第3章 程序的机器级表示
Machine-Level Programming I: Basics

100076202： 计算机系统导论
I： 基础
I: Basics

任课教师：
计卫星 宿红毅 张艳

原作者：
Randal E. Bryant and David R. O’Hallaron

议题: 程序的机器级表示I：基础
Machine Programming I: Basics

- **Intel处理器和体系结构历史 History of Intel processors and architectures**
- 汇编语言基础：寄存器、操作数、传送类指令
  Assembly Basics: Registers, operands, move
- 算术和逻辑运算 Arithmetic & logical operations
- C、汇编和机器代码 C, assembly, machine code

Intel x86处理器 Intel x86 Processors

- **主导了笔记本/台式机/服务器的市场 Dominate laptop/desktop/server market**
- **不断进化的设计 Evolutionary design**
  - 从1978年引入的8086开始，实现了后向兼容 Backwards compatible up until 8086, introduced in 1978
  - 随着时间的推移增加更多的功能 Added more features as time goes on
- **复杂指令集计算机CISC Complex instruction set computer (CISC)**
  - 很多不同指令有不同的格式 Many different instructions with many different formats
    - 但Linux程序仅使用其中小的子集 But, only small subset encountered with Linux programs
  - 很难匹敌精简指令集计算机RISC的性能 Hard to match performance of Reduced Instruction Set Computers (RISC)
  - 但是，Intel已经这么做了 But, Intel has done just that!
    - 速度方面提升很快，但是很少用于低功耗环境 In terms of speed. Less so for low power.

Intelx86的演进：里程碑
Intel x86 Evolution: Milestones

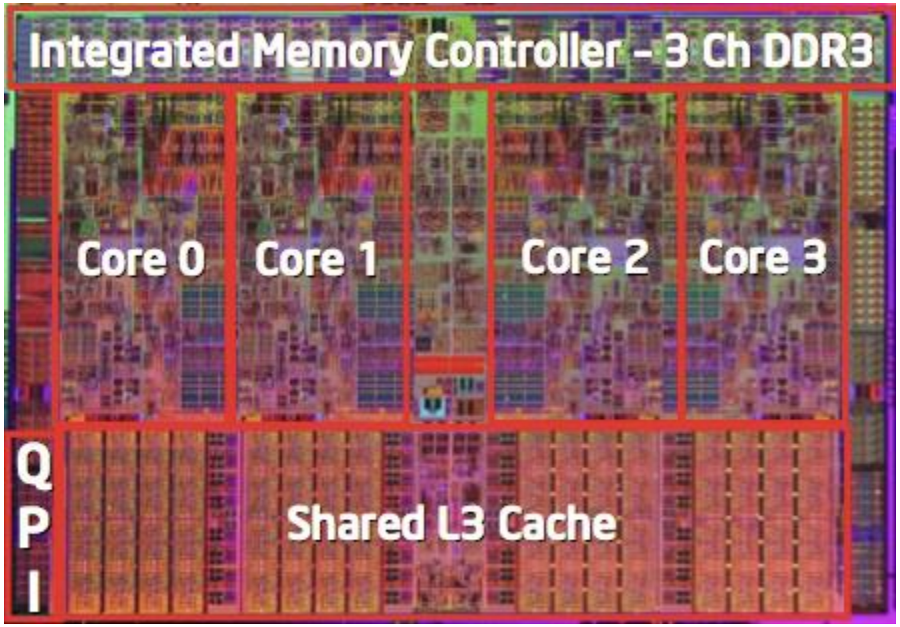
Name	*Date*	*Transistors晶体管*	*MHz*
8086	**1978**	**29K**	**5-10**
386	**1985**	**275K**	**16-33**
Pentium 4E	**2004**	**125M**	**2800-3800**
Core 2	**2006**	**291M**	**1060-3500**
Core i7	**2008**	**731M**	**1700-3900**

- **8086** — **1978** — **29K** — **5-10**
  - First 16-bit Intel processor. Basis for IBM PC & DOS
  - 1MB address space
- **386** — **1985** — **275K** — **16-33**
  - First 32 bit Intel processor , referred to as IA32
  - Added “flat addressing”, capable of running Unix
- **Pentium 4E** — **2004** — **125M** — **2800-3800**
  - First 64-bit Intel x86 processor, referred to as x86-64
- **Core 2** — **2006** — **291M** — **1060-3500**
  - First multi-core Intel processor
- **Core i7** — **2008** — **731M** — **1700-3900**
  - Four cores (our shark machines)

Intel x86 Processors, cont.

- **机器进化 Machine Evolution**

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



- **增加的功能 Added Features**
  - 支持多媒体操作的指令 Instructions to support multimedia operations
  - 使能更多高效条件操作的指令 Instructions to enable more efficient conditional operations
  - 从32位向64位转换 Transition from 32 bits to 64 bits
  - 更多的核 More cores

Intel x86 Processors, cont.

- **前辈 Past Generations** — **Process technology**
  - 1st Pentium Pro — 1995 — 600 nm
  - 1st Pentium III — 1999 — 250 nm
  - 1st Pentium 4 — 2000 — 180 nm
  - 1st Core 2 Duo — 2006 — 65 nm
- **最近和下一代**
- **Recent & Upcoming Generations**

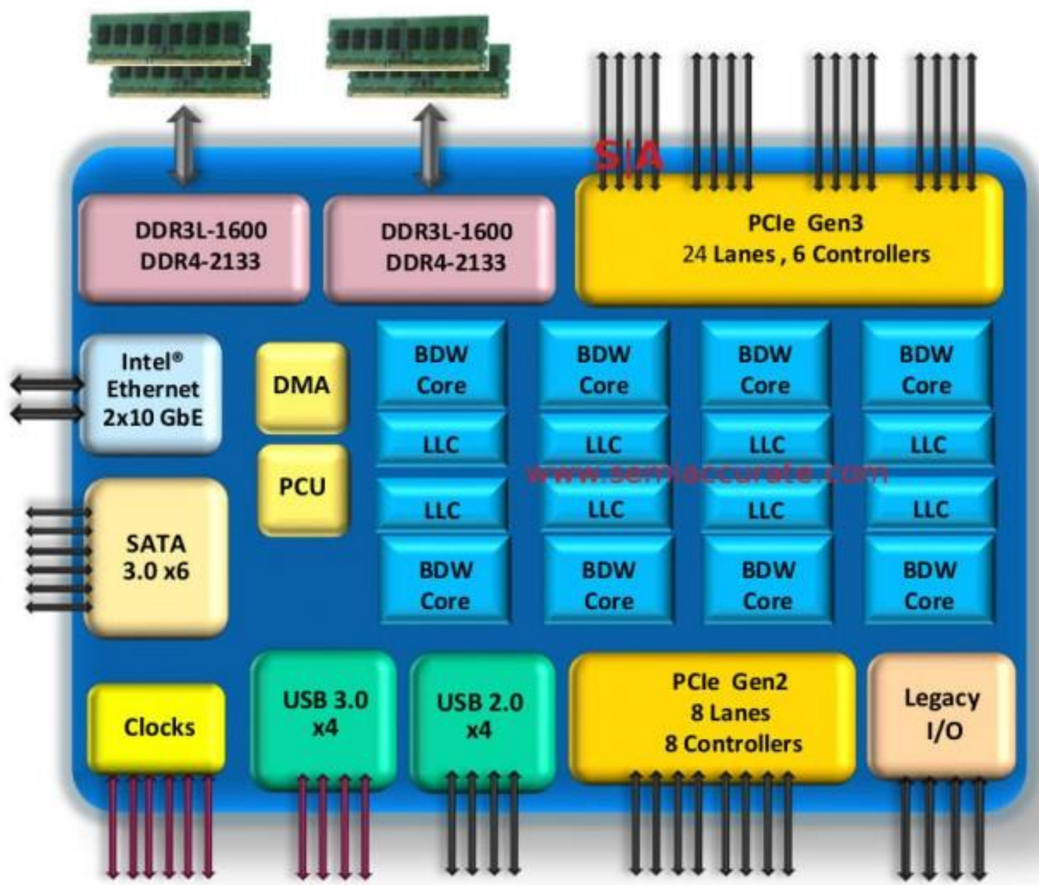
1.	Nehalem	2008	45 nm
2.	Sandy Bridge	2011	32 nm
3.	Ivy Bridge	2012	22 nm
4.	Haswell	2013	22 nm
5.	Broadwell	2014	14 nm
6.	Skylake	2015	14 nm
7.	Kaby Lake	2016	14 nm
8.	Coffee Lake	2017	14 nm
9.	Cannon Lake	2018	10 nm
10.	Ice Lake	2019	10 nm
11.	Tiger Lake	2020	10 nm
12.	Alder Lake	2022	“intel 7” (10nm+++)

处理器技术维度=最窄线宽
(10纳米 ≈ 100原子宽)
Process technology dimension
= width of narrowest wires
(10 nm ≈ 100 atoms wide)

(但现在在变化)
(But this is changing now.)

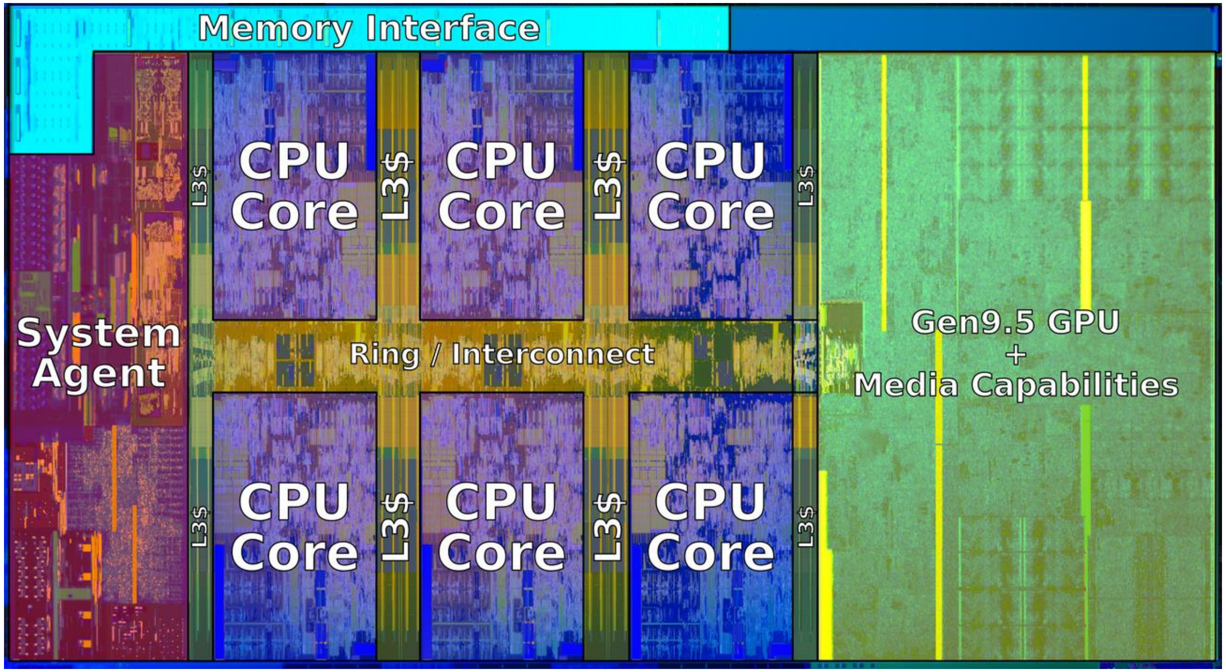
2015年最先进的处理器 2015 State of the Art

- **Core i7 Broadwell 2015**
  - 第五代处理器架构
- **台式机模型 Desktop Model**
  - 4核 4 cores
  - 集成图形卡Integrated graphi
  - 3.3-3.8 GHz
  - 65W
- **服务器模型 Server Model**
  - 8核 8 cores
  - 集成I/O Integrated I/O
  - 2-2.6 GHz
  - 45W



2018年最先进的处理器

2018 State of the Art: Coffee Lake



- **Mobile Model: Core i7**
  - 2.2-3.2 GHz
  - 45 W
- **Desktop Model: Core i7**
  - Integrated graphics
  - 2.4-4.0 GHz
  - 35-95 W
- **Server Model: Xeon E**
  - Integrated graphics
  - Multi-socket enabled
  - 3.3-3.8 GHz
  - 80-95 W

x86克隆： AMD
x86 Clones: Advanced Micro Devices (AMD)

- **历史 Historically**
  - 紧跟Intel AMD has followed just behind Intel
  - 稍微慢一点，便宜很多 A little bit slower, a lot cheaper
- **后来 Then**
  - 从DEC公司和其它走下坡路公司招募了顶级电路设计师 Recruited top circuit designers from Digital Equipment Corp. and other downward trending companies
  - 构建了皓龙：奔腾4的强力竞争者 Built Opteron: tough competitor to Pentium 4
  - 开发了x86-64，扩展为64位体系结构 Developed x86-64, their own extension to 64 bits
-

x86克隆： AMD
x86 Clones: Advanced Micro Devices (AMD)

- **最近几年 Recent Years**
  - Intel齐心协力 Intel got its act together
    - 1995-2011:全球领先的半导体“晶圆厂” Lead semiconductor “fab” in world
    - 2018: 第二大厂 #2 largest by $$ (第一是三星 #1 is Samsung)
    - 2019: 重新夺回第一 reclaimed #1
  - AMD落后：从GlobalFoundaries分拆出来 AMD fell behind: Spun off GlobalFoundaries
  - 2019-20:领先！将台积电作为部分晶圆厂 Pulled ahead! Used TSMC for part of fab
  - 2022: 英特尔重新领先 Intel re-took the lead

Intel的64位处理器历史 Intel’s 64-Bit History

- **2001年Intel尝试从IA32向IA64的激进迁移 2001: Intel Attempts Radical Shift from IA32 to IA64**
  - 完全不同的体系结构（安腾）Totally different architecture (Itanium)
  - 仅仅作为遗留执行IA32 代码 Executes IA32 code only as legacy
  - 性能令人失望 Performance disappointing
- **2003年AMD采用革命性解决方案步伐 2003: AMD Steps in with Evolutionary Solution**
  - x86-64 (现在称为AMD64 now called “AMD64”)
- **Intel感到必须要聚焦到IA64 Intel Felt Obligated to Focus on IA64**
  - 很难承认犯错或AMD更佳 Hard to admit mistake or that AMD is better
- **2004年Intel宣布EM64T扩展到IA32 2004: Intel Announces EM64T extension to IA32**
  - 扩展64位内存技术Extended Memory 64-bit Technology
  - 几乎和x86-64相同 Almost identical to x86-64!
- **除了低端都支持x86-64 All but low-end x86 processors support x86-64**
  - 但很多代码仍然运行在32位模式 But, lots of code still runs in 32-bit mode

我们课程主要讲授 Our Coverage

- **IA32**
  - 传统的x86 The traditional x86
  - 第二版
- **x86-64**
  - 目前的标准 The standard
  - `shark> gcc hello.c`
  - `shark> gcc -m64 hello.c`

- **幻灯片 Presentation**
  - 教材采用x86-64 Book covers x86-64
  - 网站旁注包含IA32 Web aside on IA32
  - 我们仅讨论x86-64 We will only cover x86-64

议题: 程序的机器级表示I：基础
Machine Programming I: Basics

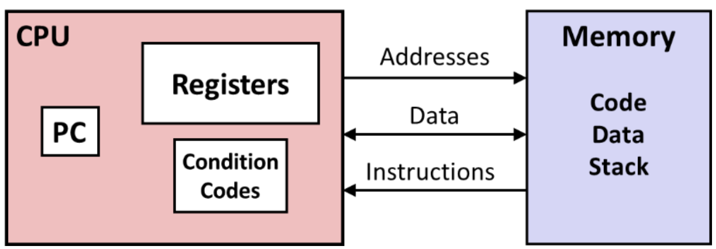
- Intel处理器和体系结构的历史 History of Intel processors and architectures
- **汇编语言基础：寄存器、操作数和传送类指令 Assembly Basics: Registers, operands, move**
- 算术和逻辑运算 Arithmetic & logical operations
- C、汇编和机器代码 C, assembly, machine code

抽象级别 Levels of Abstraction

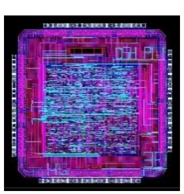
C语言程序员
C programmer

```c
#include <stdio.h>
int main(){
  int i, n = 10, t1 = 0, t2 = 1, nxt;
  for (i = 1; i <= n; ++i){
    printf("%d, ", t1);
    nxt = t1 + t2;
    t1 = t2;
    t2 = nxt; }
  return 0; }
```

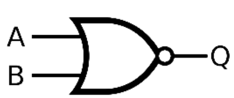
汇编语言程序员
Assembly programmer

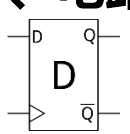


计算机设计师 Computer Designer



门电路、时钟、电路布局。。。 Gates, clocks, circuit layout, ...

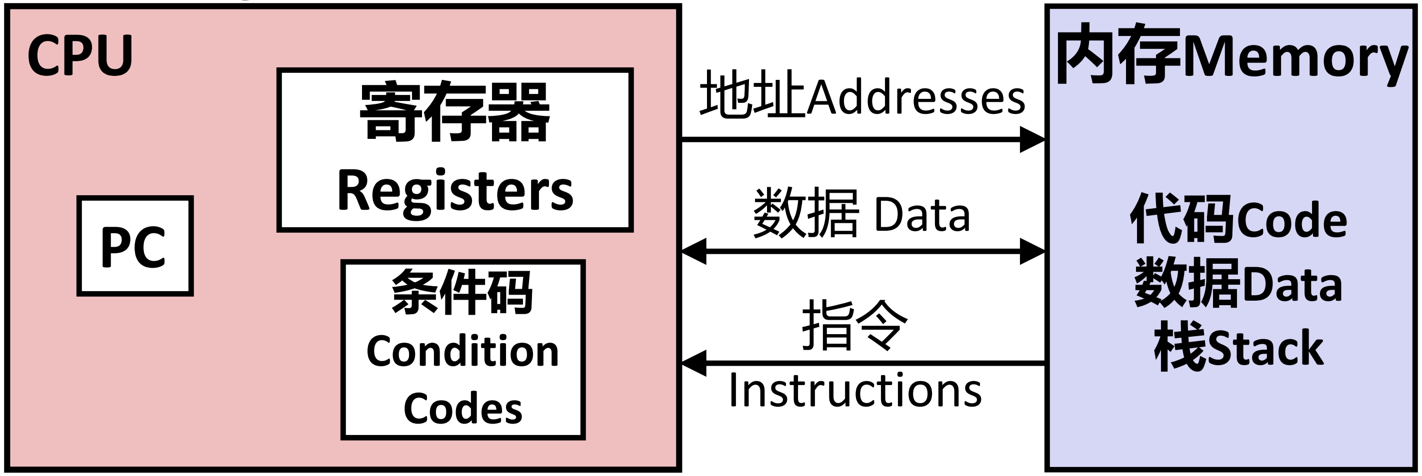




定义 Definitions

- **体系结构(又称ISA：指令集体系结构) 要进行处理器设计，需要理解或者写汇编/机器代码 Architecture: (also ISA: instruction set architecture) The parts of a processor design that one needs to understand or write assembly/machine code.**
  - 例如：指令集规范、寄存器 Examples: instruction set specification, registers.
- **微体系结构：体系结构的实现 Microarchitecture: Implementation of the architecture.**
  - 例如：cache大小和核心频率 Examples: cache sizes and core frequency.
- **代码形式 Code Forms:**
  - 机器代码：处理器执行的字节级程序 Machine Code: The byte-level programs that a processor executes
  - 汇编代码：机器代码的文本表示 Assembly Code: A text representation of machine code
- **示例ISA Example ISAs: Intel: x86, IA32, Itanium, x86-64**
  - ARM: 用于几乎所有移动电话 Used in almost all mobile phones
  - RISC V: 新的开源ISA New open-source ISA

汇编/机器代码视图 Assembly/Machine Code View



程序员可见的状态Programmer-Visible State

- **PC程序计数器 PC: Program counter**
  - 下条指令地址 Address of next instruction
  - 称为RIP Called “RIP” (x86-64)
- **寄存器堆 Register file**
  - 程序频繁使用的数据 Heavily used program data
- **条件码 Condition codes**
  - 存储有关最近的算术和逻辑运算的状态信息 Store status information about most recent arithmetic or logical operation
  - 用于条件分支 Used for conditional branching
- **内存 Memory**
  - 字节可寻址的数组 Byte addressable array
  - 代码和用户数据Code and user data
  - 支持过程的栈 Stack to support procedures

汇编语言：数据类型
Assembly: Data Types

- **1,2,4或8字节“整数”数据 “Integer” data of 1, 2, 4, or 8 bytes**
  - 数据值 Data values
  - 地址（无类型指针） Addresses (untyped pointers)
- **4,8或10字节的浮点数据 Floating point data of 4, 8, or 10 bytes**
- **（SIMD向量数据类型8、16、32或64字节）(SIMD vector data types of 8, 16, 32 or 64 bytes)**
- **代码：字节序列编码一序列指令 Code: Byte sequences encoding series of instructions**
- **没有聚合类型例如数组或结构 No aggregate types such as arrays or structures**
  - 仅在内存中分配连续的字节 Just contiguously allocated bytes in memory

寄存器名字 Register names

```
addq    %rbx,  %rax
```

相当于 is

```
rax += rbx
```

这些是64位寄存器，因此我们认为这是64位加法 These are 64-bit registers, so we know this is a 64-bit add

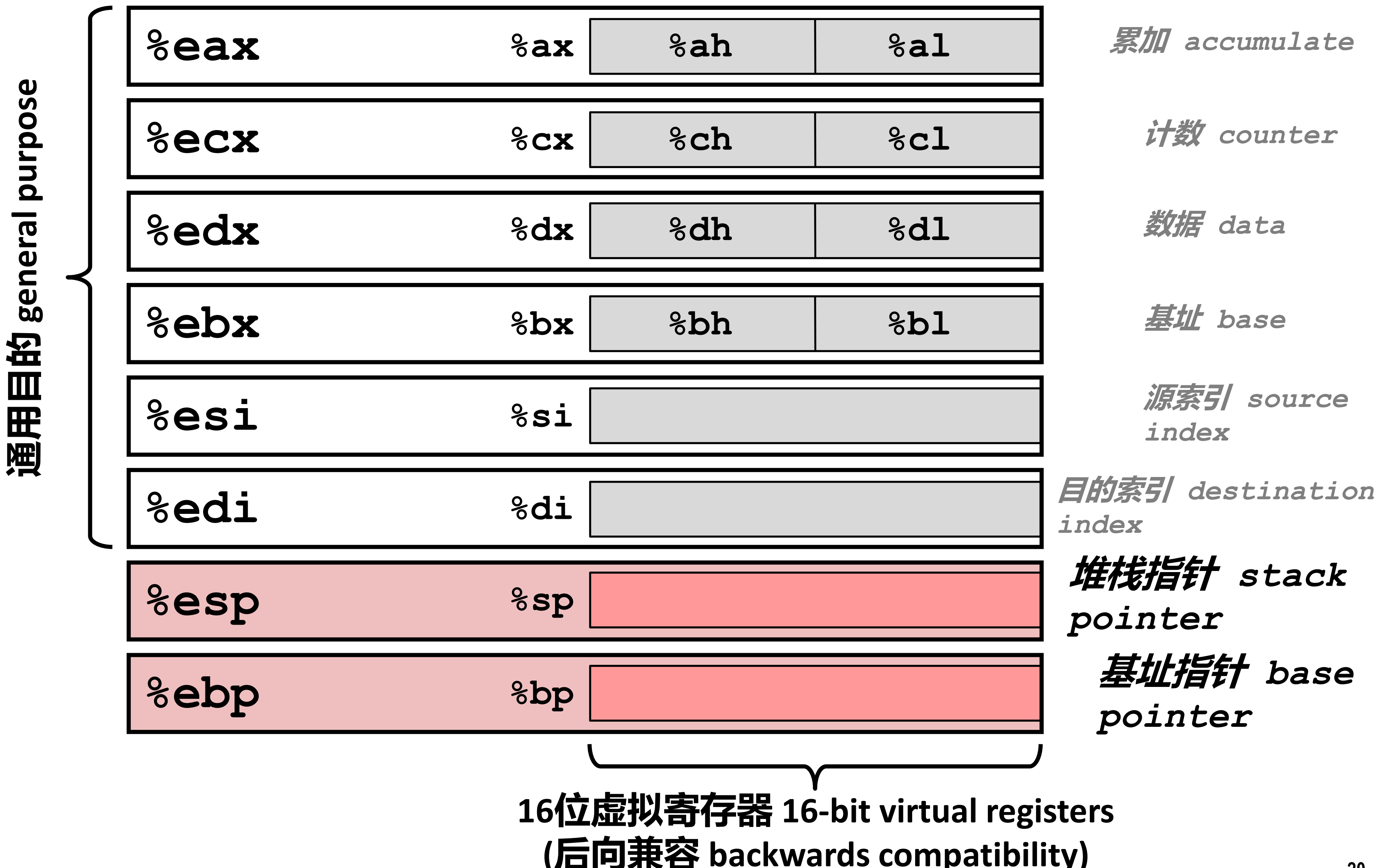
x86-64 Integer Registers 整数寄存器

%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

- 可以引用低4字节（也可以引用低1或2字节） Can reference low-order 4 bytes (also low-order 1 & 2 bytes)
- 不是内存（或cache）的一部分 Not part of memory (or cache)

一些历史：IA32寄存器
Some History: IA32 Registers

原始 Origin
(多半已过时
mostly obsolete)



汇编语言：操作 Assembly: Operations

- **在内存和寄存器之间传送数据 Transfer data between memory and register**
  - 从内存装载数据到寄存器 Load data from memory into register
  - 存储寄存器数据到内存 Store register data into memory

- **对寄存器或内存数据执行算术运算 Perform arithmetic function on register or memory data**

- **传递控制 Transfer control**
  - 无条件跳转到/从过程 Unconditional jumps to/from procedures
  - 条件分支 Conditional branches
  - 间接分支 Indirect branches

传送数据 Moving Data

- **传送数据 Moving Data**

  `movq` *Source, Dest*:

- **操作数类型 Operand Types**
  - ***立即数*: 常量整数 *Immediate:*** Constant integer data
    - 例如：Example: `$0x400, $-533`
    - 类似C常量，**$前缀** Like C constant, but prefixed
    - 编码1，2或4字节 Encoded with 1, 2, or 4 bytes
  - ***寄存器*: 16个整数寄存器之一 *Register:*** One of 16
    - 例如：Example: `%rax`，`%r13`
    - 但`%rsp`保留有特殊用处 But `%rsp` reserved f
    - 其它对特定指令有特殊用途 Others have special uses for particular instructions
  - ***内存*: 8个连续的字节，由寄存器给出内存地址 *Memory:*** 8 consecutive bytes of memory at address given by register
    - 最简单的例子：Simplest example: `(%rax)`
    - 各种其它"寻址方式" Various other "address modes"

Warning: Intel docs use mov *Dest, Source*

`%rax`
`%rcx`
`%rdx`
`%rbx`
`%rsi`
`%rdi`
`%rsp`
`%rbp`
`%rN`

Movq指令操作数组合
movq Operand Combinations

	Source	Dest	Src,Dest	C Analog类比
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;`

单条指令中不能进行内存到内存的传送

Cannot do memory-memory transfer with a single instruction

简单内存寻址方式 Simple Memory Addressing Modes

- **正常 Normal (R) Mem[Reg[R]]**
  - 寄存器R指定内存地址 Register R specifies memory address
  - C语言中的指针间接引用 Aha! Pointer dereferencing in C

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

- **变址 Displacement D(R) Mem[Reg[R]+D]**
  - 寄存器R指定内存区域的起始地址 Register R specifies start of memory region
  - 常量D指定偏移 Constant displacement D specifies offset

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

简单寻址方式示例
Example of Simple Addressing Modes

```
void
whatAmI(<type> a, <type> b)
{
      ????
}
```

%rdi %rsi

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

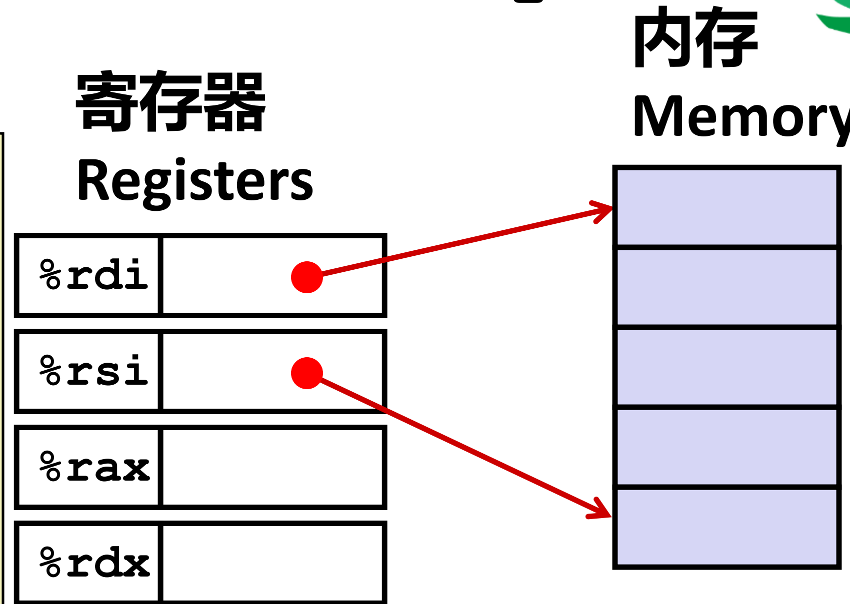
简单寻址方式示例
Example of Simple Addressing Modes

```c
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() Understanding 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() Understanding Swap()

寄存器 Registers

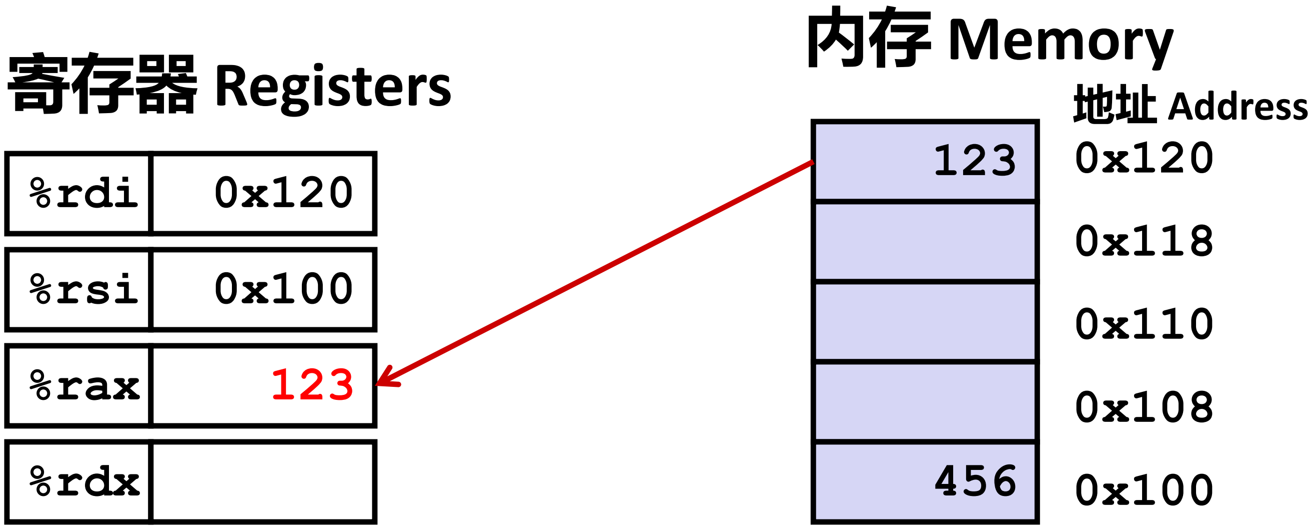
%rdi	0x120
%rsi	0x100
%rax	
%rdx	

内存 Memory

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

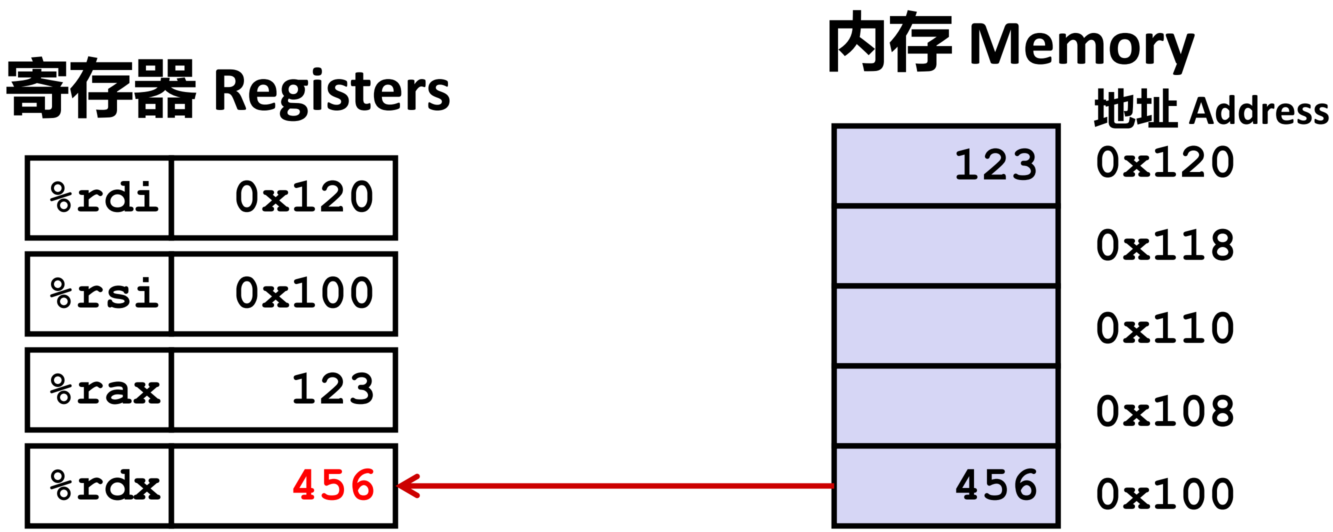
```
swap:
   movq    (%rdi), %rax   # t0 = *xp
   movq    (%rsi), %rdx   # t1 = *yp
   movq    %rdx, (%rdi)   # *xp = t1
   movq    %rax, (%rsi)   # *yp = t0
   ret
```

理解Swap() Understanding Swap()



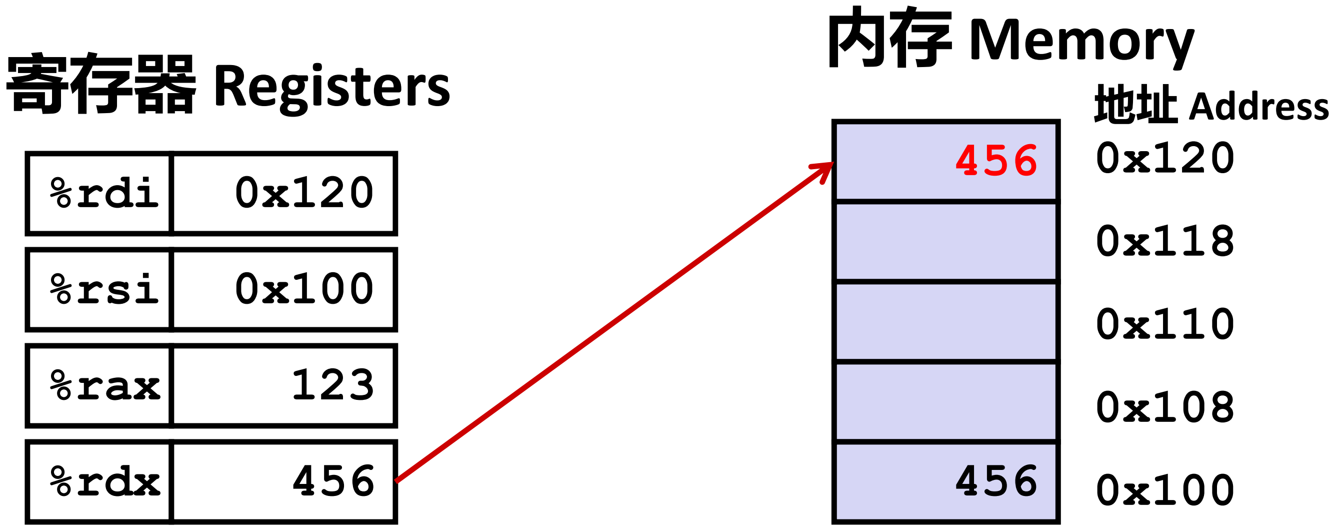

```
swap:
    movq     (%rdi), %rax   # t0 = *xp
    movq     (%rsi), %rdx   # t1 = *yp
    movq     %rdx, (%rdi)   # *xp = t1
    movq     %rax, (%rsi)   # *yp = t0
    ret
```

理解Swap() Understanding Swap()



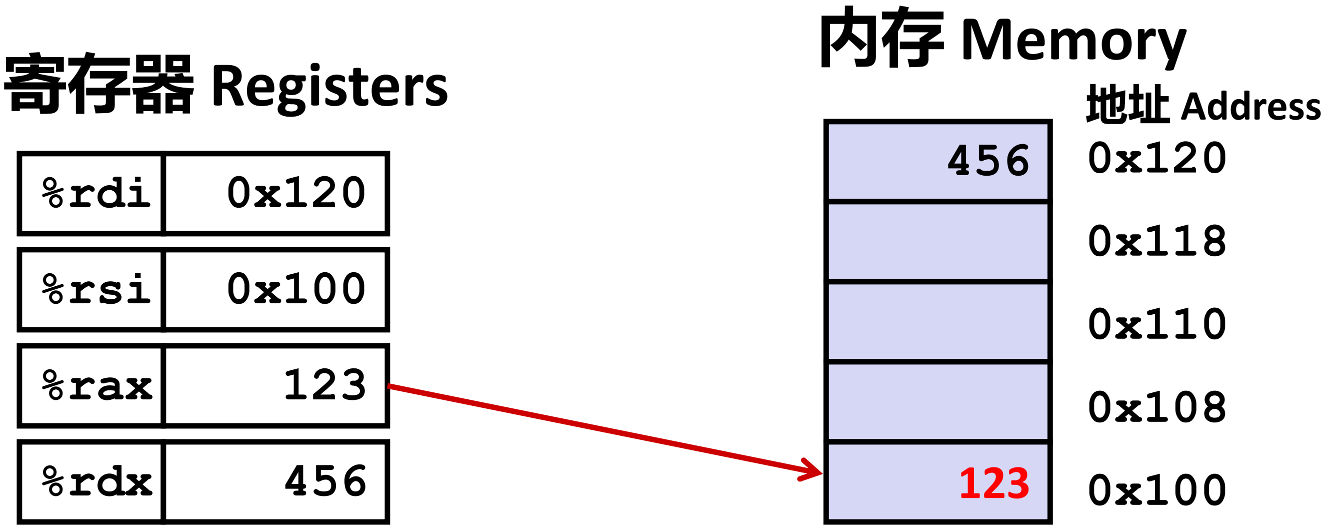

```
swap:
    movq     (%rdi), %rax   # t0 = *xp
    movq     (%rsi), %rdx   # t1 = *yp
    movq     %rdx, (%rdi)   # *xp = t1
    movq     %rax, (%rsi)   # *yp = t0
    ret
```

理解Swap() Understanding 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() Understanding Swap()




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

简单内存寻址方式 Simple Memory Addressing Modes

- **正常 Normal (R) Mem[Reg[R]]**
  - 寄存器R指定内存地址 Register R specifies memory address
  - C语言中的指针间接引用 Aha! Pointer dereferencing in C

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

- **变址 Displacement D(R) Mem[Reg[R]+D]**
  - 寄存器R指定内存区域的起始地址 Register R specifies start of memory region
  - 常量D指定偏移 Constant displacement D specifies offset

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

复杂内存寻址方式

Complete Memory Addressing Modes

- **最通用形式 Most General Form**

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

  - D: 常量“变址”1,2或4字节 Constant “displacement” 1, 2, or 4 bytes
  - Rb: 基指针： 16个整数寄存器中任意一个 Base register: Any of 16 integer registers
  - Ri: 索引寄存器：任意，除了%rbp Index register: Any, except for `%rsp`
  - S: 比例因子：1,2,4或8(为何这几个数) Scale: 1, 2, 4, or 8 (*why these numbers?*)

- **特殊情况 Special Cases**

  (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
Address Computation Examples

`%rdx`	`0xf000`
`%rcx`	`0x0100`

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

- D: Constant “displacement” 1, 2, or 4 bytes
- Rb: Base register: Any of 16 integer registers
- Ri: Index register: Any, except for `%rsp`
- S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

表达式 Expression	地址计算 Address Computation	地址 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`

简单的数据传送指令

Instruction		Effect	Description
MOV	S, D	$D \leftarrow S$	Move
movb			Move byte
movw			Move word
movl			Move double word
movq			Move quad word
movabsq	I, R	$R \leftarrow I$	Move absolute quad word

- mov (general)
- movb (move byte：8位)
- movw (move word：一个字16位)
- movl (move double word：双字32位)
- movq (move quad word：四字64位)
  - 立即数仅表示为32位有符号数，经符号扩展传送到目的 Immediate can only be represented as 32-bit signed sign extension to the destination
- movabsq (move absolute quad word)

- **仅更新指定寄存器字节 Only update the specific register byte**
  - 除了movl指令，它将寄存器的高四个字节更新为0 except movl which updates the higher-order 4 bytes of the register to 0

数据传送示例 Data Movement Example

movl	$0x4050, %eax	immediate	register, 4 bytes
movw	%bp, %sp	register	register, 2 bytes
movb	(%rdi, %rcx), %al	memory	register, 1 byte
movb	$-17, (%rsp)	immediate	memory, 1 byte
movq	%rax, -12(%rbp)	register	memory, 8 byte

零扩展 数据传送指令

Instruction	Effect	Description
MOVZ S, R	$R \leftarrow \mathrm{ZeroExtend}(S)$	Move with zero extension
movzbw		Move zero-extended byte to word
movzbl		Move zero-extended byte to double word
movzwl		Move zero-extended word to double word
movzbq		Move zero-extended byte to quad word
movzwq		Move zero-extended word to quad word

- **没有movzlq指令，这个功能由movl指令实现 no movzlq, it is implemented by movl**

符号扩展 数据传送指令

Instruction	Effect	Description
MOVS S, R	$R \leftarrow \text{SignExtend}(S)$	Move with sign extension
`movsbw`		Move sign-extended byte to word
`movsbl`		Move sign-extended byte to double word
`movswl`		Move sign-extended word to double word
`movsbq`		Move sign-extended byte to quad word
`movswq`		Move sign-extended word to quad word
`movslq`		Move sign-extended double word to quad word
`cltq`	`%rax` $\leftarrow$ SignExtend(`%eax`)	Sign-extend `%eax` to `%rax`

示例 Example

```
long exchange(long *xp, long y)
{
        long x = *xp ;
        *xp = y ;
        return x ;
}
```

xp in %rdi, y in %rsi

```
1.  exchange:
2.      movq (%rdi), %rax       Get x at xp. Set as return value
3.      movq %rsi, (%rdi)       Store y at xp
4.      ret                     return
```

汇编语言的数组 Array in Assembly

持续使用 Persistent usage

```
long a[16];
void f(void){
    long i ;
    for(i=0; i<16; i++)
       a[i]=i;
}
movq %rdx, a(,%rdx,8)

i:    %rdx
```

数据传送示例 Data Movement Example

初始值 Initial value %dl=8d   %rax =0000000098765432

1	**movb**	**%dl, %al**	**%rax=000000009876548d**
2	**movsbl**	**%dl, %eax**	**%rax=00000000ffffff8d**
3	**movzbq**	**%dl, %rax**	**%rax=000000000000008d**

议题：程序的机器级表示I：基础
Machine Programming I: Basics

- Intel处理器和体系结构历史 History of Intel processors and architectures
- 汇编语言基础：寄存器、操作数和传送类指令 Assembly Basics: Registers, operands, move
- **算术和逻辑运算 Arithmetic & logical operations**
- C语言、汇编和机器代码 C, assembly, machine code

地址计算指令 Address Computation Instruction

- `leaq` ***Src, Dst***
  - *Src*是寻址方式表达式 *Src* is address mode expression
  - 设置Dst成为表达式指示的地址 Set *Dst* to address denoted by expression

- **用法 Uses**
  - 计算地址不必引用内存 Computing addresses without a memory reference
    - 例如 E.g., translation of `p = &x[i];`
  - 计算x + k*y形式的算术表达式 Computing arithmetic expressions of the form x + k*y
    - k = 1, 2, 4, or 8

- **Example**

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

编译器转换成汇编程序

Converted to ASM by compiler:

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

一些算术运算Some Arithmetic Operations

- **双操作数指令 Two Operand Instructions:**

格式*Format* 计算*Computation*

`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shlq *Also called shlq*
`sarq`	*Src,Dest*	Dest = Dest >> Src	算术 *Arithmetic*
`shrq`	*Src,Dest*	Dest = Dest >> Src	逻辑 *Logical*
`xorq`	*Src,Dest*	Dest = Dest ^ Src	
`andq`	*Src,Dest*	Dest = Dest & Src	
`orq`	*Src,Dest*	Dest = Dest | Src	

- **观察参数的顺序！（警告：Intel文档中用法是“op *Dest,Src*”） Watch out for argument order! (Warning: Intel docs use “op *Dest,Src*”)**
- **有/无符号整数之间没有区别（为何？） No distinction between signed and unsigned int (why?)**

一些算术运算 Some Arithmetic Operations

- **单操作数指令 One Operand Instructions**

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

- **参见教材了解更多的指令 See book for more instructions**

算术表达式示例
Arithmetic Expression Example

```
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 Interesting Instructions

- `leaq`: 地址计算 address computation
- `salq`: 移位 shift
- `imulq`: 乘法 multiplication
  - 但是，仅使用一次 But, only used once

理解算术表达式示例
Understanding Arithmetic Expression Example

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

算术和逻辑运算
Arithmetic and Logical Operations

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

寄存器 Register	值 Value
%rax	0x100
%rcx	0x1
%rdx	0x3

指令 Instruction	目的地址 Destination	值 Value
addq %rcx, (%rax)	0x100	0x100
subq %rdx, 8(%rax)	0x108	0xA8
imulq $16, (%rax, %rdx, 8)	0x118	0x110
incq 16(%rax)	0x110	0x14
decq %rcx	%rcx	0x0
subq %rdx, %rax	%rax	0xFD

Lea指令示例
Examples for Lea Instruction

%rax holds x, %rcx holds y

表达式 Expression	结果 Result
leaq 6(%rax), %rdx	6+x
leaq (%rax, %rcx), %rdx	x+y
leaq (%rax, %rcx, 4), %rdx	x+4*y
leaq 7(%rax, %rax, 8), %rdx	7+9*x
leaq 0xA(, %rcx, 4), %rdx	10+4*y
leaq 9(%rax, %rcx, 2), %rdx	9+x+2*y

特殊的算术运算
Special Arithmetic Operations

imulq S	R[%rdx]:R[%rax] ←S*R[%rax]	有符号数完整乘法 Signed full multiply
mulq S	R[%rdx]:R[%rax] ←S*R[%rax]	无符号数完整乘法 Unsigned full multiply
cqto	R[%rdx]:R[%rax] ← SignExtend(R[%rax])	转换成8个字 Convert to oct word
idivq S	R[%rdx] ← R[%rdx]:R[%rax] mod S R[%rax] ← R[%rdx]:R[%rax] ÷ S	有符号数除法 Signed divide
divq S	R[%rdx] ←R[%rdx]:R[%rax] mod S R[%rax] ← R[%rdx]:R[%rax] ÷ S	无符号数除法 Unsigned divide

示例 Examples

Initially dest in %rdi, x in %rsi, y in %rdx (*dest= x*y)

```
1 movq %rsi, %rax
2 mulq %rdx
3 movq %rax, (%rdi)
4 movq %rdx, 8(%rdi)
```

Initially x in %rdi, y in %rsi, qp in %rdx, rp in %rcx (*qp = x / y, *rp = x % y)

```
1 movq %rdx, %r8
2 movq %rdi, %rax
2 cqto
3 idivq %rsi
4 movq %rax, (%r8)
5 movq %rdx, (%rcx)
```

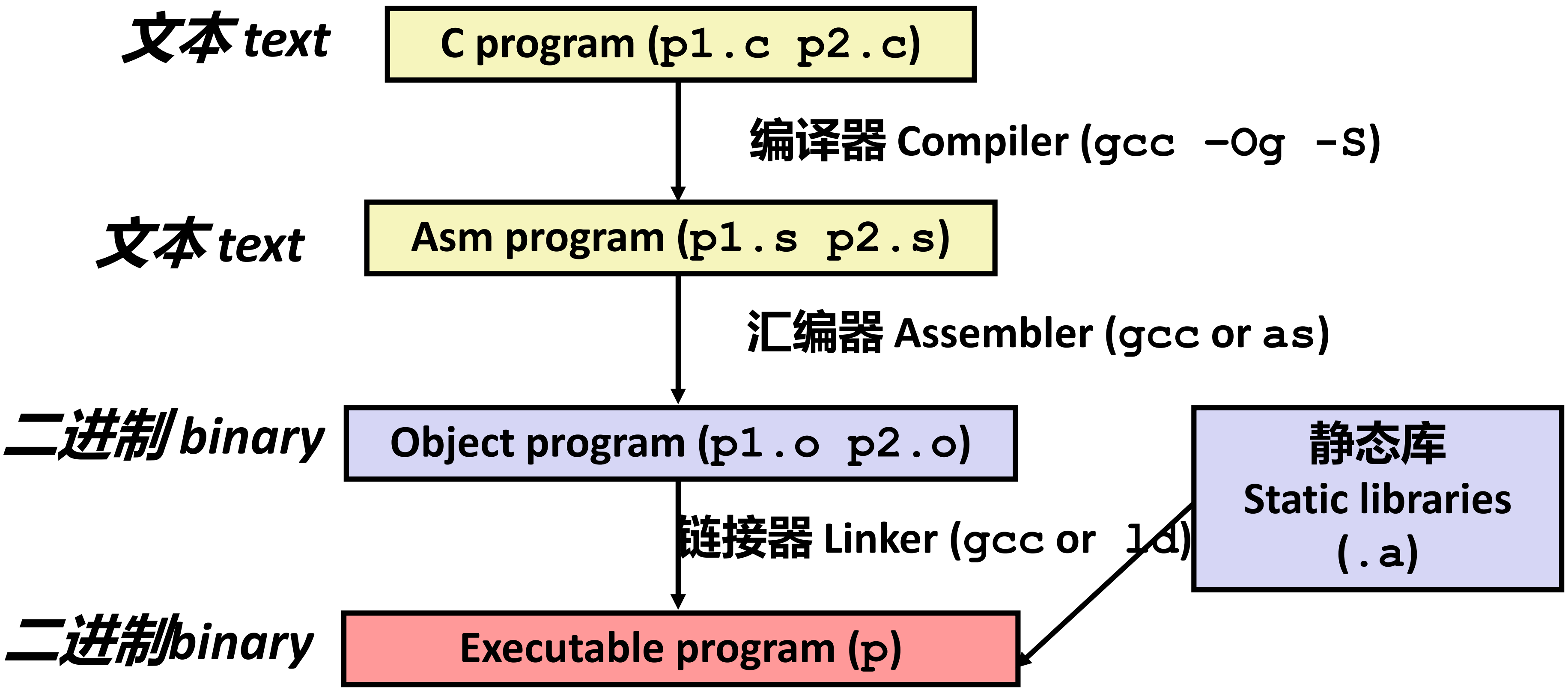

52

议题: 程序的机器级表示I：基础
Machine Programming I: Basics

- Intel处理器和体系结构的历史 History of Intel processors and architectures
- 汇编语言基础：寄存器、操作数和传送类指令 Assembly Basics: Registers, operands, move
- 算术和逻辑运算 Arithmetic & logical operations
- C语言、汇编和机器代码 C, assembly, machine code

转换C源程序为目标代码
Turning C into Object Code

- 代码在文件p1和p2中 Code in files `p1.c p2.c`
- 编译命令 Compile with command: `gcc -Og p1.c p2.c -o p`
  - 基本优化(-Og)Use basic optimizations (`-Og`) [New to recent versions of GCC]
  - 将二进制结果放在文件p中 Put resulting binary in file `p`



编译成汇编程序 Compiling Into Assembly

C Code (sum.c)

```
long plus(long x, long y);

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

Generated x86-64 Assembly

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

用下面命令获得 Obtain (on shark machine) with command

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

生成汇编程序文件 Produces file `sum.s`

_警告_：由于gcc版本和和编译器设置不同在非Shark机器上会得到不同的结果 _Warning_: Will get very different results on non-Shark machines (Andrew Linux, Mac OS-X, …) due to different versions of gcc and different compiler settings.

汇编程序真实的样子

What it really looks like

```
        .globl  sumstore
        .type   sumstore, @function
sumstore:
.LFB35:
        .cfi_startproc
        pushq   %rbx
        .cfi_def_cfa_offset 16
        .cfi_offset 3, -16
        movq    %rdx, %rbx
        call    plus
        movq    %rax, (%rbx)
        popq    %rbx
        .cfi_def_cfa_offset 8
        ret
        .cfi_endproc
.LFE35:
        .size   sumstore, .-sumstore
```

https://sourceware.org/binutils/docs-2.31/as/CFI-directives.html#CFI-directives

汇编程序真实的样子

What it really looks like

```
        .globl  sumstore
        .type   sumstore, @function
sumstore:
.LFB35:
        .cfi_startproc
        pushq   %rbx
        .cfi_def_cfa_offset 16
        .cfi_offset 3, -16
        movq    %rdx, %rbx
        call    plus
        movq    %rax, (%rbx)
        popq    %rbx
        .cfi_def_cfa_offset 8
        ret
        .cfi_endproc
.LFE35:
        .size   sumstore, .-sumstore
```

看起来有些奇怪且“.”作前导的语句一般是伪指令 Things that look weird and are preceded by a ‘.’ are generally directives.

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

目标代码 Object Code

Code for `sumstore`

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

- **总共14字节 Total of 14 bytes**
- **每条指令1,3或5字节 Each instruction 1, 3, or 5 bytes**
- **起始地址为 Starts at address `0x0400595`**

- **汇编器 Assembler**
  - 翻译.s成.o文件 Translates `.s` into `.o`
  - 每条指令二进制编码 Binary encoding of each instruction
  - 接近执行代码的完整映像 Nearly-complete image of executable code
  - 缺少不同文件中代码的链接 Missing linkages between code in different files
- **链接器 Linker**
  - 解析文件之间的引用 Resolves references between files
  - 与静态运行时库组合在一起 Combines with static run-time libraries
    - E.g., code for **`malloc`**, **`printf`**
  - 有些库采用动态链接 Some libraries are *dynamically linked*
    - 当程序开始执行时进行链接 Linking occurs when program begins execution

机器指令示例 Machine Instruction Example

```
*dest = t;
```

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

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

- **C代码 C Code**
  - t的值存储到dest所指地方 Store value `t` where designated by `dest`
- **汇编程序 Assembly**
  - 传送8字节值到内存 Move 8-byte value to memory
    - x86-64用语中为四字 Quad words in x86-64 parlance
  - 操作数 Operands:

`t:`	Register `%rax`
`dest:`	Register `%rbx`
`*dest:`	Memory `M[%rbx]`

- **目标代码 Object Code**
  - 3字节指令 3-byte instruction
  - 存储在地址`0x40059e`处 Stored at address `0x40059e`

反汇编目标代码 Disassembling Object Code

反汇编Disassembled

```
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
```

- **反汇编程序 Disassembler**

  `objdump -d sum`

  - 检查目标代码的有用工具 Useful tool for examining object code
  - 分析一系列指令的比特位模式 Analyzes bit pattern of series of instructions
  - 生成汇编代码的大致格式 Produces approximate rendition of assembly code
  - 可以对a.out（完整的可执行文件）或.o文件运行该程序 Can be run on either `a.out` (complete executable) or `.o` file

替代的反汇编工具 Alternate Disassembly

目标代码Object

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

反汇编Disassembled

```
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
```

- **gdb调试工具中 Within gdb Debugger**

  `gdb sum`

  `disassemble sumstore`

  - 反汇编过程 Disassemble procedure

  `x/14xb sumstore`

  - 检查sumstore开始的14字节 Examine the 14 bytes starting at `sumstore`

反汇编能做什么?
What Can be Disassembled?

```
% 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逆向工程
Reverse engineering forbidden by Microsoft End User License Agreement

- **任何内容可以解释为可执行代码 Anything that can be interpreted as executable code**
- **反汇编器检查字节并重新构造汇编语言源程序 Disassembler examines bytes and reconstructs assembly source**

程序的机器级表示I：小结

Machine Programming I: Summary

- **Intel处理器和体系结构历史 History of Intel processors and architectures**
  - 不断进化的设计导致很多怪异的老古董 Evolutionary design leads to many quirks and artifacts
- **C语言、汇编和机器代码 C, assembly, machine code**
  - 可见状态的新形式：程序计数器、寄存器等 New forms of visible state: program counter, registers, ...
  - 编译器将状态、表达式和过程翻译成低级指令序列Compiler must transform statements, expressions, procedures into low-level instruction sequences
- **汇编语言基础：寄存器、操作数和传送类指令 Assembly Basics: Registers, operands, move**
  - x86-64传送类指令覆盖广泛的数据传送形式 The x86-64 move instructions cover wide range of data movement forms
- **运算 Arithmetic**
  - C语言编译器会算出不同指令以及计算出进位 C compiler will figure out different instruction combinations to carry out computation