机器级编程 0
Machine-Level Programming

课 程 名 ：计算机系统

第 2 讲 （2025 年 4 月 23 日）

主 讲 人 ：杜海文

本课内容

- **Intel 处理器及其架构沿革**
- C、汇编和机器代码
- 汇编语言基础：寄存器、操作数、move 指令族
- 算术逻辑操作

Intel x86 处理器

- 在笔记本、台式机、服务器市场一度占据统治地位
- 进化式的设计理念
  - 可向后兼容至 8086（1978 年）
  - 后续产品中加入越来越多的功能、特性
- 复杂指令系统计算机CISC
  - 指令条数、格式众多
    - 但仅有一小部分在 Linux 程序中用到
  - 性能难以跟媲美 精简指令系统计算机RISC
  - 但是 Intel 做到了！
    - 此处的性能主要指速度而非功耗

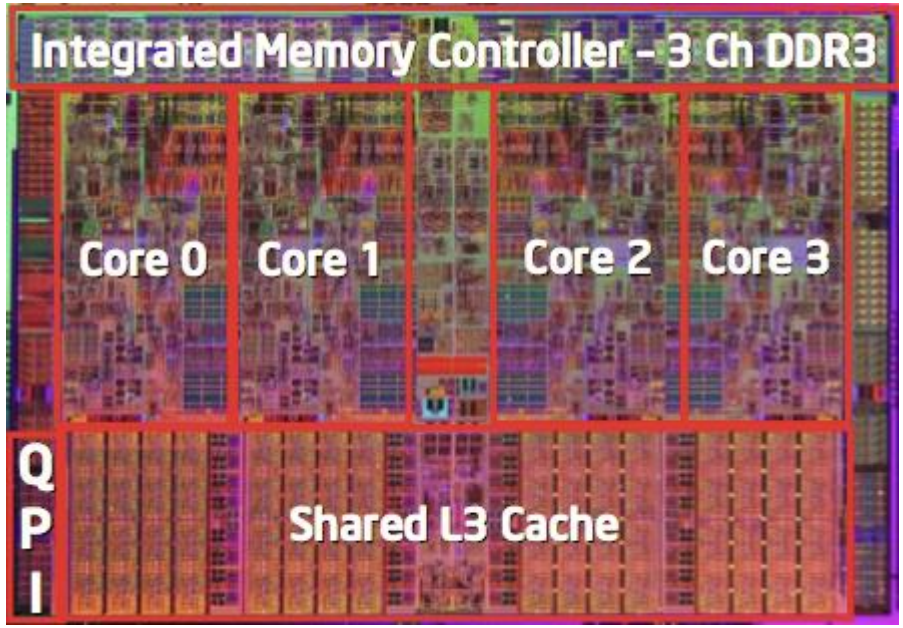
Intel x86 演化：标志性产品

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

- 8086
  - 第一个 16-bit Intel 处理器，基于其上推出 IBM PC 和 DOS 操作系统
  - 1MB 寻址空间
- 386
  - 第一个 32-bit Intel 处理器，其指令集架构称作 IA32
  - 添加了“平坦寻址”，能够运行 Unix
- Pentium 4E
  - 第一个 64-bit Intel 处理器，其指令集架构称作 x86-64
- Core 2
  - 第一个多核 Intel 处理器
- Core i7
  - 四个核心

Intel x86 处理器（续）

- 机型演化
  - 386 1985 0.3M
  - Pentium 1993 3.1M
  - Pentium/MMX 1997 4.5M
  - PentiumPro 1995 6.5M
  - Pentium III 1999 8.2M
  - Pentium 4 2000 42M
  - Core 2 Duo 2006 291M
  - Core i7 2008 731M



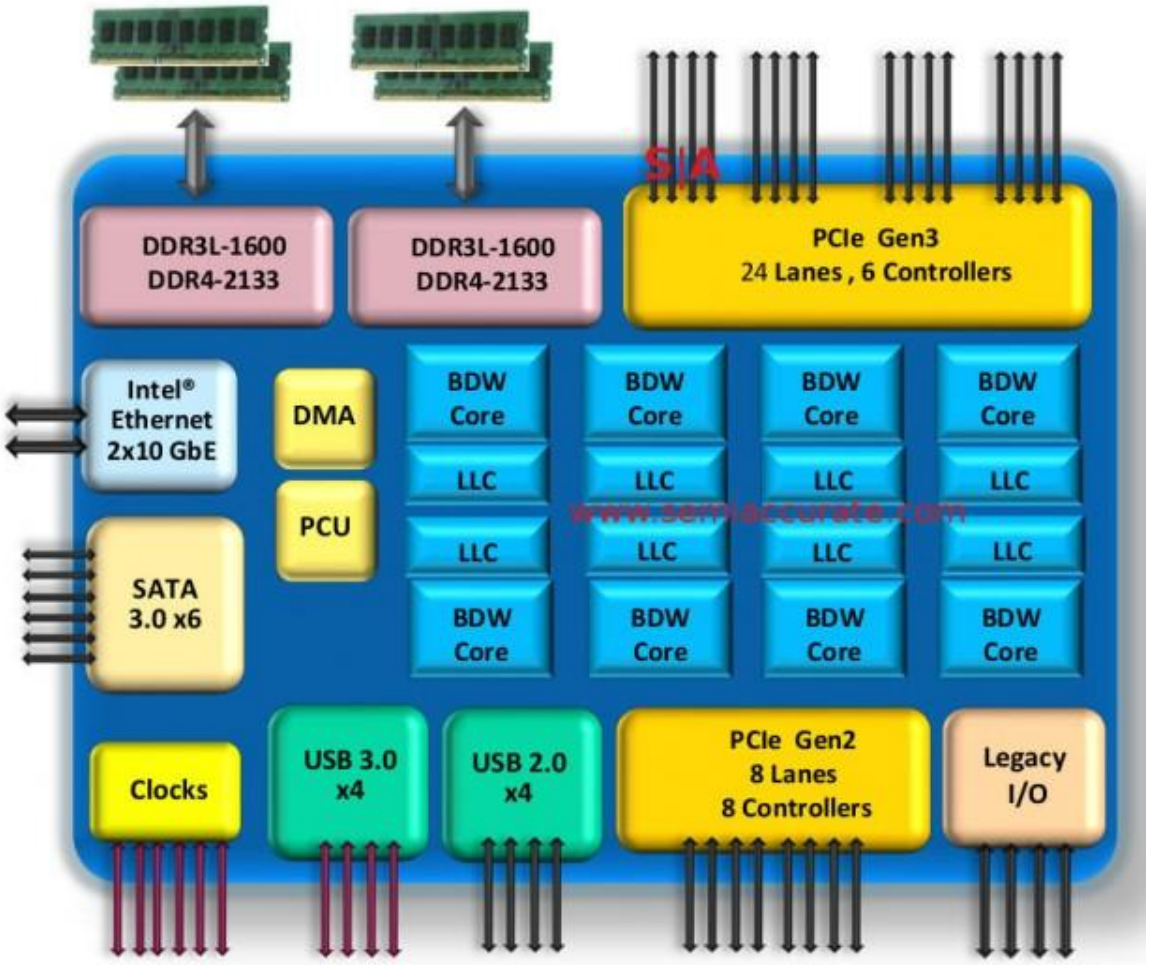
- 陆续加入的特性
  - 多媒体指令，支持多媒体操作
  - 支持更高效的条件操作的指令
  - 由 32 位机转向 64 位机
  - 更多的 CPU 核心

本课程示例机型

- Core i7 Broadwell 2015（课堂演示 Core i5 Broadwell 2015）

- 台式机型
  - 4 核
  - 集成显卡
  - 3.3-3.8 GHz
  - 65 W

- Server Model
  - 8 cores
  - Integrated I/O
  - 2-2.6 GHz
  - 45W



Intel x86 的克隆：AMD 处理器

- 曾经
  - AMD（Advanced Micro Devices, Inc.超威半导体） 作为 Intel 的跟随者
  - 其产品性能（速度）略慢，价格很低
- 后来
  - 从 DEC、飞思卡尔等公司聘请了顶级硬件设计师
  - 率先突破 1 GHz 处理器主频
  - 开发了自研的 x86-64，为 AMD 自己对 IA32 进行的扩展
  - 2006年，收购显示芯片厂商 ATI
  - 2009年，将半导体制造业务剥离，成为专注于设计的 fabless 厂商
  - 推出 Zen 微架构（服务器端 EPYC，桌面端 Ryzen）
  - 2022年，收购 Xilinx，集齐 CPU+GPU+FPGA 产品线

Intel 的 64 位处理器发展史

- 2001: Intel 尝试了一条激进路线，由 IA32 转向 IA64
  - 完全不同的架构（安腾Itanium）
  - Executes IA32 code only as legacy
  - 表现不尽如人意
- 2003: AMD 倡导了一个基于 IA32 的演进版本
  - x86-64（又称 AMD64）
- Intel 出于道义而坚守其 IA64 路线
  - 认错不易，承认 AMD 更好不易
- 2004：Intel 推出 IA32 的 EM64T 扩展
  - Extended Memory 64-bit Technology
  - 几乎与 x86-64 一模一样！
- 目前 x86 系列处理器除少数低端产品外，均支持 x86-64
  - 但是，许多代码仍运行在 32-bit 模式

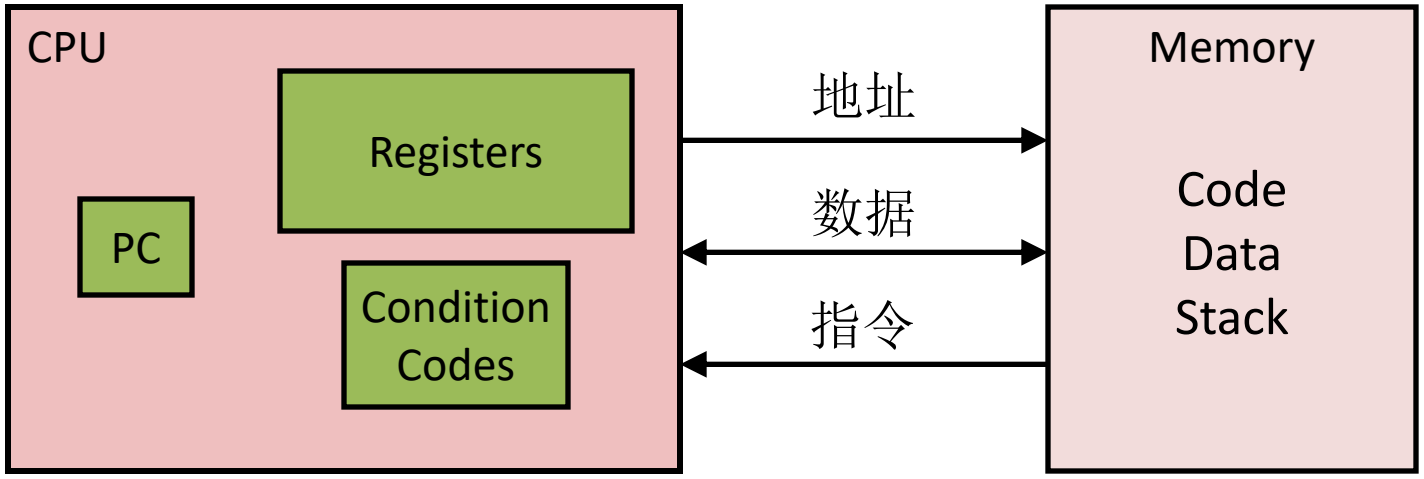
本课内容

- Intel 处理器及其架构沿革
- **C、汇编和机器代码**
- 汇编语言基础：寄存器、操作数、move 指令族
- 算术逻辑操作

名词解释

- 架构architecture（又称指令集架构ISA）：汇编（机器）语言程序员需要了解的处理器的属性
  - 例如：指令集详情、寄存器组织
- 微架构microarchitecture：架构的实现
  - 例如：Cache 的容量、主频
- 代码的形式：
  - 机器代码machine code：以二进制（按字节分组）表示的程序
  - 汇编代码assembly code：将机器代码用文本助记符表示的形式
- 除 x86 外的其它 ISA 举例
  - MIPS、RISC-V
  - ARM：几乎所有手机均采用

汇编（机器）代码

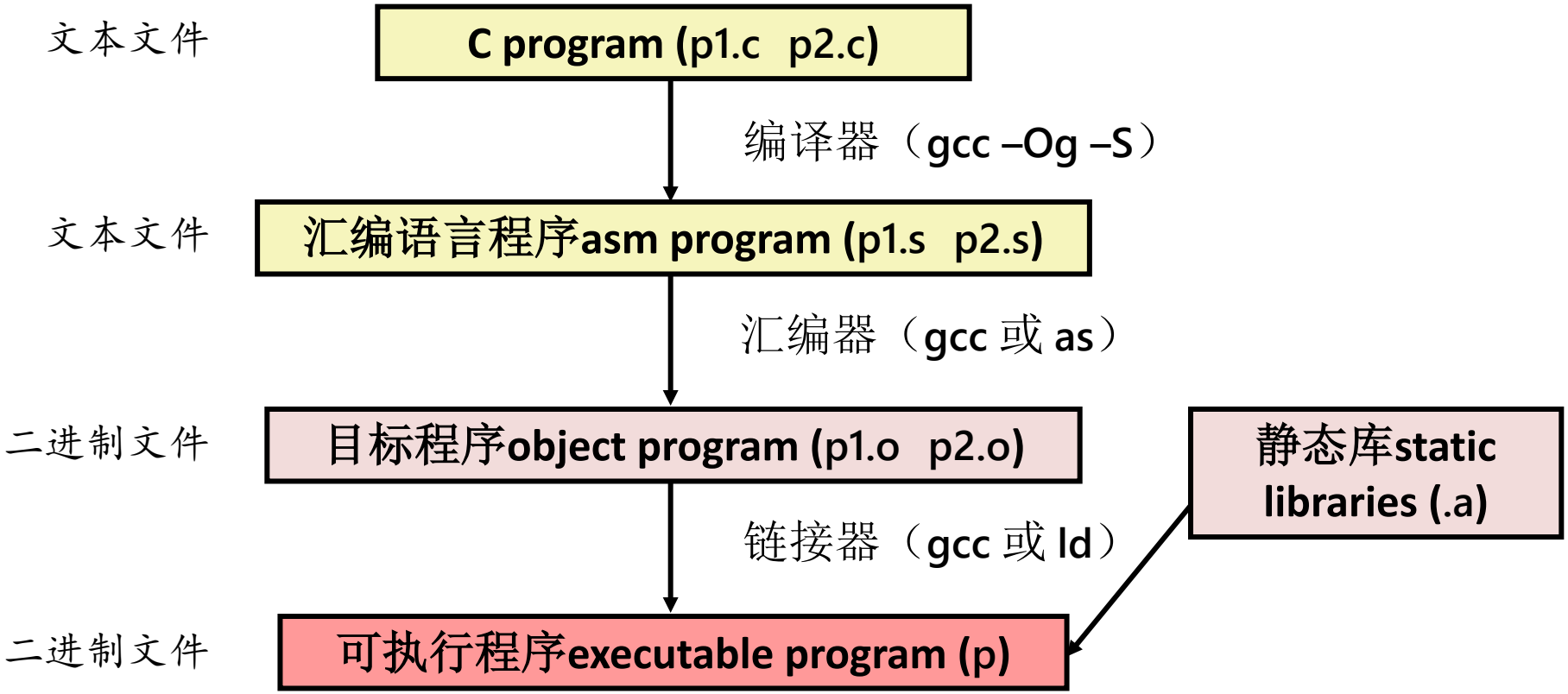


程序员可见状态

- **PC：程序计数器**
  - 下一条指令的地址
  - **x86-64 称其为 RIP**
- **寄存器registers**
  - 程序重度使用的数据
- **条件码conditional codes**
  - 存放最近一次算术逻辑操作的状态信息
  - 用于条件转移
- **存储器memory**
  - 可按字节访问的数组
  - 代码和数据
  - 用于支持过程调用的栈

将 C 程序转换为目标代码

- **源程序文件：p1.c 和 p2.c**
- **编译命令：gcc –Og p1.c p2.c -o p**
  - 使用基本的优化选项 -Og
  - 生成可被机器执行的二进制文件 p



编译成汇编语言

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 汇编代码

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

使用如下命令生成汇编语言程序文件 sum.s：

```
openEuler> gcc –Og –S sum.c
```

注意：由于 GCC 的版本、设置各异，在不同机器上生成的汇编代码可能亦有差别

汇编语言的特性：数据类型

- “整数”类型：1、2、4 或 8 个字节
  - 数值
  - 地址（无所谓类型的指针）
- 浮点数：4、8 或 10 个字节
- 代码code：指令序列，有两种形式
  - 汇编代码，以助记符书写每条指令
  - 机器代码，每条指令以二进制形式表示
- 不存在数组、结构体之类的复合数据类型
  - 各元素、成员变量仅仅是内存中连续存放的若干字节

汇编语言的特性：操作

- 对寄存器、内存单元中的数据执行算术逻辑运算

- 在寄存器、内存单元之间传送数据
  - 将数据从存储器取至寄存器load
  - 将数据从寄存器存至存储器store

- 控制转移
  - 无条件跳转unconditional jump
    - 在地址不连续的指令之间
    - 在过程procedure与过程之间
  - 条件转移conditional branch

目标代码

`sumstore` 函数代码

```
0x040113b:
  0x53
  0x48
  0x89
  0xd3
  0xe8
  0xf2
  0xff
  0xff
  0xff
  0x48
  0x89
  0x03
  0x5b
  0xc3
```

- 共 14 字节
- 每条指令占 3 或 5 字节
- 起始地址为 0x0400595

- **汇编器assembler**
  - 将 `.s` 翻译成二进制的 `.o` 文件
  - 将每条指令转换成对应的二进制形式
  - 接近最终的可执行代码
  - 但还缺少不同文件之间的链接
- **链接器linker**
  - 对文件与文件之间的相互引用进行解析
  - 与静态运行时库合并
    - 例：`malloc`，`printf` 的代码
  - 有的库则采用动态链接*dynamically linked*
    - 链接发生在程序执行时

1、

机器指令举例

```
*dest = t;
```

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

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

- C 代码
  - 将变量 t 的值存至 dest 指向的内存单元
- 汇编代码
  - 将一 8 字节数据存至内存
    - Quad words in x86-64 parlance
  - 操作数

    t: 寄存器 %rax

    dest: 寄存器 %rbx

    *dest: 内存单元 M[%rbx]
- 目标代码
  - 3 字节指令
  - 从内存 0x401144 地址处开始保存

目标代码的反汇编

反汇编结果

```
000000000040113b <sumstore>:
  40113b:  53              push   %rbx
  40113c:  48 89 d3          mov    %rdx,%rbx
  40113f:  e8 f2 ff ff ff    callq  400590 <plus>
  401144:  48 89 03          mov    %rax,(%rbx)
  401147:  5b              pop    %rbx
  401148:  c3              ret
```

- **反汇编器**

  objdump –d sum

  - 查看目标代码的利器
  - 分析指令序列的二进制表示
  - 生成与 .s 类似的汇编代码
  - 可用于 a.out 文件（完全可执行程序）或 .o 目标文件

反汇编的方法之二

目标代码

```
0x040113b:
  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:
 0x000000000040113b <+0>:   push   %rbx
 0x000000000040113c <+1>:   mov    %rdx,%rbx
 0x000000000040113f <+4>:   callq  0x400590 <plus>
 0x0000000000401144 <+9>:   mov    %rax,(%rbx)
 0x0000000000401147 <+12>:  pop    %rbx
 0x0000000000401148 <+13>:  ret
```

- **利用 GDB 调试器**

  gdb sum

  - 进入 GDB 调试 sum 程序（需使用 –g 选项生成 sum）

  disassemble sumstore

  - 对 `sumstore` 过程实施反汇编

  x/14xb sumstore

  - 查看从 `sumstore` 开始的 14 字节

哪些内容可以反汇编？

```
openEuler> objdump -d WINWORD.EXE

WINWORD.EXE:     file format pei-i386


Disassembly of section .text:

00401000 <.text>:
  401000:       e8
  401005:       eb
  401007:       8b
  40100d:       5(
  40100e:       57
  40100f:       bf
```

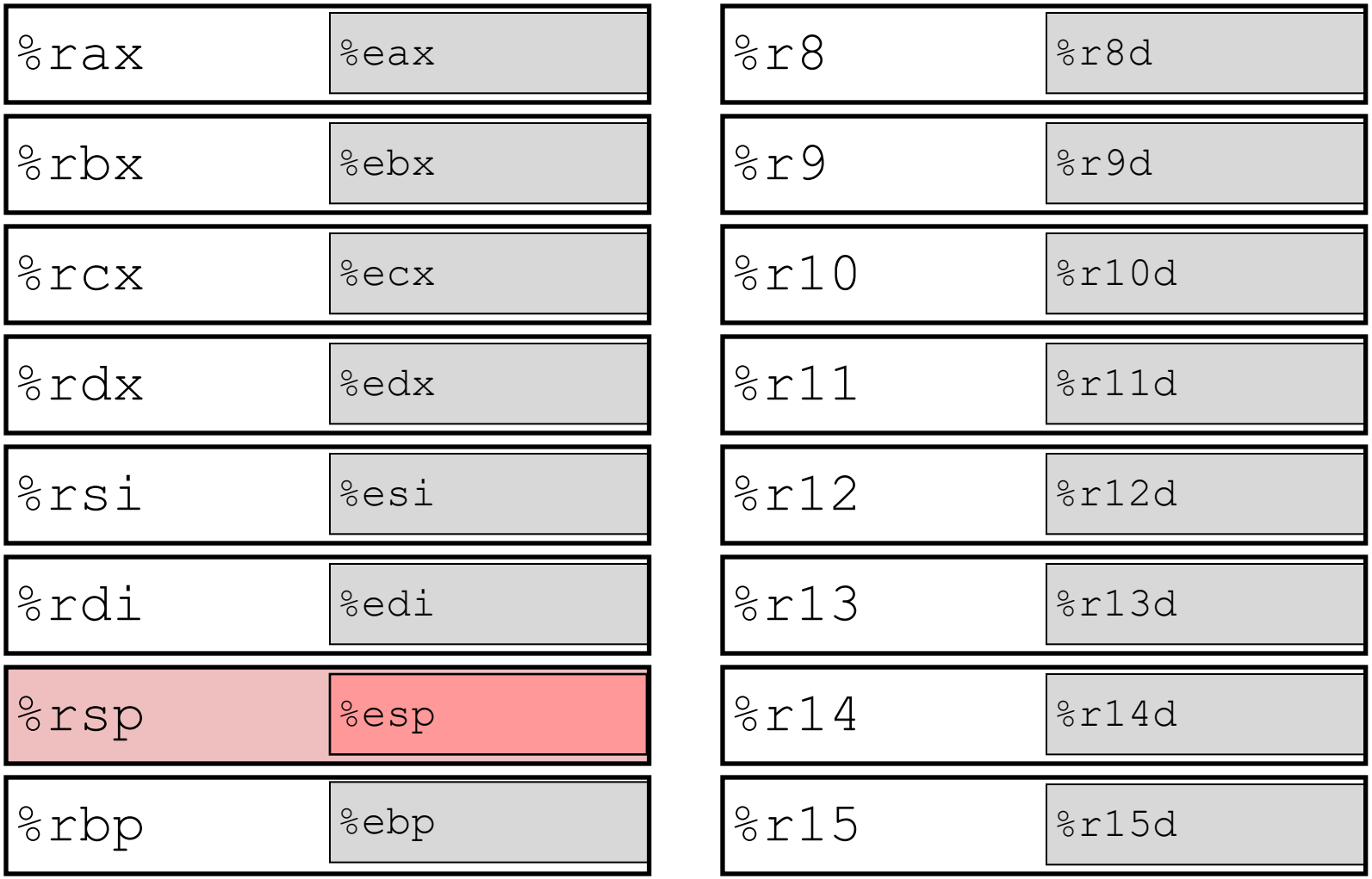
逆向工程reverse engineering
受 Microsoft End User License Agreement 禁止

- 任何可执行代码均可反汇编
- 反汇编器检查目标代码各个字节，据此重建汇编代码

本课内容

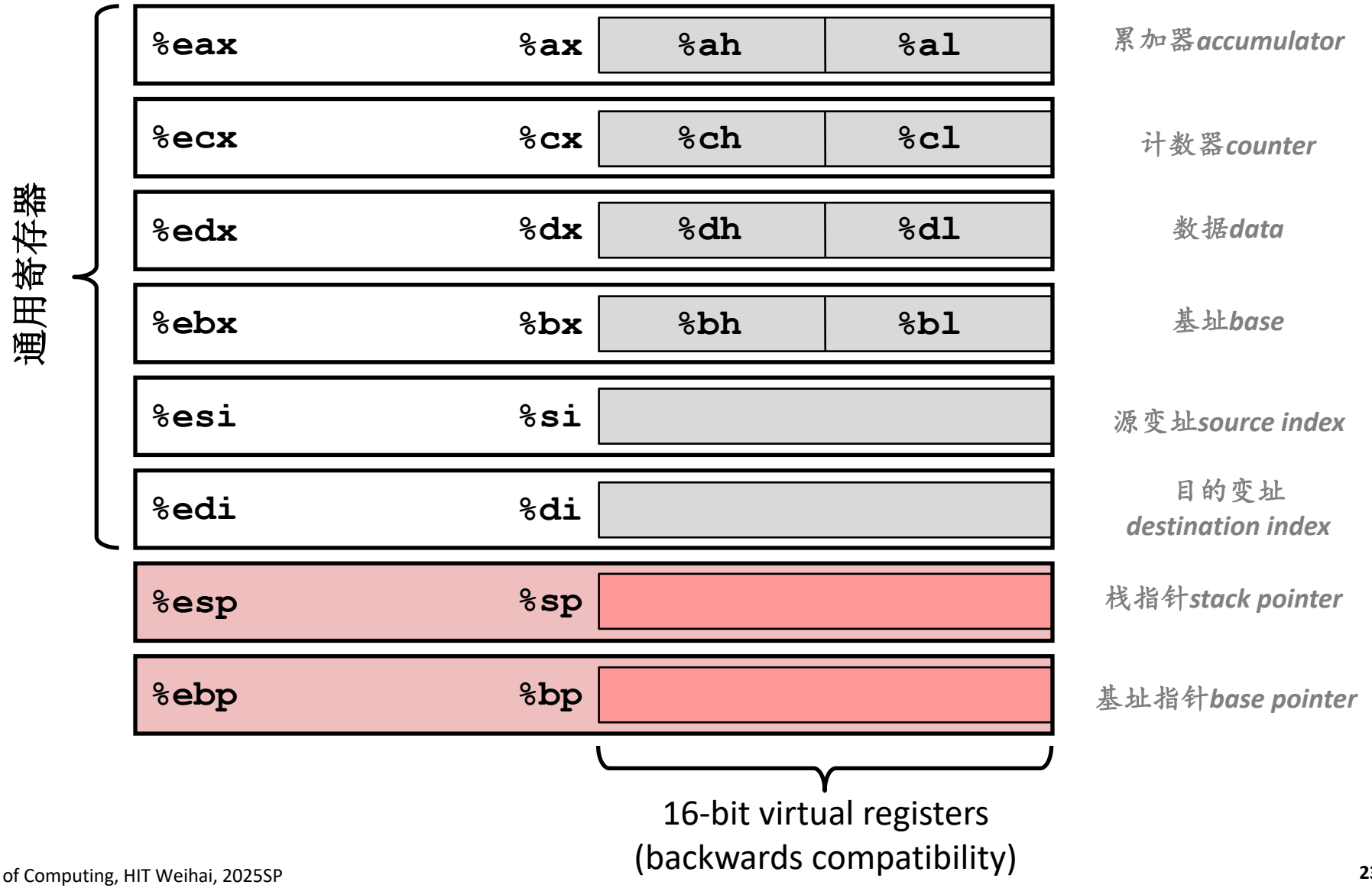
- Intel 处理器及其架构沿革
- C、汇编和机器代码
- **汇编语言基础：寄存器、操作数、move 指令族**
- 算术逻辑操作

x86-64 整数寄存器



- 可独立访问低 4 字节，也可独立访问低 1、2 字节

历史演变：IA32 寄存器

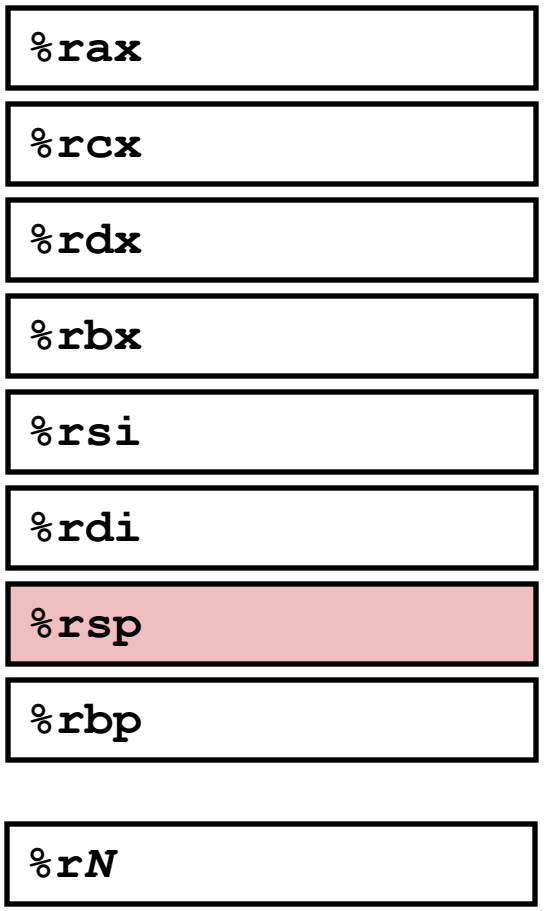


数据传送

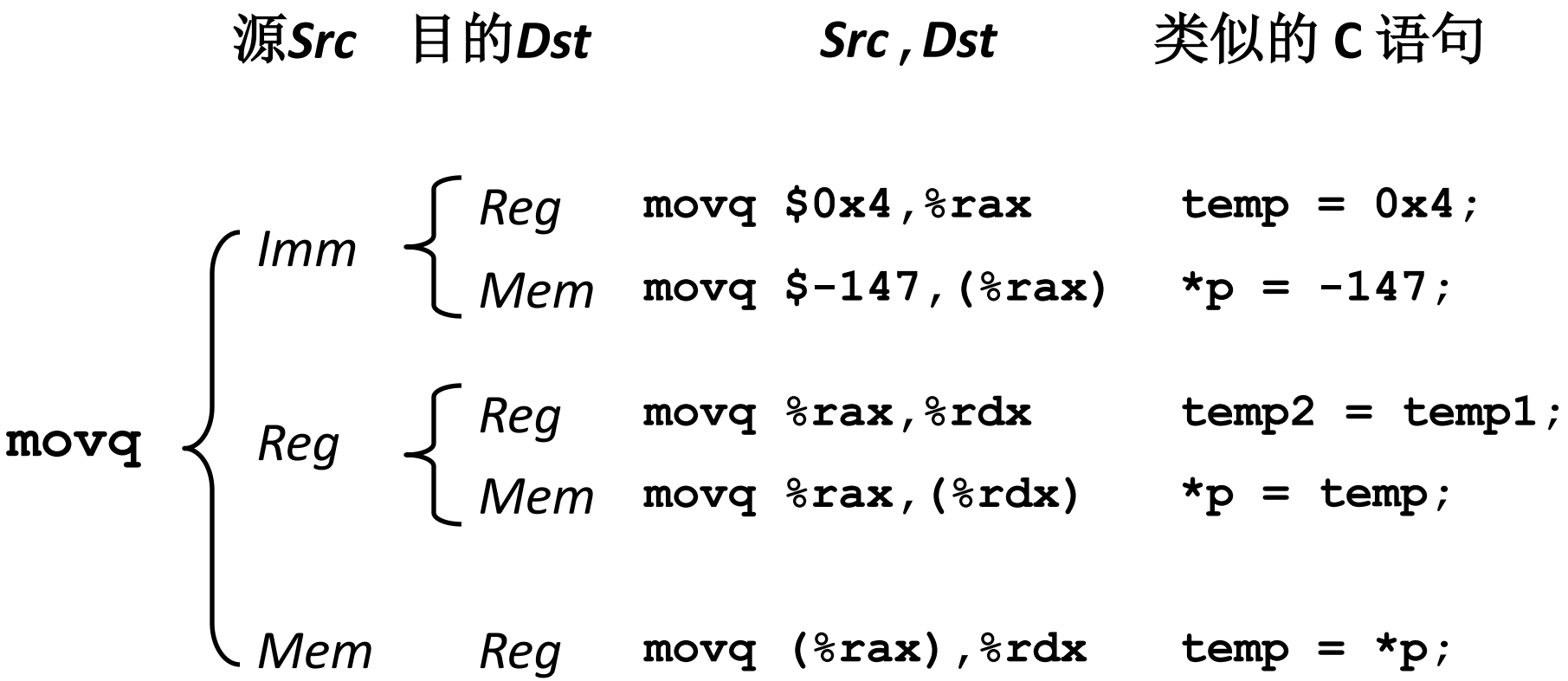
- 数据传送

  `movq` *Src*, *Dst*

- 操作数类型
  - 立即数immediate：整数常量
    - 例：`$0x400`，`$-533`
    - 类似 C 的常量，但需要加'`$`'前缀
    - 对应编码占 1、2 或 4 字节
  - 寄存器register：16 个整数寄存器之一
    - 例：`%rax`、`%r13`、`%ebx`
    - 注意 `%rsp` 为专用
    - 其它寄存器为特定指令所专用
  - 存储单元memory：存储器中的 8 个连续字节，其首地址由寄存器指明
    - 最简单的例子：`(%rax)`
    - 另有其它寻址方式



movq 操作数组合



不允许在一条 move 类指令中出现两个操作数均来自内存的情况

简单的内存寻址方式

- 普通：　　(*R*)　　Mem[Reg[*R*]]
  - 寄存器 *R* 指明内存地址
  - C 的指针解引用

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

- 相对：　　D (*R*)　　Mem[Reg[*R*]+*D*]
  - 寄存器 *R* 指明内存中的起始地址
  - 常数 *D* 为相对偏移量

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

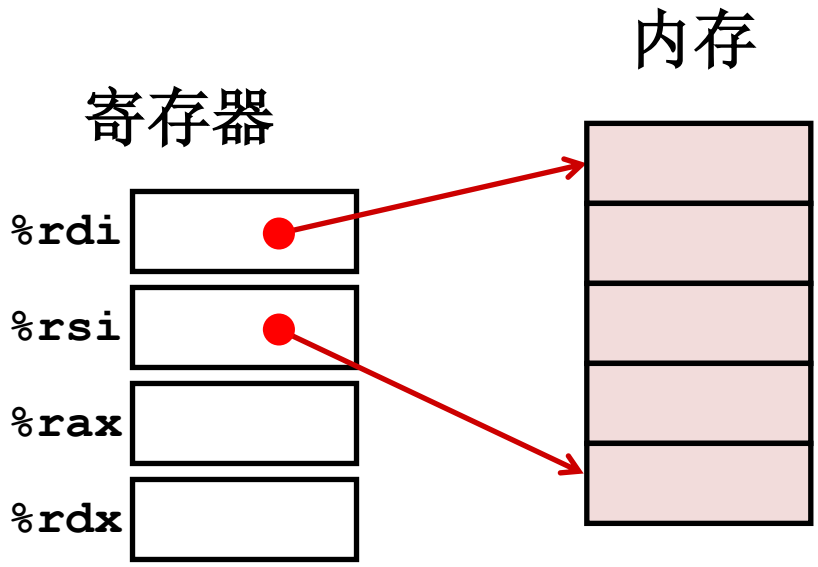
简单的内存寻址方式举例

```
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
```

swap() 详解

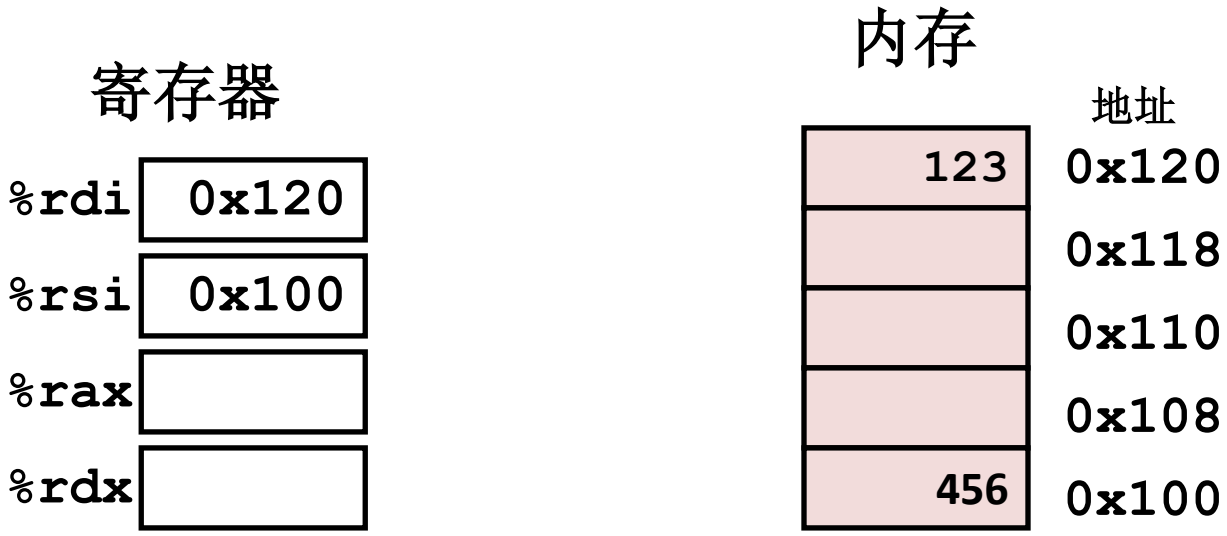
```
void swap
     (long *xp, long *yp)
{
     long t0 = *xp;
     long t1 = *yp;
     *xp = t1;
     *yp = t0;
}
```



寄存器	值
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

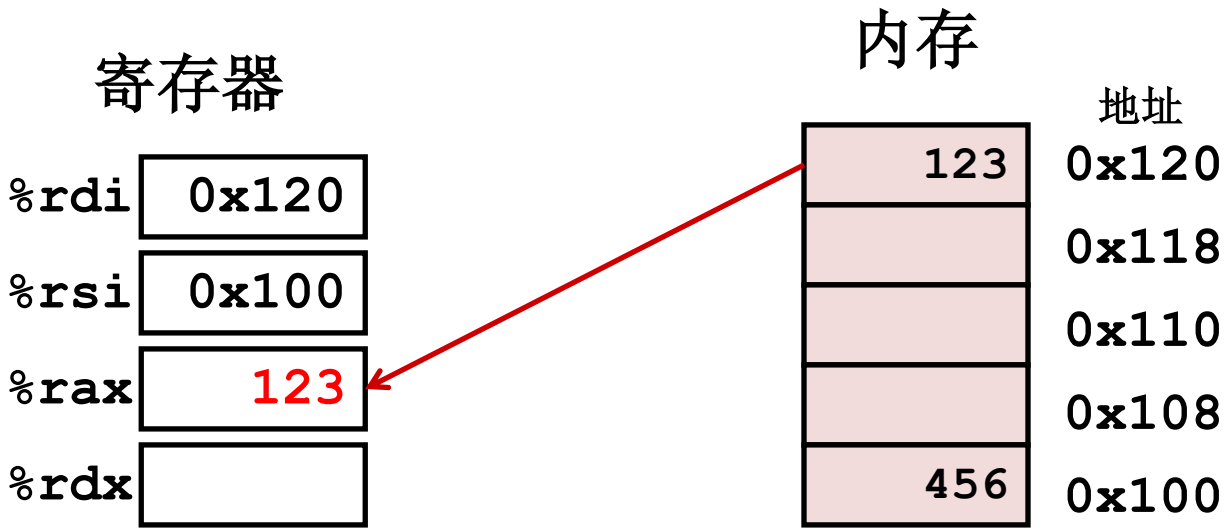
```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

swap() 详解



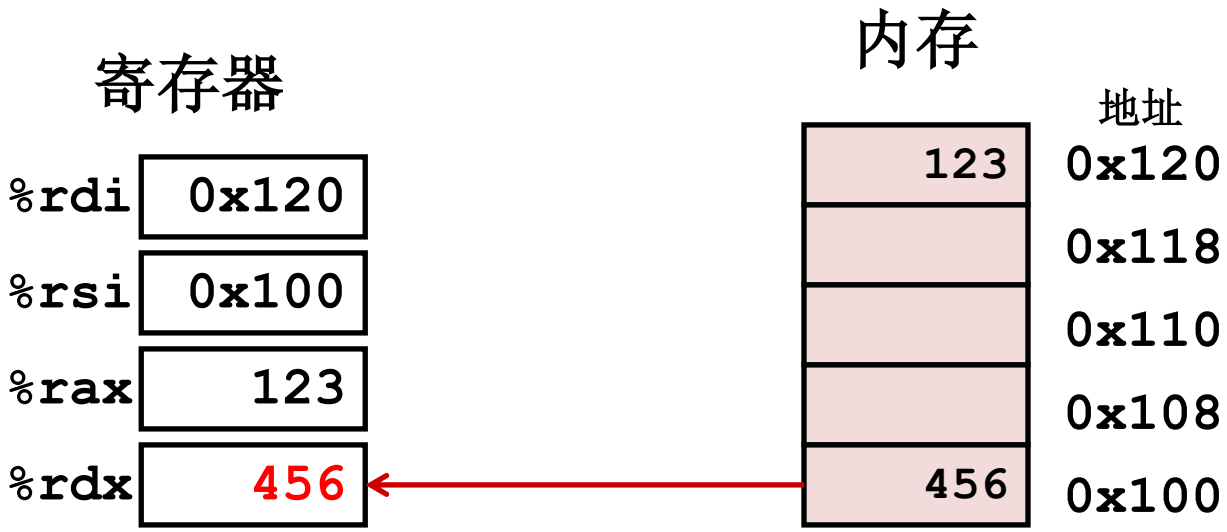

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

swap() 详解



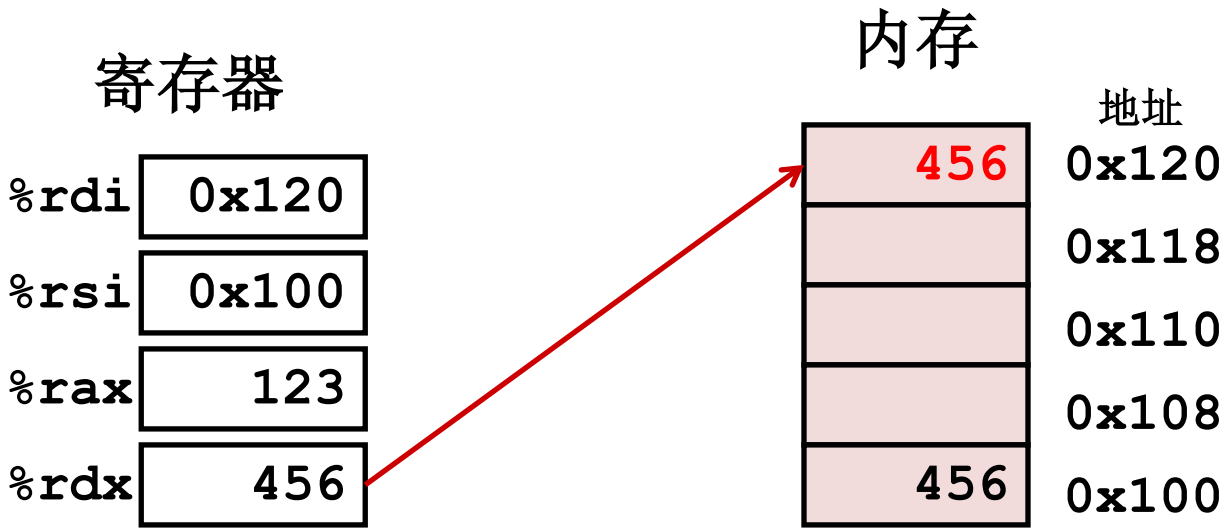

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

swap() 详解



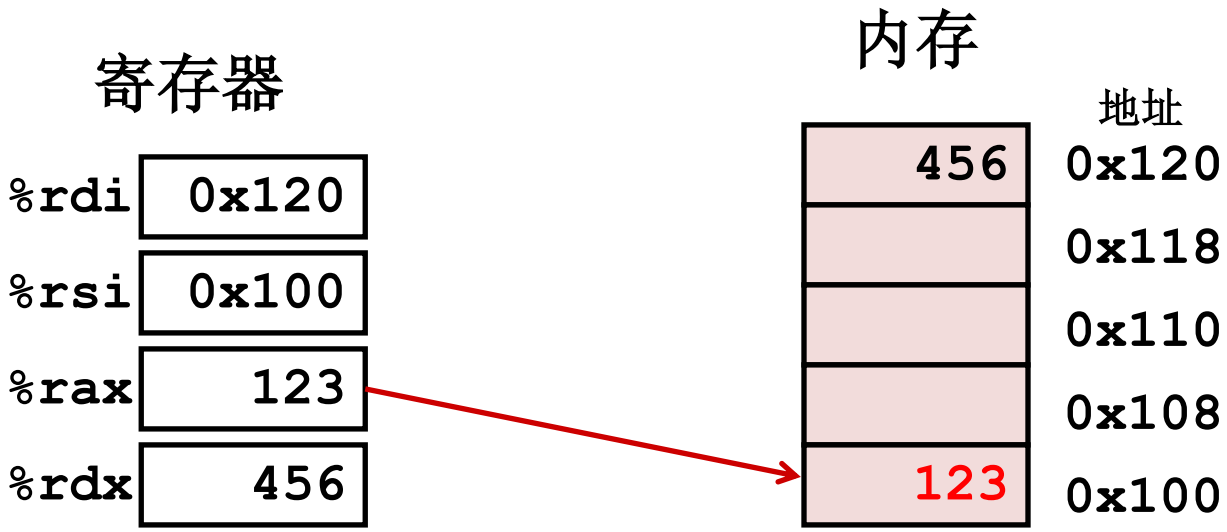

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

swap() 详解




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

swap() 详解




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

简单的内存寻址方式

- 普通：　　(*R*)　　Mem[Reg[*R*]]
  - 寄存器 *R* 指明内存地址
  - C 的指针解引用

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

- 相对：　　*D*(*R*)　　Mem[Reg[*R*]+*D*]
  - 寄存器 *R* 指明内存中的起始地址
  - 常数 *D* 为相对偏移量

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

完整的内存寻址方式

- 一般形式

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

  - *D*：常数相对偏移量，1、2 或 4 字节
  - *Rb*：基址寄存器，16 个整数寄存器之一
  - *Ri*：变址寄存器，除 `%rsp` 之外的任一整数寄存器
  - *S*：比例因子，1、2、4 或 8（为何是这些数？）

- 特例

  (*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*]]

地址计算举例

%rdx	0xf000
%rcx	0x0100

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

本课内容

- Intel 处理器及其架构沿革
- C、汇编和机器代码
- 汇编语言基础：寄存器、操作数、move 指令族
- **算术逻辑操作**

地址计算指令

- `leaq Src,Dst`
  - Src 为寻址方式表达式
  - 将 *Dst* 的内容设置为 *Src* 表示的地址
- 应用
  - 完成地址计算（无需访存）
    - 例：对语句 `p = &x[i];` 的翻译
  - 计算形如 *x* + *k* * *y* 的算术表达式
    - k = 1, 2, 4, or 8
- 举例

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

编译后的结果：

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

常见算术逻辑操作

- 双操作数指令

指令格式		所作的运算	
`addq`	Src,Dst	*Dst = Dst + Src*	
`subq`	Src,Dst	*Dst = Dst – Src*	
`imulq`	Src,Dst	*Dst = Dst * Src*	
`salq`	Src,Dst	*Dst = Dst << Src*	又称 `shlq`
`sarq`	Src,Dst	*Dst = Dst >> Src*	算术移位
`shrq`	Src,Dst	*Dst = Dst >> Src*	逻辑移位
`xorq`	Src,Dst	*Dst = Dst ^ Src*	
`andq`	Src,Dst	*Dst = Dst & Src*	
`orq`	Src,Dst	*Dst = Dst | Src*	

- 注意操作数的顺序！
- 对于 `signed` 和 `unsigned int` 没有区别（为什么？）
  - `sarq` 和 `shrq` 除外

常见算术逻辑操作（续）

- **单操作数指令**

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

- **更多常见指令见教材**

- **全部指令见 Intel 处理器手册**

算术表达式举例

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

有用的指令

- leaq：地址计算指令
- salq：移位指令
- imulq：乘法指令
  - But, only used once

算术表达式举例（续）

```
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;
}
```

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

```
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
```

机器级编程 I：小结

- Intel 处理器及其架构发展演进
  - 进化式的设计导致其中包含了许多奇怪的特性
- C、汇编、机器代码
  - New forms of visible state: program counter, registers, ...
  - 编译器须将语句、表达式、过程翻译成底层的指令序列
- 汇编语言基础：寄存器、操作数、move 指令族
  - x86-64 的 move 类指令涵盖了各种数据传送形式
- 算术逻辑操作
  - 针对不同的算术逻辑操作，C 编译器会生成对应的指令序列