概述

计算机系统基础

任课教师：
龚奕利
yiligong@whu.edu.cn

本节内容

- Intel 处理器的历史与架构
- C 代码，汇编代码，机器级代码
- 汇编语言基础：寄存器，操作数，move 数据传送指令
- 算术和逻辑操作

Intel x86 处理器

- 统治笔记本电脑、台式电脑和服务器市场

- 不断进化的设计
  - 后向兼容的特性（可以兼容到1978年设计的8086）
  - 随时间发展不断增加新的特性

- 复杂指令集计算机 (CISC)
  - 有许多不同的指令和许多不同的形式
    - 但是，Linux程序只用到很小的一部分
  - 难以比得上精简指令集计算机(RISC)的性能
  - 但Intel做到了！
    - 在速度方面是做到了，但是在低功耗方面就做得没那么成功

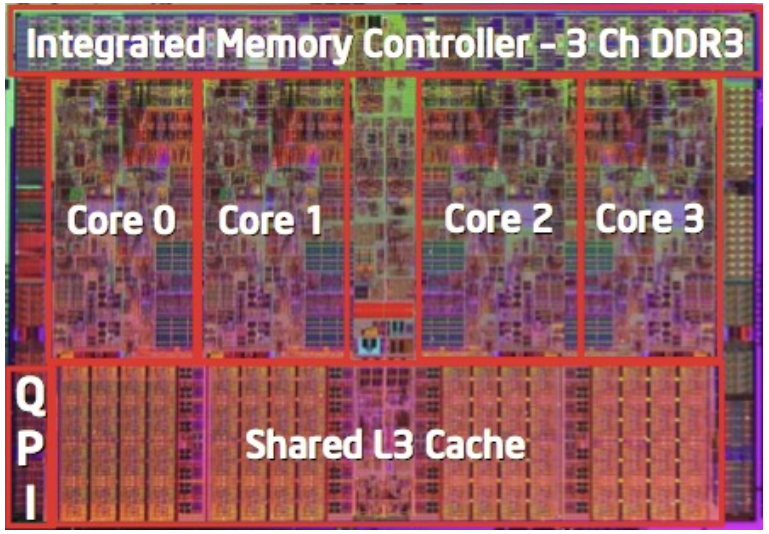
Intel x86 演进过程中的里程碑

名称	日期	晶体管数量	频率(MHz)
8086	1978	29K	5-10
386	1985	275K	16-33
Pentium 4E	2004	125M	2800-3800
Core 2	2006	291M	1060-3500
Core i7	2008	731M	1700-3900

- 8086
  - 第一代16位 Intel 微处理器，IBM PC 和 MS-DOS操作系统的基础
  - 1MB 地址空间
- 386
  - 第一台32位处理器，被称为IA32
  - 增加了“平面寻址模式 flat addressing”，支持Unix操作系统
- Pentium 4E
  - 第一台64位 Intel x86 处理器，被称作x86-64
- Core 2
  - Intel的第一个多核微处理器
- Core i7
  - 四核

Intel x86 处理器（续）

- 机器的演进
  - 386 1985 0.3M
  - Pentium 1993 3.1M
  - Pentium/MMX 1997 4.5M
  - PentiumPro 1995 6.5M
  - Pentium III 1999 8.2M
  - Pentium 4 2001 42M
  - Core 2 Duo 2006 291M
  - Core i7 2008 731M
- 新特性
  - 支持多媒体的操作的指令
  - 支持更高效条件操作的指令
  - 从32位过渡到64位
  - 更多的核



前沿技术 2022.8.20

- Intel 13th Gen Meteor Lake处理器
- 桌面型号
  - P核达到6.0GHz，E核达到4.3GHz
  - 可以有8个P核，16个E核
  - 36MB L3高速缓存

x86领域另一个巨头：AMD

- 一开始……
  - AMD紧随Intel之后
  - 速度慢一点，价格便宜很多
- 之后……
  - 从DEC公司和其他走下坡路的公司里招募了许多顶级电路设计师
  - 打造 Opteron 处理器：Pentium 4的劲敌
  - 发展x86-64，AMD 在64位领域的自主尝试
- 近些年……
  - Intel采取行动
    - 引领世界半导体技术发展
  - AMD暂时落后
    - 依赖外部的半导体制造商
  - 卷土重来？

Intel 64位的历史

- **2001: Intel 从 IA32 到 IA64 的激进尝试**
  - 完全不同的架构 (Itanium 安腾)
  - IA32 的代码作为历史遗留执行
  - 然而表现不尽人意
- **2003: AMD 开始发展自己的解决方案**
  - x86-64 (现在称作 “AMD64”)
- **Intel 察觉到，应该着重研发 IA64**
  - 不想承认自己的错误，或者AMD更为优秀
- **2004: Intel 宣布 IA32 的 EM64T 扩展**
  - 扩展内存至64位技术
  - 几乎与 x86-64 一模一样!
- **除了部分较为低廉的x86处理器外，大都支持x86-64**
  - 但是，仍有许多代码以32位模式运行

2022 Apple M2

Apple M2是苹果公司设计的一系列基于 ARM 架构的片上系统（SoC），作为其 Mac 台式机、笔记本电脑和 iPad Pro 平板电脑的 CPU 和 GPU。

本课程选用教材的内容涵盖

- **IA32**
  - 传统的x86架构
- **x86-64**
  - 现行的标准
  - `linux> gcc hello.c`
  - `linux> gcc -m64 hello.c`
- 呈现形式
  - 书中主体内容围绕 x86-64 展开
  - 网络旁注中涉及 IA32
  - 我们只对 x86-64 作讲解

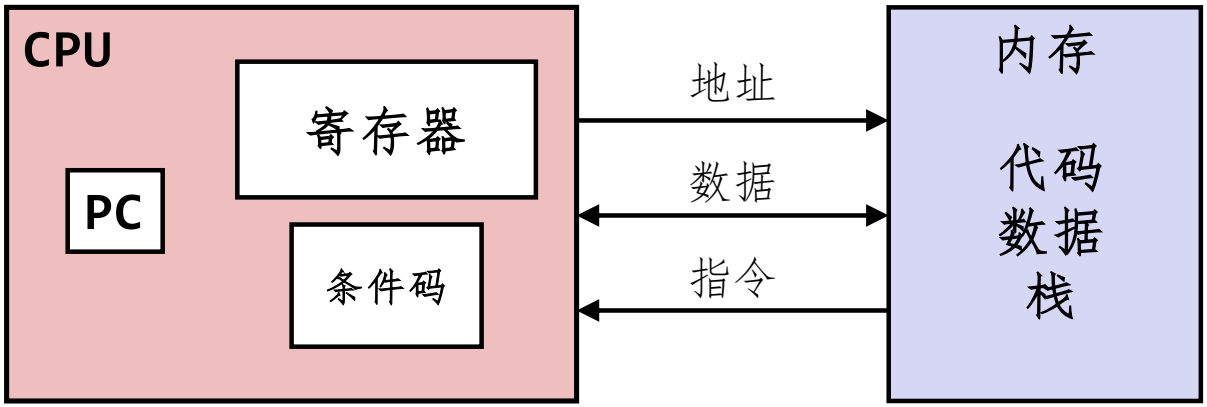
本节内容

- Intel 处理器的历史与架构
- C 代码，汇编代码，机器级代码
- 汇编语言基础：寄存器，操作数，move 数据传送指令
- 算术和逻辑操作

一些定义

- 架构（指令集体系结构，ISA）：理解或书写汇编代码、机器级代码应该掌握的处理器设计的知识
  - 例如：指令集的规范，寄存器
- 微体系结构：架构的实现
  - 例如：高速缓存的大小、核的频率
- 代码格式：
  - 机器代码：处理器执行的字节级程序
  - 汇编代码：机器代码的文本表示
- ISA举例：
  - Intel：x86，IA32，Itanium，x86-64
  - ARM：Apple M1/M2，几乎应用在所有的手机上
  - MIPS，RISC-V
  - Loongson：LoongArch

汇编代码&机器代码之观

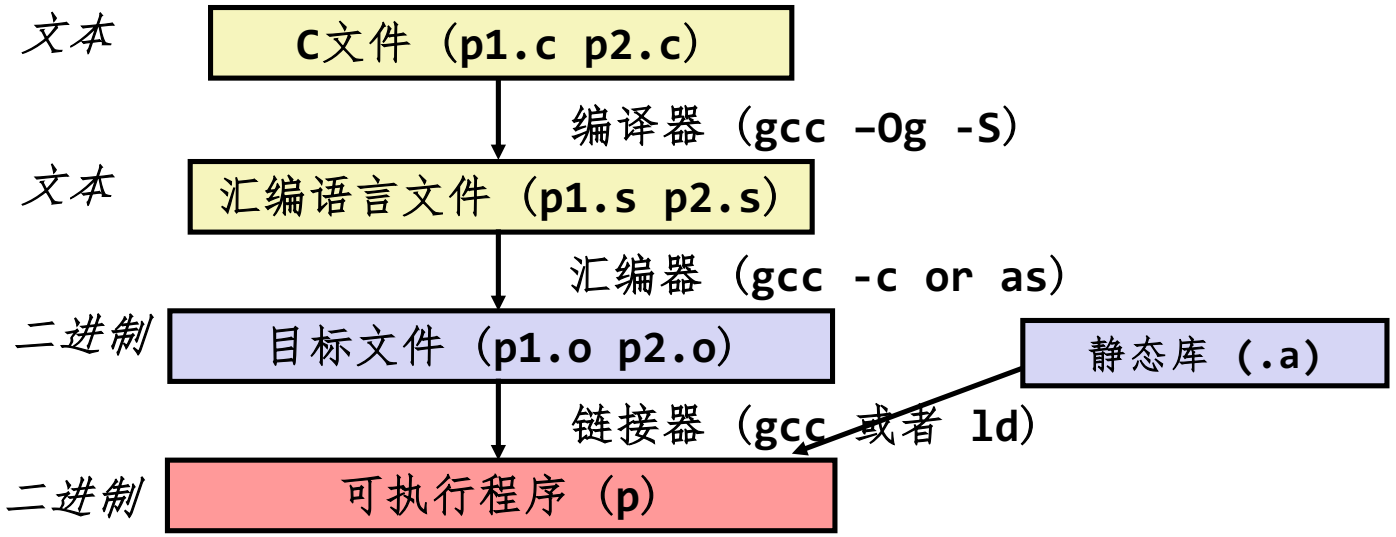


一个程序员应该知道：

- **PC：程序计数器**
  - 下一条指令的地址
  - 称作“RIP”（x86-64）
- **寄存器文件**
  - 程序频繁使用的数据
- **条件码**
  - 储存最近一次算术或逻辑运算的结果
  - 用于条件分支语句
- **内存**
  - 以字节为单位的数组
  - 代码和用户数据
  - 为程序创建栈帧

C 文件是如何转变为可执行程序的？

- 代码文件： **p1.c p2.c**
- 使用如下命令编译： **gcc –Og p1.c p2.c -o p**
  - 使用最基本的优化选项（**-Og**）
  - 得到的二进制文件存放在 **p** 中



编译生成汇编语言

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汇编语言文件

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

使用命令

gcc -Og -S sum.c

生成文件 **sum.s**

*注意：由于不同版本的gcc或不同的编译器设置，在不同的机器（**Linux，Mac OS-X**）上会得到非常不同的结果。*

计算机系统基础-深入理解计算机系统（第三版）

汇编语言特点：数据类型

- 1、2、4或8字节的“整型”
  - 数据值
  - 地址（无类型的指针）
- 4、8或10字节的浮点型数据
- 代码：指令的字节序列编码
- 没有像数组或结构体这样的聚合类
  - 只是在内存中分配连续的字节

汇编语言特点：操作

- 对存储在寄存器或内存上的数据进行算术操作
- 在内存和寄存器之间传送数据
  - 从内存向寄存器加载数据
  - 向内存存储寄存器的数据
- 控制的转移
  - 无条件跳转到过程或从过程返回
  - 条件分支语句

目标代码

sumstore的代码

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

- 一共14个字节
- 每一个指令有1,3或5字节
- 从地址0x0400595开始

- 汇编器
  - 将 .s 文件汇编为 .o 文件
  - 每条指令的二进制编码
  - 几近完整的可执行代码
  - 还缺少不同文件之间的链接
- 链接器
  - 解析文件之间的引用
  - 与静态运行时库结合
    - 例如 **malloc, printf** 的代码
  - 一些库是*动态链接*的
    - 链接发生在程序开始执行时

机器指令示例

```
*dest = t;
```

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

```
0x40059e:  48 89 d3
```

- C 代码
  - 将 t 的值存放在 dest 指向的地方
- 汇编语言
  - 将8字节的值存储在内存中
    - 在 x86-64 术语中称为*四字*
  - 操作数：

    t: 寄存器 **%rax**

    dest: 寄存器 **%rbx**

    ***dest:** 内存 **M[%rbx]**
- 目标代码
  - 字节指令
  - 存储在地址 **0x40059e** 处

反汇编目标代码

反汇编后

```
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

  objdump -d sum

  - 检查目标代码的有用工具
  - 分析指令序列的位模式
  - 得到目标代码大致对应的汇编代码
  - 可被用于 .out（完全可执行文件）和 .o 文件

另一种反汇编的方式

目标代码

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

反汇编后

```
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 gdb 调试器中

  gdb sum

  disassemble sumstore

  - 反汇编过程

  x/14xb sumstore

  - 检查从 sumstore 开始的14个字节

什么是可以反汇编的？

```
% objdump -d WINWORD.EXE

WINWORD.EXE:   file format pei-i386

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

30001000 <.text>:
30001000:
30001001:
30001003:
30001005:
3000100a:
```

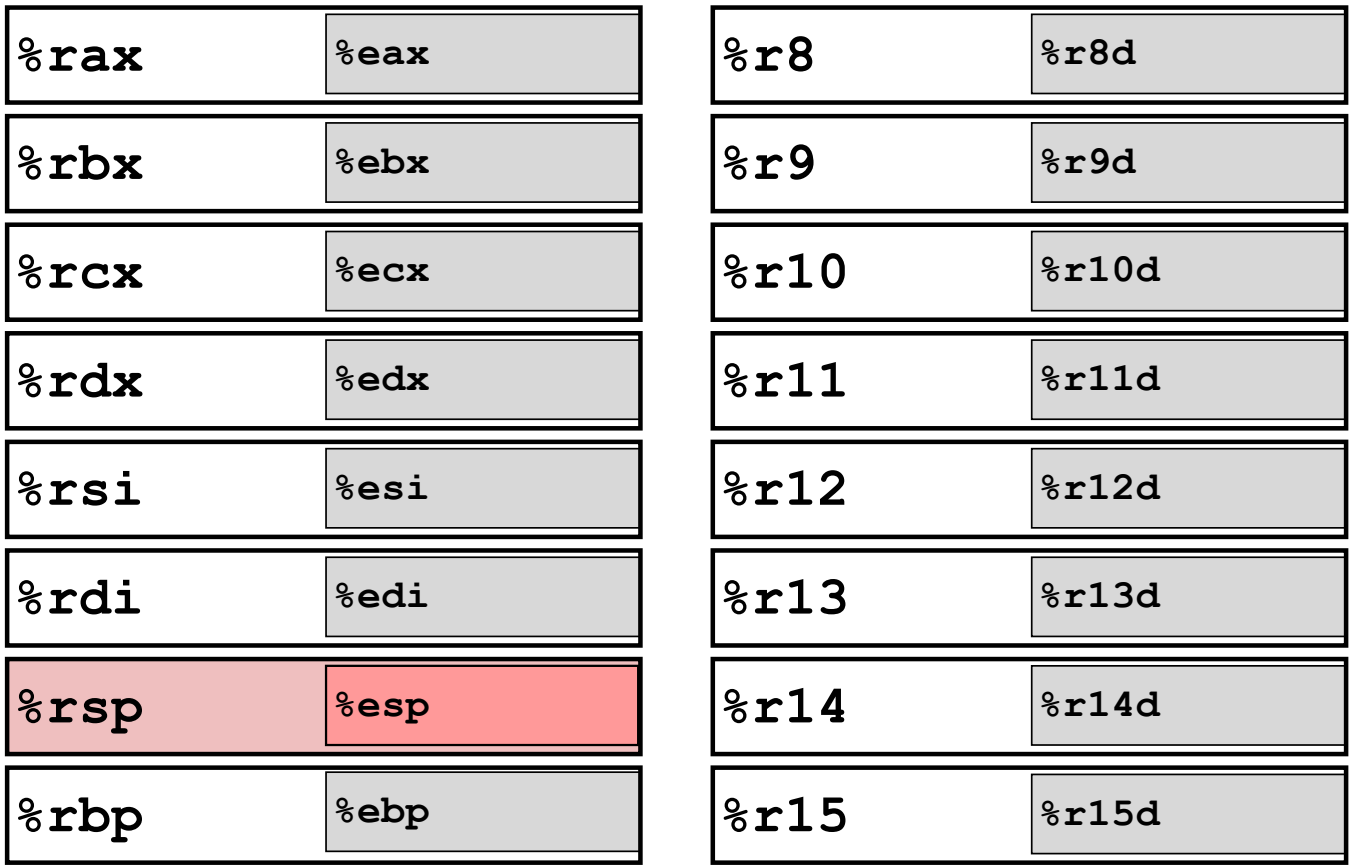
微软最终用户许可协议禁止反向工程

- 任何可以被解释为可执行代码的文件
- 反汇编器检查字节内容并重构汇编源代码

本节内容：

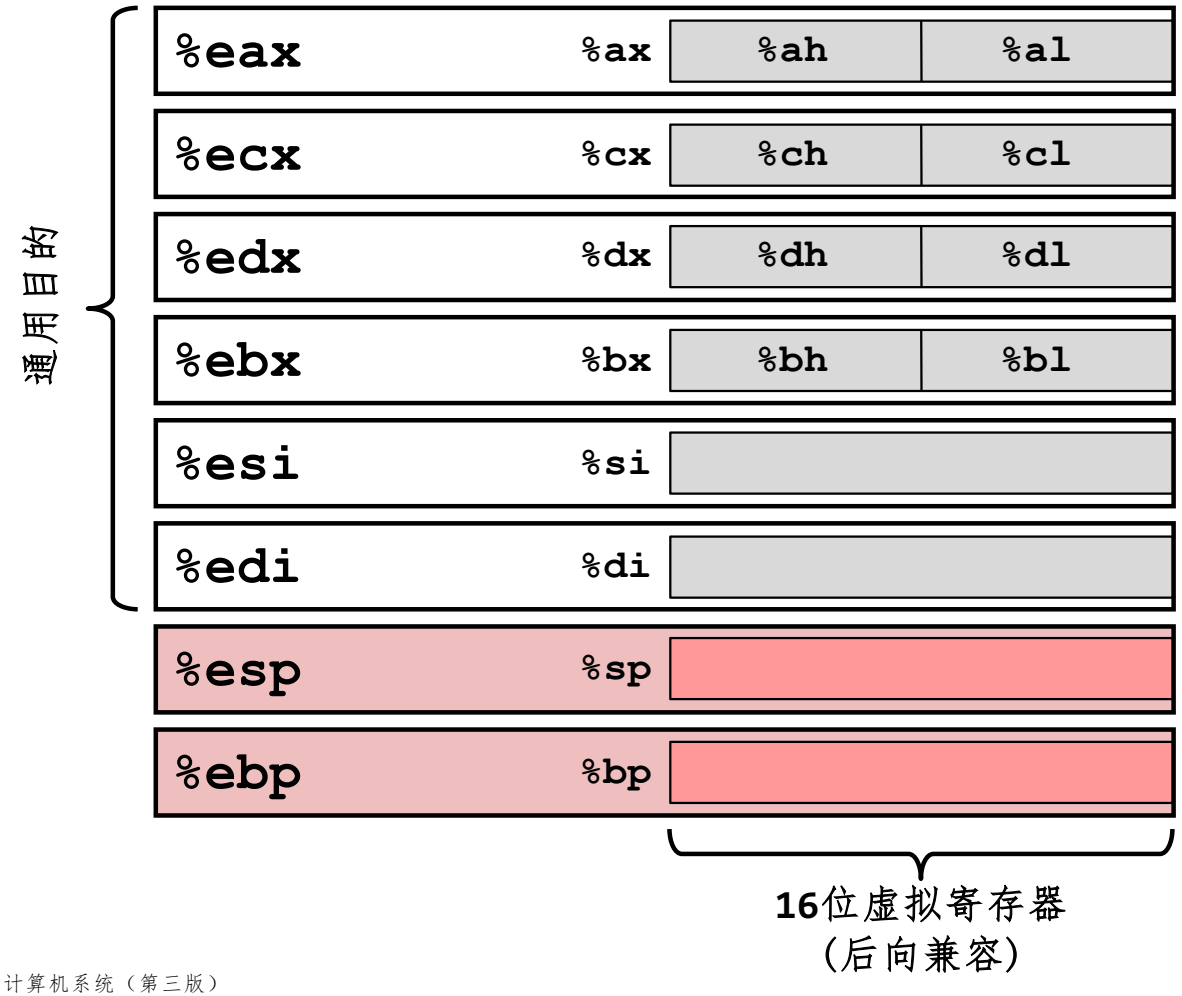
- Intel 处理器的历史与架构
- C 代码，汇编代码，机器级代码
- 汇编语言基础：寄存器，操作数，move 数据传送指令
- 算术和逻辑操作

x86-64整型寄存器



- 也可以只引用低1,2或4字节

一些历史：IA32寄存器



起源
（大部分已经淘汰）

累加 accumulate

计数器 counter

数据 data

基址 base

源索引 sourceindex

目的索引 destination index

栈指针 stack pointer

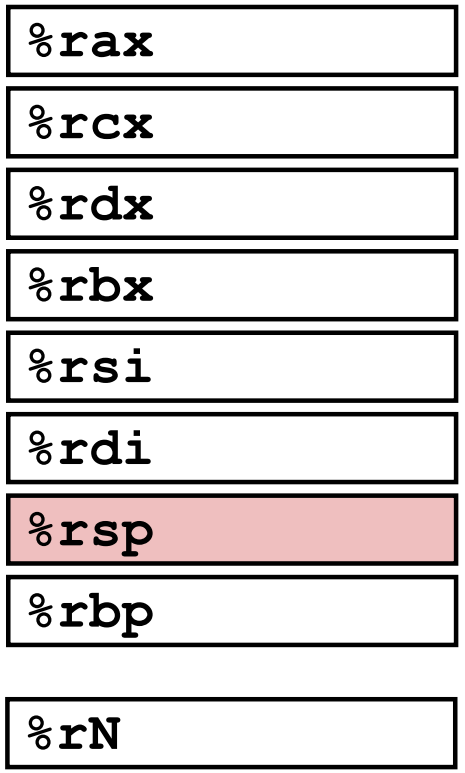
基址指针 base pointer

传送数据

- 传送数据

  movq *Source*, *Dest*

- 操作数类型
  - *立即数：* 常整型数据
    - 例如：**$0x400, $-533**
    - 类似于 C 常量，只是多了前缀 **$**
    - 用1，2或4字节编码
  - *寄存器：* 16个整型寄存器中的一个
    - 例如：**%rax, %r13**
    - 但 **%rsp** 保留为特殊用途
    - 其它的寄存器对个别指令有特殊用途
  - *内存：* 内存中从寄存器指向的地址开始的8个连续字节
    - 最简单的例子：**(%rax)**
    - 也有一些其它的寻址方式



movq 操作数组合示例

	Source	Dest	Src,Dest	对应的 C 代码
movq	立即数	Reg	movq $0x4,%rax	temp = 0x4;
		Mem	movq $-147,(%rax)	*p = -147;
	寄存器	Reg	movq %rax,%rdx	temp2 = temp1;
		Mem	movq %rax,(%rdx)	*p = temp;
	内存	Reg	movq (%rax),%rdx	temp = *p;

单个指令无法实现数据的内存到内存传送

简单寻址模式

- 通常 (R) Mem[Reg[R]]
  - 寄存器 R 表示内存地址
  - 类似于 C 中的解引用

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

- 偏移量 D(R) Mem[Reg[R]+D]
  - 寄存器 R 表示一段内存区间的起始地址
  - 常偏移量 D 表示偏移量的大小

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

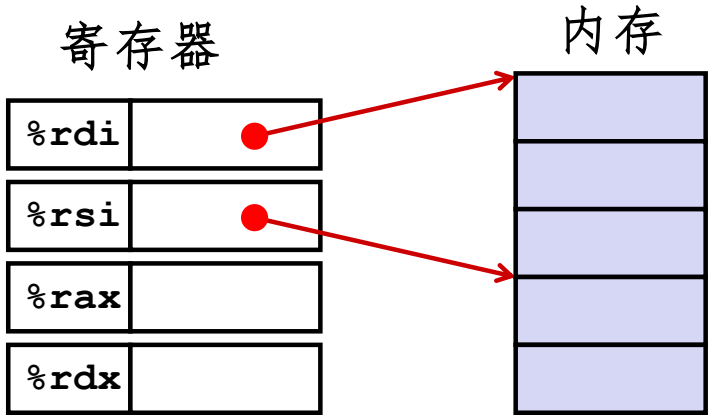
示例

```
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
```

理解函数Swap()

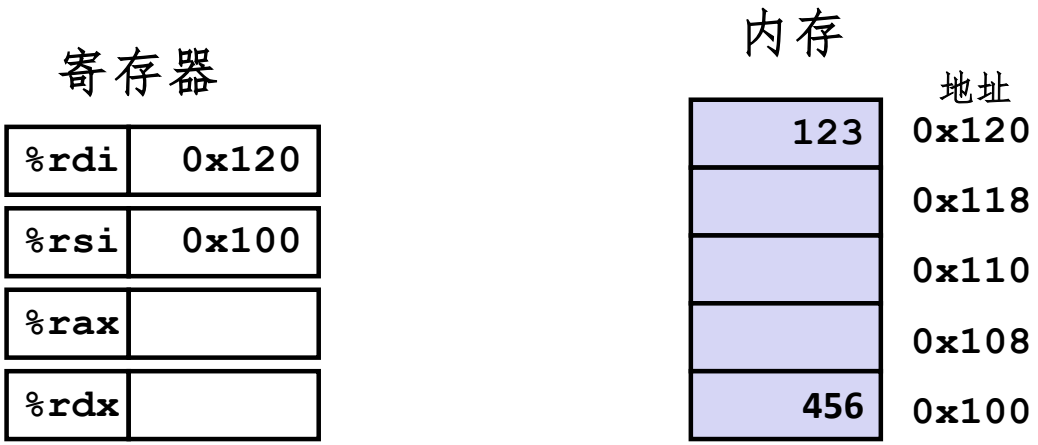
```
void swap(long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



寄存器	值
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

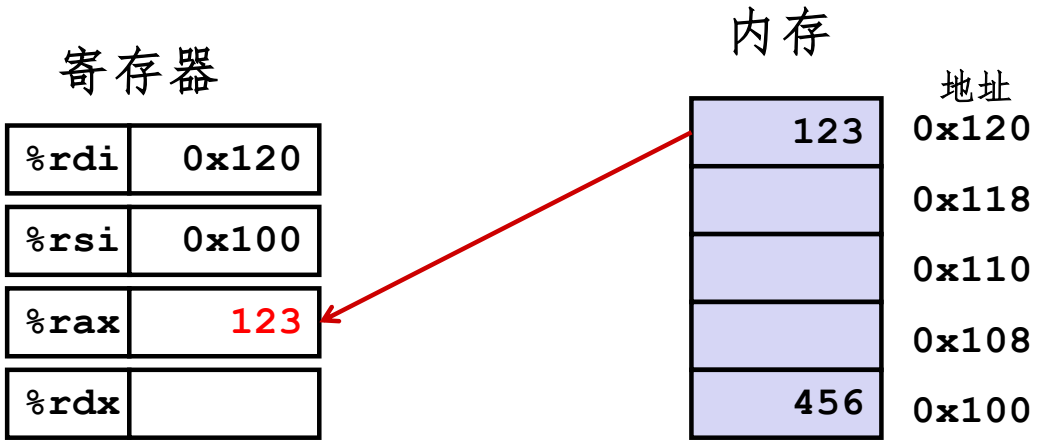
```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

理解函数Swap()



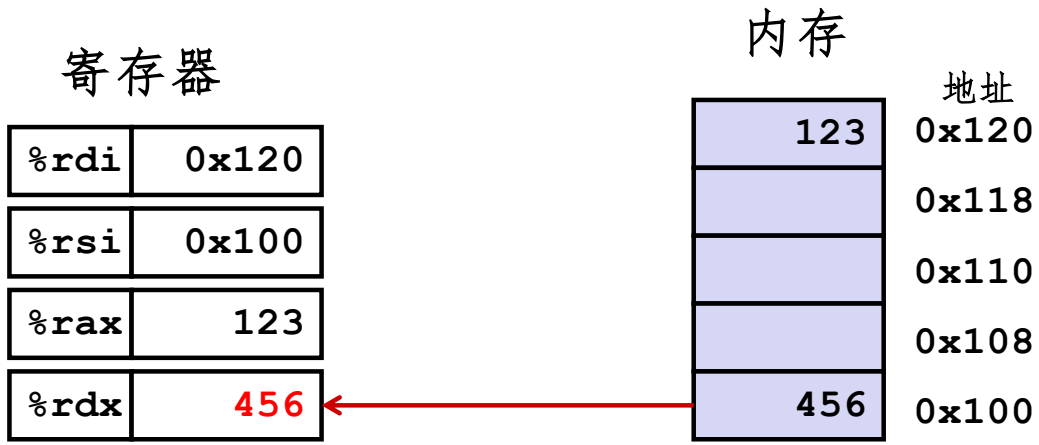

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

理解函数Swap()



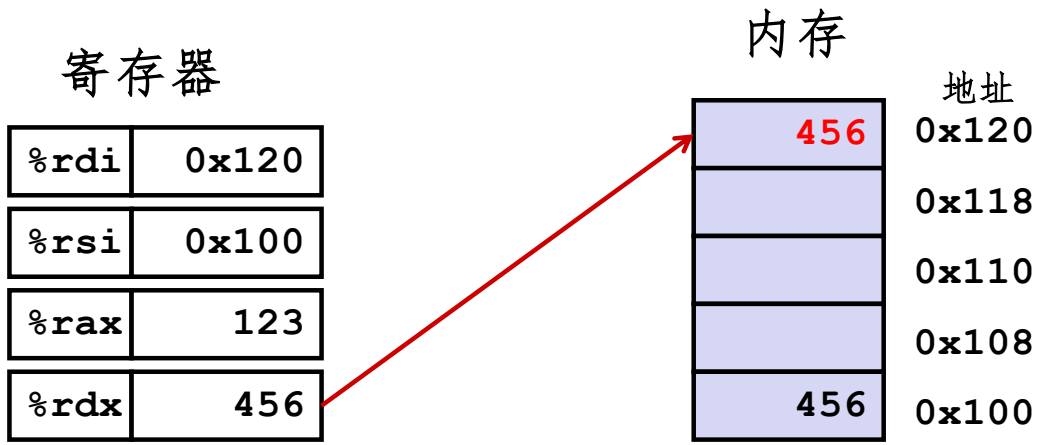

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

理解函数Swap()



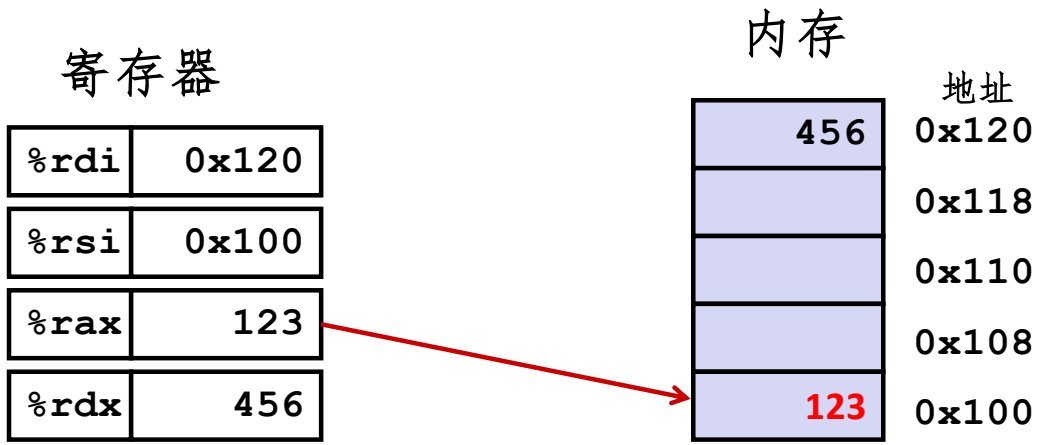

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

理解函数Swap()




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

理解函数Swap()




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

简单内存寻址模式

- 通常 (R) Mem[Reg[R]]
  - 寄存器R表示内存地址
  - 类似于C中的解引用

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

- 偏移量 D(R) Mem[Reg[R]+D]
  - 寄存器 R 表示一段内存区间的起始地址
  - 常偏移量 D 表示偏移量的大小

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

完整寻址模式

- 最通用的形式

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

  - D:　常偏移量，1、2或4字节
  - Rb:　基址寄存器，16个整型寄存器中的任意一个
  - Ri:　索引寄存器，除 **%rsp** 外的任一寄存器
  - S:　伸缩值，1，2，4或8（*为什么是这些数？*）

- 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]]

地址计算示例

%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)	2*0xf000 + 0x80	0x1e080

本节内容

- Intel 处理器的历史与架构
- C 代码，汇编代码，机器级代码
- 汇编语言基础：寄存器，操作数，move 数据传送指令
- 算术和逻辑操作

计算机系统基础-深入理解计算机系统（第三版）

地址计算指令

- **leaq Src, Dst**
  - Src是寻址模式表达式
  - Dst目的地址表达式

- 用途
  - 计算地址但不发生内存引用
    - 例如，p = &x[i];
  - 计算形如 x + k*y 的算术表达式
    - k = 1, 2, 4或8
- 例如

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

编译器转换到汇编语言

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

一些算术操作

- 二元操作

	格式	计算结果	
addq	Src,Dest	Dest = Dest + Src	
subq	Src,Dest	Dest = Dest – Src	
imulq	Src,Dest	Dest = Dest * Src	
salq	Src,Dest	Dest = Dest << Src	也称为：shlq
sarq	Src,Dest	Dest = Dest >> Src	算术
shrq	Src,Dest	Dest = Dest >> Src	逻辑
xorq	Src,Dest	Dest = Dest ^ Src	
andq	Src,Dest	Dest = Dest & Src	
orq	Src,Dest	Dest = Dest | Src	

- 请注意参数的顺序
- 无符号整型和有符号整型之间没有区别（为什么？）

一些算术操作

- 一元操作

```
incq    Dest      Dest = Dest + 1
decq    Dest      Dest = Dest – 1
negq    Dest      Dest = – Dest
notq    Dest      Dest = ~Dest
```

- 更多的指令见课本教材

算术表达式示例

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

有趣的指令

- **leaq**: 加载有效地址
- **salq**: 移位
- **imulq**: 乘法
  - 但是只使用了一次

理解算术表达式示例

```
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
   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

程序的机器级表示I：总结

- Intel 处理器的历史与架构
  - 不断演进的设计产生了很多奇怪之处
- C 代码，汇编代码，机器级代码
  - 新的可见状态：程序计数器，寄存器...
  - 编译器需要将语句、表达式和过程转换成低级的指令序列
- 汇编语言基础：寄存器，操作数，move 数据传送指令
  - x86-64 的move 指令囊括了各类型的数据传送形式
- 算术和逻辑操作
  - C 编译器会分辨出不同的指令组合，并执行计算