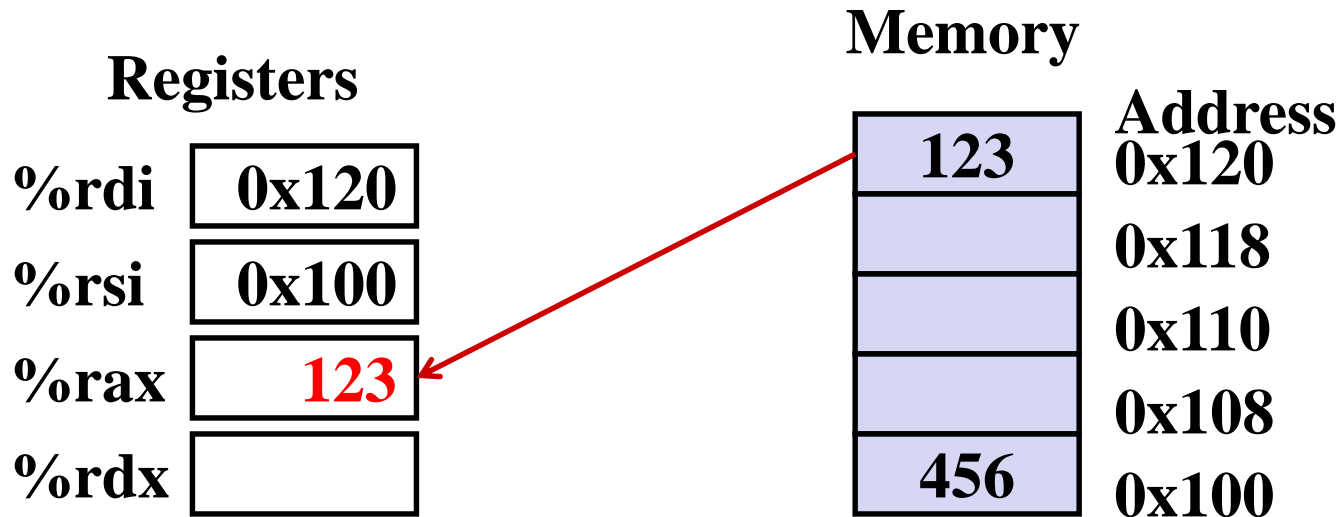
swap:

```

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

```

理解Swap()



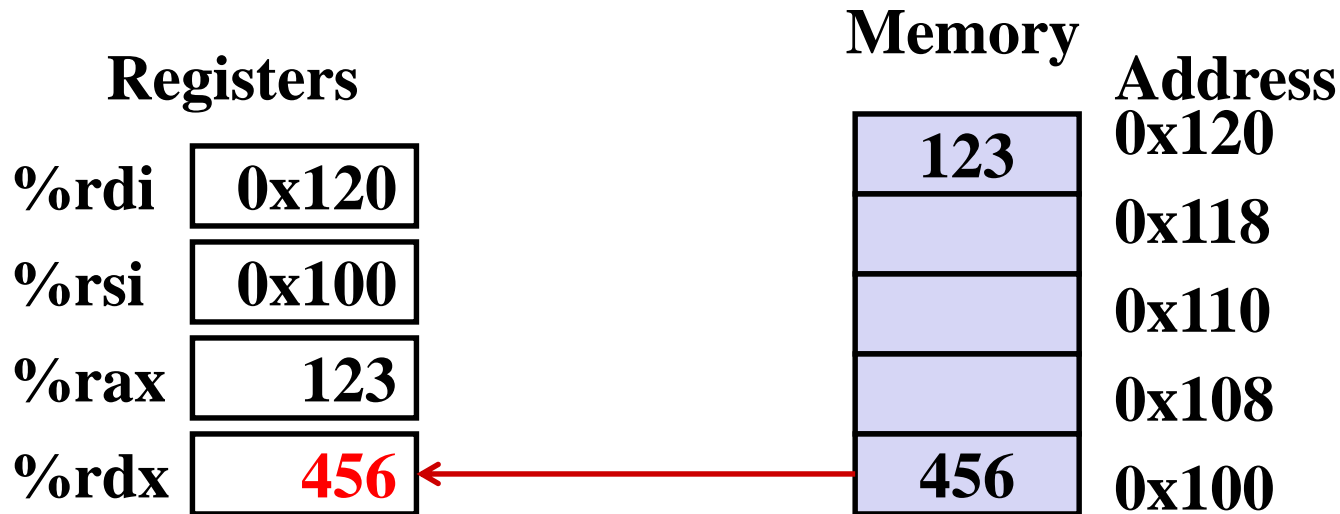
swap:

```

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

```

理解Swap()



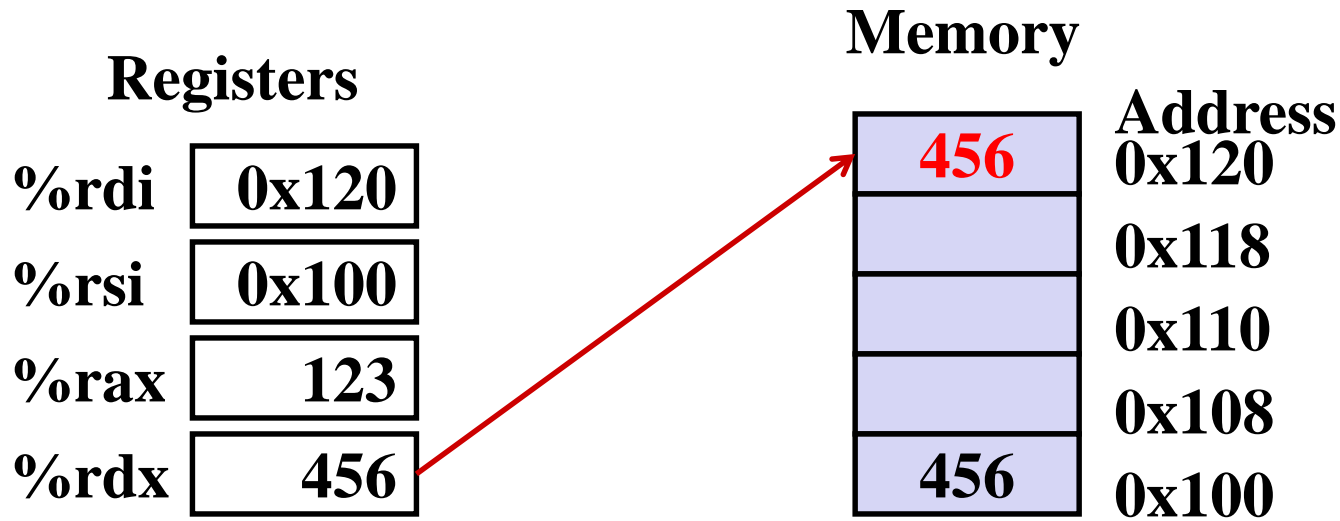
swap:

```

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

```

理解Swap()



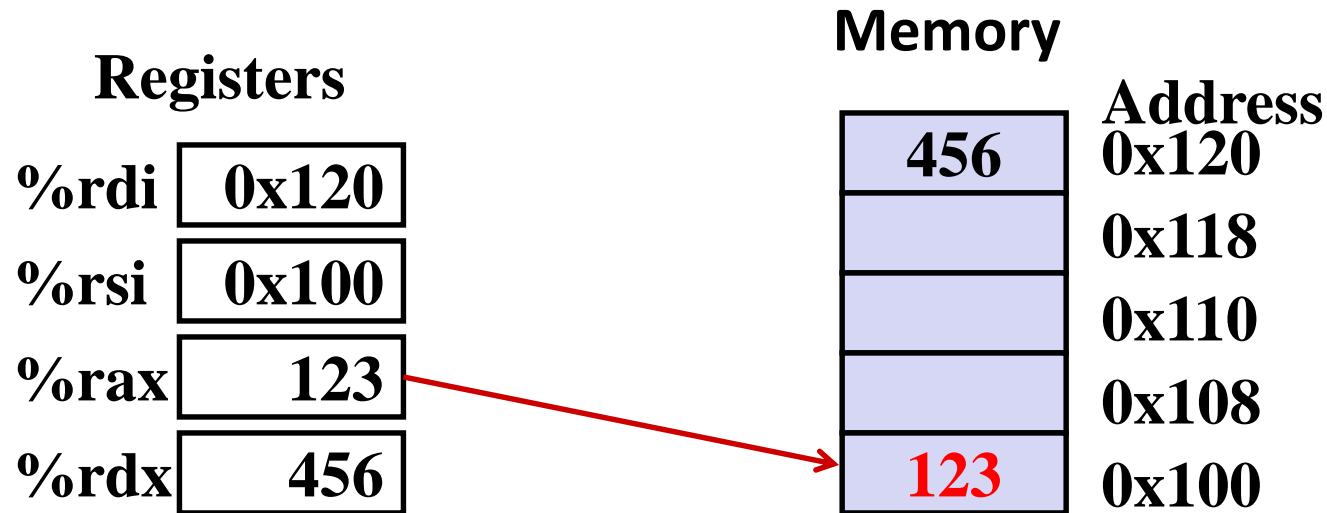
swap:

```

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): 任意16个整数寄存器
- 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)	$Mem[Reg[Rb] + S * Reg[Ri]]$

数据传送的例子

■ 全局变量定义：

```
.data
```

```
plnt: .quad 0 # $varx
```

```
varx: .int 124, -2345, 0x34, 0x1234
```

```
vary: .int 1, 2, 3, 4
```

■ 汇编指令

```
mov $-1, %rax      # %rax = 0xffffffff ffffffff = -1
```

```
movq $varx, %rax   # %rax = 0x6005b0 6292912
```

```
mov varx, %ebx     # %rbx = 0x7c = 124
```

```
mov varx+4, %ecx   # %rcx = 0xff ff f6 d7 != -2345
                  # %ecx = -2345
```

```
mov (%rax), %edx   # %rdx = 0x7c = 124
```

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)	(0x108)0x13
(%eax,%edx,4)	(0x10C)0x11

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
- 算术和逻辑运算

取地址指令

■ `leaq Src, Dst`

- `Src` 地址模式表达式
- 将表达式对应的地址保存到 `Dst` 中

■ 用法

- 不引用内存，仅计算地址
 - 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;
}
```

编译器生成的ASM

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

算术运算指令

- 注意参数顺序!
- 有/无符号数整数之间没差别(why?)

■ 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 同 shlq
sarq	Src, Dest	# Dest = Dest >> Src 算术移位
shrq	Src, Dest	# Dest = Dest >> Src 逻辑移位
xorq	Src, Dest	# Dest = Dest ^ Src
andq	Src, Dest	# Dest = Dest & Src
orq	Src, Dest	# Dest = Dest Src

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
notq	Dest	# Dest = ~Dest

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

- 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
- 算术和逻辑运算

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

Linux汇编程序——两种格式的语法对比

■ 两种汇编格式: AT&T 汇编、Intel汇编

■ 1、寄存器前缀%

AT&T: %eax

Intel: eax

■ 2、源/目的操作数顺序

AT&T: movl %eax,%ebx

Intel: mov ebx,eax

■ 3、常数/立即数的格式 \$

AT&T: movl \$_value, %ebx #把变量_value的地址放入ebx

movl \$0xd00d, %ebx

Intel: mov eax, offset _value

mov ebx,0xd00d

■ 4、操作数长度标识:b-1字节, w-2 字节, l-4 字节,q-8字节

AT&T: movw var_x, %bx

Intel: mov bx, word ptr var_x

Linux汇编程序——两种格式的语法对比

■ 5、寻址方式

AT&T: $\text{imm32}(\text{basepointer}, \text{indexpointer}, \text{indexscale})$

Intel: $[\text{basepointer} + \text{indexpointer} * \text{indexscale} + \text{imm32}]$

Linux工作于保护模式下，使用32位线性地址，计算地址时不用考虑segment:offset的问题，上式地址为：

$\text{imm32} + \text{basepointer} + \text{indexpointer} * \text{indexscale}$

(1) 直接寻址

AT&T: `movl $0xd00d, var` # var是一个全局变量

注意：`$var`表示变量地址引用，`var`表示变量值引用

Intel: `mov var, 0xd00d ; mov [var], 0xd00d`

(2) 寄存器间接寻址

AT&T :

`movl (%ebx), %eax`

`movl 3(%ebx), %eax`

Intel :

`mov eax, [ebx]`

`mov eax, [ebx+3]`

`mov eax, 3[ebx]`

Linux汇编程序——两种格式的语法对比

(3) 变址寻址

AT&T: `movl %ecx, var(%eax)`
 `movl %ecx, array(,%eax,4)`
 `movl %ecx, array(%ebx,%eax,8)`

Intel: `mov [eax + var], ecx`
 `mov [eax*4 + array], ecx`
 `mov [ebx + eax*8 + array], ecx`

■ 嵌入式汇编

```
asm( "pushl %eax\n\t"
      "movl $0,%eax\n\t"
      "popl %eax");
asm("movl %eax,%ebx");
asm("xorl %ebx,%edx");
asm("movl $0,_booga);
```