清華大學
Tsinghua University

2024秋季

计算机系统概论

Introduction to Computer Systems

80X86汇编语言与C语言-1

韩文弢

hanwentao@tsinghua.edu.cn

计算机体系结构

main.c

```
int array[4] = {1,2,3,4};
static int brray[4] = {1,2,3,4};

static int intra_sum (int x[4],int y)
{
   return x[y-1];
}
int main()
{
    int val = intra_sum(array, 3) + inter_sum(brray,3);
    return val;
}
```

编译

```
intra_sum:
        movslq  %esi, %rsi
        movl    -4(%rdi,%rsi,4), %eax
        ret
main:
        pushq   %rbx
        movl    $3, %esi
        movl    $array, %edi
        call    intra_sum
        movl    %eax, %ebx// val -> ebx
        movl    $3, %esi
        movl    $brray, %edi
        movl    $0, %eax
        call    inter_sum
        addl    %ebx, %eax// val -> eax
        popq    %rbx
        ret
brray:
        .long   1
        .long   2
        .long   3
        .long   4
array:
        .long   1
        .long   2
        .long   3
        .long   4
```

汇编指令

main.o

链接

```
00000000004004de <main>:
  4004de:       53                      push   %rbx
  4004df:       be 03 00 00 00          mov    $0x3,%esi
  4004e4:       bf 40 10 60 00          mov    $0x601040,%edi
  4004e9:       e8 e8 ff ff ff          callq  4004d6 <intra_sum>
  4004ee:       89 c3                   mov    %eax,%ebx
  4004f0:       be 03 00 00 00          mov    $0x3,%esi
  4004f5:       bf 30 10 60 00          mov    $0x601030,%edi
  4004fa:       b8 00 00 00 00          mov    $0x0,%eax
  4004ff:       e8 04 00 00 00          callq  400508 <inter_sum>
  400504:       01 d8                   add    %ebx,%eax
  400506:       5b                      pop    %rbx
  400507:       c3                      retq
```

仅给出部分内容

内存地址

运行

CPU

P C Registers Condition Codes

C程序在硬件层面的表示

- 数据
  - ……
- 代码
  - 基本概念/基本指令/寻址方式 / 程序控制流与相关指令（第三讲）
  - ……

地址/Addresses

数据/Data

指令/Instructions

Memory

数据段

代码段

程序/数据在硬件层面的表示与运行

目 录

CONTENTS

需要掌握的要点

程序的语义等价转换：C语言 → 汇编语言

- 包括但不限于：基本计算、控制流、循环、函数调用与返回、数组/结构等复合数据类型访问等

（部分）指令/函数/数据的“定位”

- 局部变量
- 静态函数入口地址
- 函数的参数、返回值
- （条件/无条件）跳转指令的目的
- 函数返回地址

代码优化……

```
int array[4] = {1,2,3,4};
static int brray[4] = {1,2,3,4};
static int intra_sum (int x[4],int y)
{
   return x[y-1];
}
```

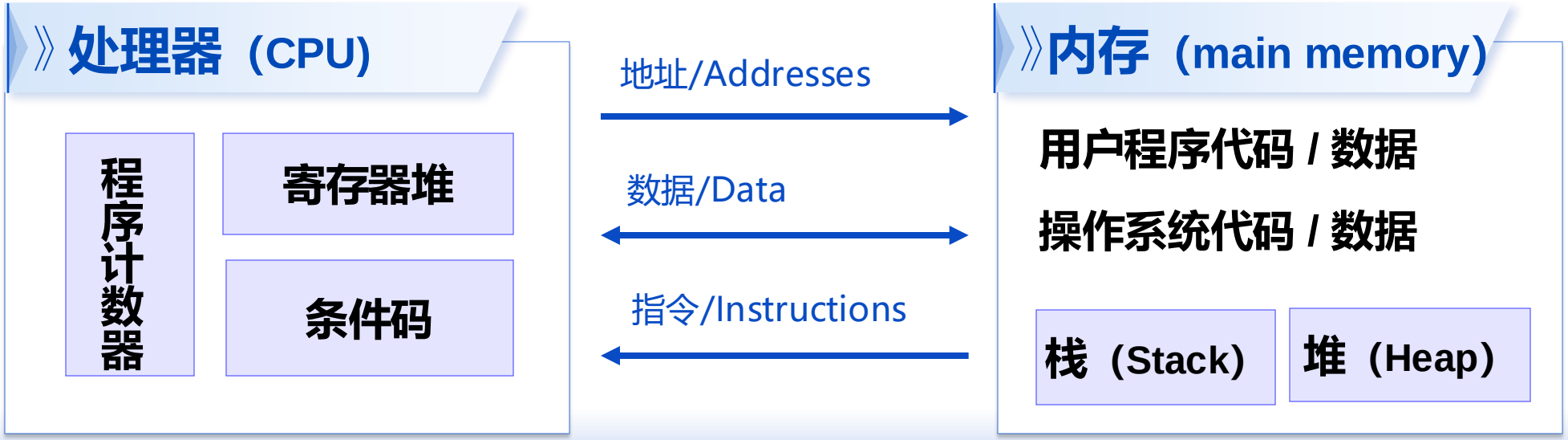
main.c

```
int main()
{
    int val = intra_sum(array, 3) +
              inter_sum(brray,3);

    if (val>=0)
       return val;
    else
       return -1;
}
```

```
    cmpl    $0, -20(%rbp)
    js      .L4
    movl    -20(%rbp), %eax
    jmp     .L5
.L4:
    movl    $-1, %eax
.L5:
    .....
```

汇编程序员眼中的体系结构 冯诺依曼架构，又称为控制流（control flow）架构



程序计数器 （PC）

- 存储下一条指令的地址
- x86-64 架构下的名字为RIP

寄存器/寄存器堆（register file）

- 在处理器内部的以名字来访问的快速存储单元

条件码（condition codes）

- 用于存储最近执行指令的结果状态信息
- 用于条件指令的判断执行

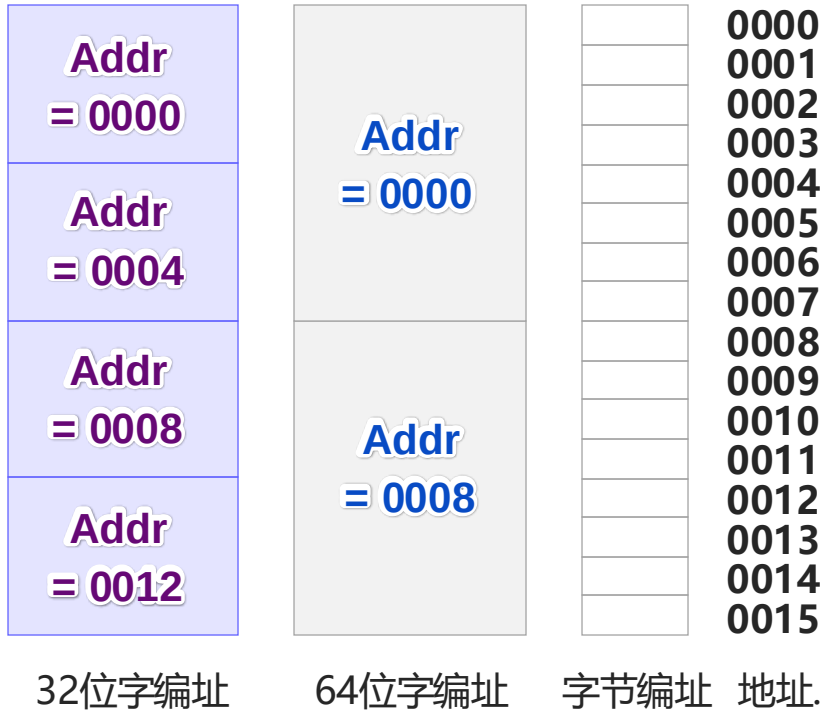
内存（main memory）

- 以字节编码的连续存储空间
- 存放程序代码、数据、栈与堆，以及操作系统代码与数据(OS data)

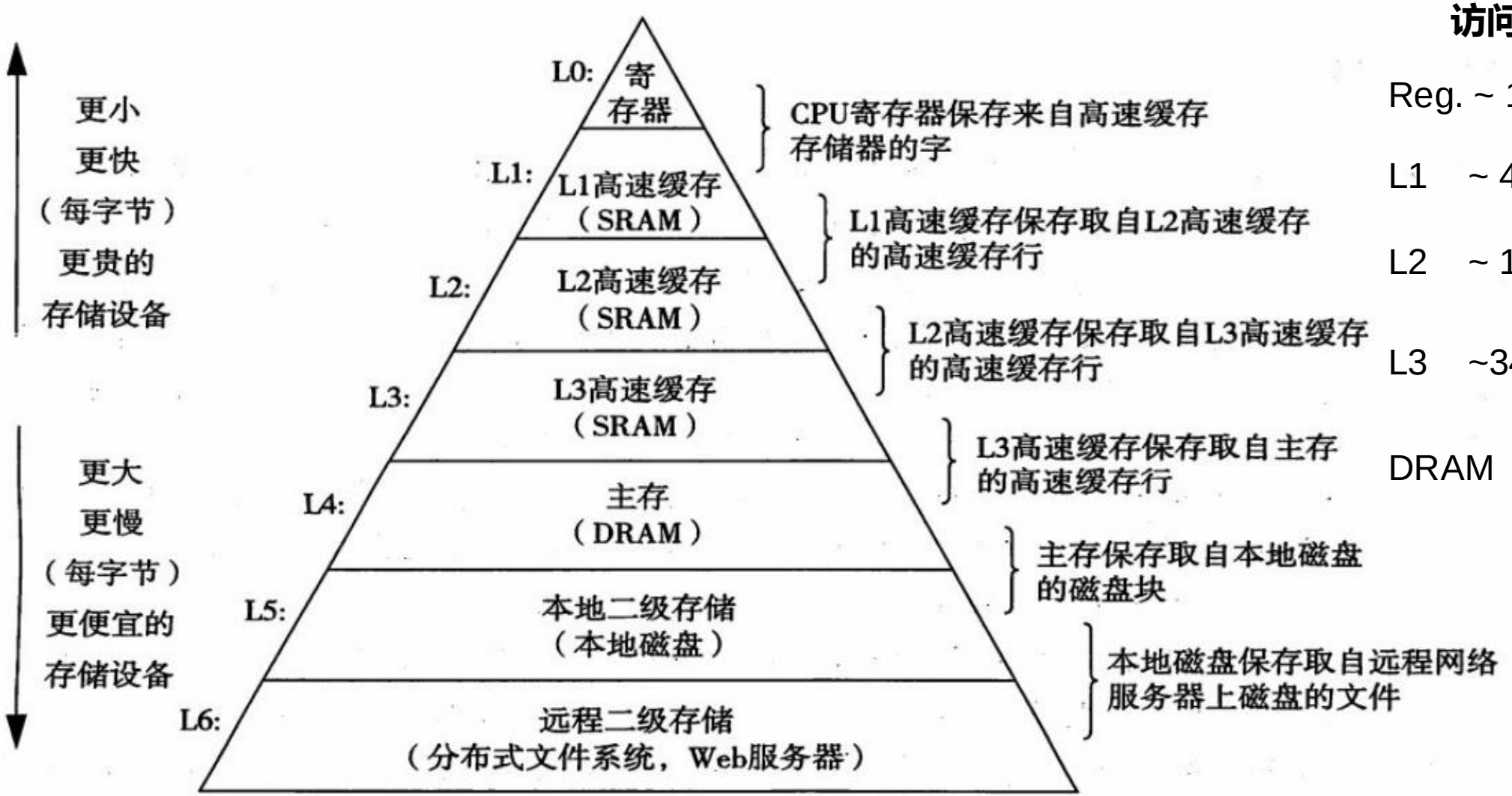
寄存器与内存的比较

项　　目	寄存器	内　　存
位　　置	CPU内部	CPU外部
访问速度	快	慢
容　　量	小	大
成　　本	高	低
表示方式	用名字表示	用地址表示
地　　址	没有	可用多种方式形成 (多种内存寻址方式)

回顾下“机器字在内存中的组织”



存储层次结构 (补充)



访问延迟

Reg. ~ 1 cycle

L1 ~ 4 cycles

L2 ~ 10 cycles

L3 ~34-40 cycles

Intel Nehalem architecture

DRAM ~100 cycles

汇编语言基本指令/寻址方式等（C语言的角度）

如何从C代码生成汇编代码

C代码

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

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

对应的X86-64汇编（AT&T格式）

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

生成汇编代码的命令行： gcc –Og –S sum.c #-fno-stack-protector

参数‘-S’生成文件sum.s（-c的话生成sum.o，即对象文件，内含机器指令）

汇编语言的基本特点 - 数据类型

整型数据

数据宽度为
1, 2, 4, or 8 bytes

- **表示数值**
- **或者内存地址**

浮点数据

数据宽度为4, 8, or
10 bytes

无复杂数据类型

结构或者数组之类的

- **就是在内存中连续分配的字节**

汇编语言的基本特点 - 操作类型

对寄存器数据或者内存数据进行算术/逻辑操作

内存与寄存器之间、或者寄存器/寄存器之间传递数据

- 将数据从内存中读入寄存器
- 将寄存器数据写入内存

程序执行顺序的转移（transfer control）

- 无条件跳转 / 过程调用及返回
- 条件跳转

汇编语言数据格式

后缀	类型	数据大小
B	BYTE	1byte (8bits)
W	WORD	2byte (16bits)
L	LONG	4byte (32bits)
Q	QUADWORD	8byte (64bits)

- **在X86中，**使用**“字（word）”**来表示16位整数类型，“双字”表示32位，“四字”表示64位

- **AT&T汇编语言指令所处理的数据类型**一般是**采用汇编指令的后缀**来进行区分的

第一条汇编指令实例

*dest = t;

C语言

在dest指定的内存地址里存储t

movq %rax, (%rbx)

汇编

将8字节宽度的值存储到到指定的内存地址中

操作数

t: 寄存器 %rax

dest: 寄存器 %rbx

*dest: 内存 M[%rbx]

0x40059e: 48 89 03

目标代码

三字节指令（指令自身的地址为0x40059e）

反汇编命令

```
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：objdump –d sum

另一种反汇编方式

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

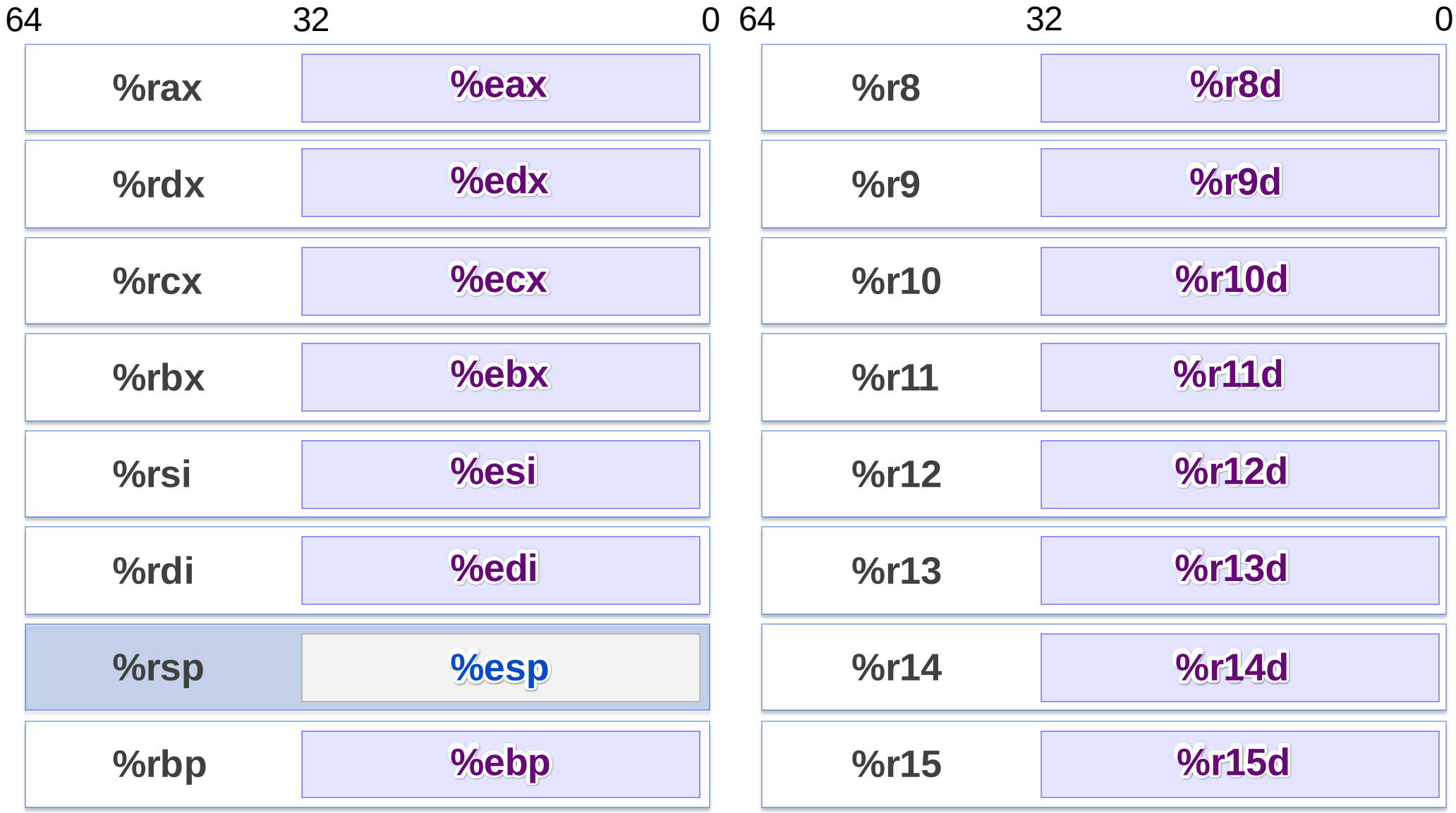
```
Dump of assembler code for function sumstore:
0x0000000000400595 <+0>: push   %rbx
0x0000000000400596 <+1>: mov    %rdx,%rbx
0x0000000000400599 <+4>: callq  0x400590 <plus>
0x000000000040059e <+9>: mov    %rax,(%rbx)
0x00000000004005a1 <+12>:pop    %rbx
0x00000000004005a2 <+13>:retq
```

gdb命令反汇编

- **gdb sum**
  disassemble sumstore
  反汇编某个过程/函数
- **x/14xb sumstore**　　**x /nfu <addr>**
  显示从“sumstore”开始的14个字节

安排有专门的课时来讲解gdb等的使用

x86-64的通用寄存器



x86-32的通用寄存器

数据传送指令（mov）

数据传输

- movq Source, Dest

操作数类型

立即数: 整型常数

- 示例: $0x400, $-533
- 类似于C代码里的常数，注意前缀‘$’
- 位宽可以是1, 2, 4, 8字节

寄存器: 16个通用寄存器之一

- 示例: %rax, %r13
- 不过%rsp具有特殊用途
- 其它某些寄存器在某些特定指令中有特定用途

内存: 8 个连续字节，起始地址由地址表达式指定

- 一种简单的内存地址示例: (%rax)
- 有多种不同的地址表达式

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

%rN
⋮

mov指令支持的不同操作数类型组合

	源操作数	目的操作数	源（左侧），目的（右侧）	类似的C语言表示
movl	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;

但是不能两个操作数都为内存数据

简单的寻址模式

间接寻址　　格式：(R)　　内存地址：Mem[Reg[R]]

- 寄存器R指定内存地址
- movq (%rcx),%rax

“基址+偏移量”寻址　格式：　D(R)　内存地址：Mem[Reg[R]+D]

- 寄存器R指定内存起始地址
- 常数D给出偏移量
- movq 8(%rbp),%rdx

变址模式

常见形式：D(Rb,Ri,S)　内存地址：Mem[Reg[Rb]+S*Reg[Ri]+D]

- D：常量（地址偏移量）
- S：比例因子 1, 2, 4, or 8
- Rb：基址寄存器，16个通用寄存器之
- Ri ：索引寄存器，%rsp不作为索引寄存器

其他变形

- (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]]

寻址模式实例

%rdx	0xf000
%rcx	0x0100

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

X86-64下的简单寻址示例

```
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
```

参数通过寄存器来传递

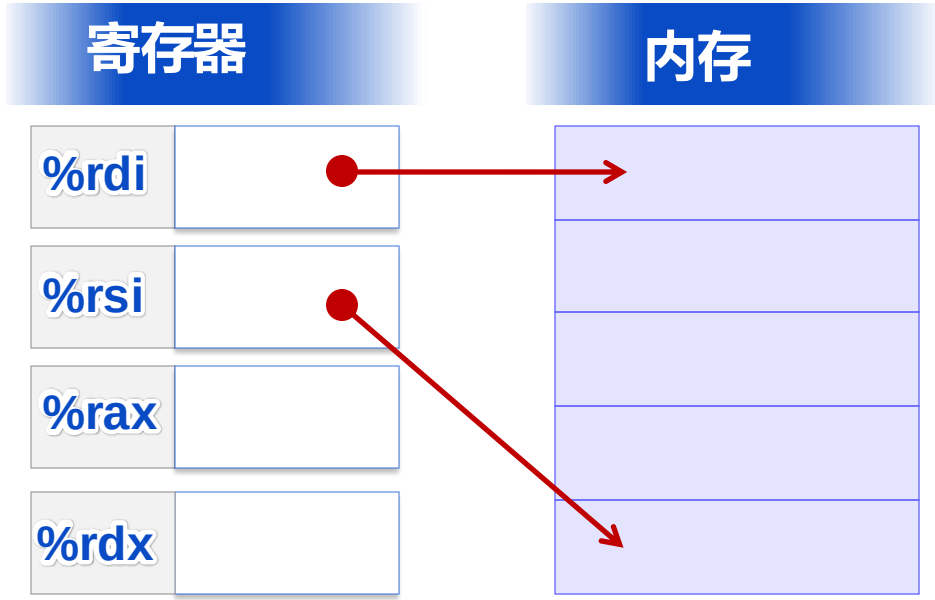
%rdi：第一个参数(xp)；%rsi: 第二个(yp)；参数类型为64位宽的指针

注意

当参数少于7个时，参数从左到右放入寄存器: rdi, rsi, rdx, rcx, r8, r9。当参数为7个及以上时，前6个传送方式不变，但后面的依次从 "右向左" 放入栈中

Swap函数执行过程分解-1

```
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
```

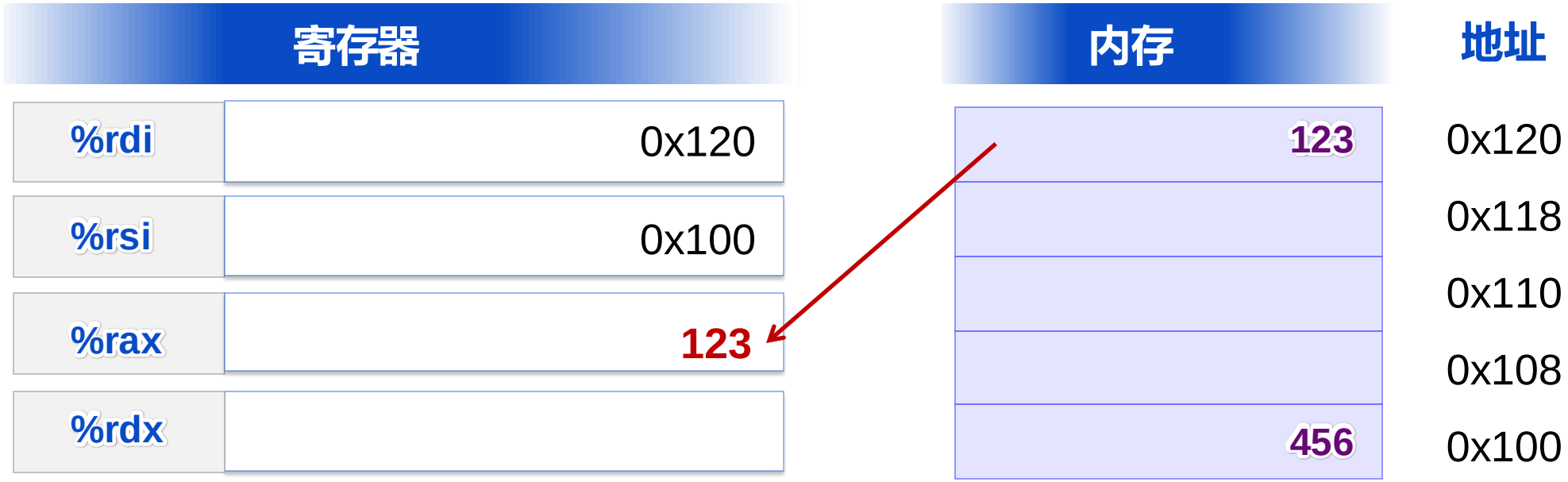
Swap函数执行过程分解-2

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

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

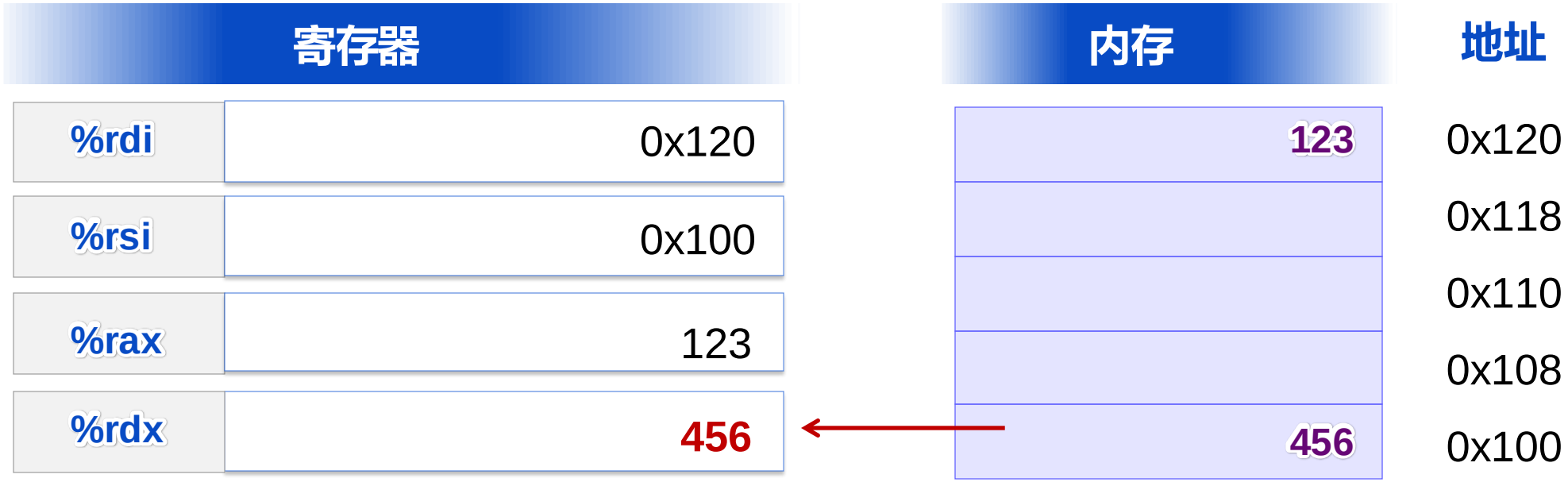
```
swap:
   movq    (%rdi), %rax  # t0 = *xp
   movq    (%rsi), %rdx  # t1 = *yp
   movq    %rdx, (%rdi)  # *xp = t1
   movq    %rax, (%rsi)  # *yp = t0
   ret
```

Swap函数执行过程分解-3



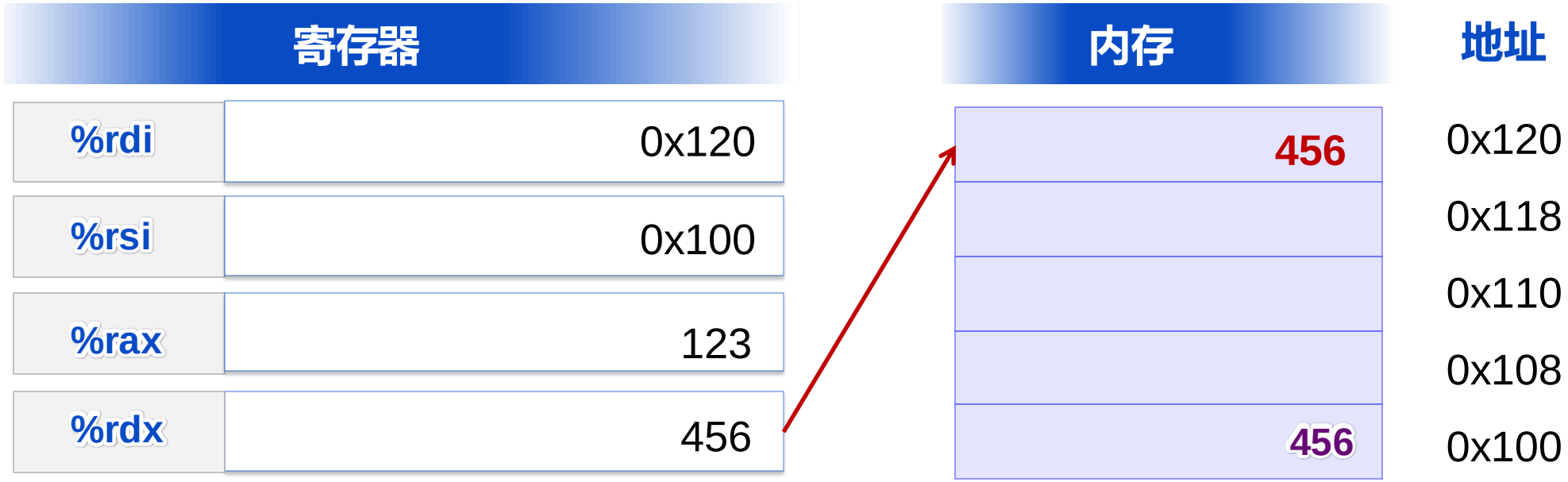

```
swap:
   movq    (%rdi), %rax  # t0 = *xp
   movq    (%rsi), %rdx  # t1 = *yp
   movq    %rdx, (%rdi)  # *xp = t1
   movq    %rax, (%rsi)  # *yp = t0
   ret
```

Swap函数执行过程分解-4



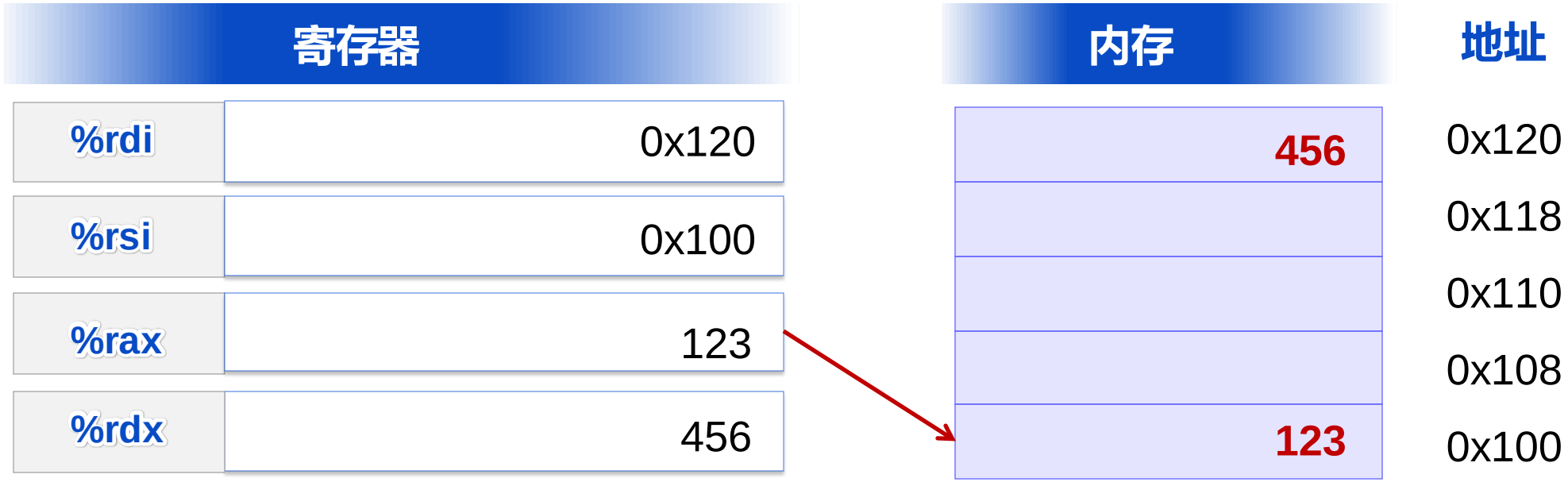

```
swap:
   movq    (%rdi), %rax  # t0 = *xp
   movq    (%rsi), %rdx  # t1 = *yp
   movq    %rdx, (%rdi)  # *xp = t1
   movq    %rax, (%rsi)  # *yp = t0
   ret
```

Swap函数执行过程分解-5




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

Swap函数执行过程分解-6




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

X86-64下另一个swap

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

```
swap:
        movl    (%rdi), %edx
        movl    (%rsi), %eax
        movl    %eax, (%rdi)
        movl    %edx, (%rsi)
        retq
```

参数通过寄存器来传递

%rdi：第一个参数(xp)；%rsi: 第二个(yp)；

参数类型为64位宽的指针

被操作的数据仍是32位

所以使用寄存器 %eax、%edx

以及movl 指令

地址计算指令

leaq Src,Dest

Src 是地址计算表达式，计算出来的地址赋给 Dest

使用实例

地址计算（无需访存）　　示例：p = &x[i];

进行x + k*y这一类型的整数计算，k = 1, 2, 4, or 8.

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

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

变址模式（回顾）

常见形式：D(Rb,Ri,S)　内存地址：Mem[Reg[Rb]+S*Reg[Ri]+D]

- D：常量（地址偏移量）
- S：比例因子 1, 2, 4, or 8
- Rb：基址寄存器，16个通用寄存器之
- Ri ：索引寄存器，%rsp不作为索引寄存器

其他变形

- (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]]

地址计算指令

leaq Src,Dest

Src 是地址计算表达式，计算出来的地址赋给 Dest

使用实例

地址计算（无需访存）　　示例：p = &x[i];

进行x + k*y这一类型的整数计算，k = 1, 2, 4, or 8.

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

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

整数计算指令

指令格式

双操作数指令	
addq	Src,Dest
subq	Src,Dest
imulq	Src,Dest
salq	Src,Dest
sarq	Src,Dest
shrq	Src,Dest
xorq	Src,Dest
andq	Src,Dest
orq	Src,Dest

指令执行的操作

Dest = Dest + Src	
Dest = Dest – Src	
Dest = Dest * Src	Multiply （取低64位）
Dest = Dest << Src	与`shll`等价
Dest = Dest >> Src	算术右移
Dest = Dest >> Src	逻辑右移
Dest = Dest ^ Src	
Dest = Dest & Src	
Dest = Dest | Src	

整数计算指令

格 式	具体操作
单操作数指令	
incq Dest	Dest = Dest + 1
decq Dest	Dest = Dest - 1
negq Dest	Dest = - Dest
notq Dest	Dest = ~ Dest

将leal指令用于计算

```
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
```

关键指令

- **leaq:** 地址计算
- ***salq***: 移位操作
- **imulq:** 乘法
  但只用了一次

将leal指令用于计算

```
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    # t4
  salq    $4, %rdx               # t4+x+4
  leaq    4(%rdi,%rdx), %rcx
  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

将leal指令用于计算

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

$2^{13} = 8192, 2^{13} – 7 = 8185$

```
logical:
    movl   %edi, %eax
    xorl   %esi, %eax
    sarl   $17, %eax
    andl   $8185, %eax
    ret
```

寄存器	用途
%edi	参数 x
%esi	参数 y
%eax	t1, t2, rval

目前为止要掌握的“定位”问题

```
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      # t4
  salq   $4, %rdx
  leaq   4(%rdi,%rdx), %rcx       # t4+x+4
  imulq  %rcx, %rax               # rval
  ret
```

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

```
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
```

参数通过寄存器来传递

%rdi：第一个参数(xp)；%rsi: 第二个(yp)；参数类型为64位宽的指针

注意

当参数少于7个时， 参数从左到右放入寄存器: rdi, rsi, rdx, rcx, r8, r9。当参数为7个及以上时， 前6个传送方式不变， 但后面的依次从 "右向左" 放入栈中

（部分）指令/函数/数据的“定位”

- **局部变量的定位***
- （条件/无条件）**跳转指令的目的寻址**
- **静态函数入口地址**
- **函数返回地址**
- **函数的参数、返回值** → **%eax / %rax**

*注意

这并不意味着局部变量只能如此“定位”。学过运行栈之后还会有补充

一些归纳

X86-64指令的特点

- **支持多种类型的指令操作数**
  立即数, 寄存器, 内存数据
- **算逻指令可以以内存数据为操作数**
- **支持多种内存地址计算模式**
  Rb + S*Ri + D
  也可用于整数计算(如leal / leaq指令)
- **变长指令**
  1 ~ 15字节长度