程序的机器级表示1

本笔记覆盖Computer System Third Edition第三章前面部分，所有插图来自于CMU 15213 课程讲义。

随记

- 由于刚开始计算机是16位字长的，因此32位的整型就是double word size，64位长整型就是quadratic word size。但是由于双精度浮点数也是quadratic word size的，因此对于这两种指令来说都是以l结尾，但是这并不会造成混淆，因为整型操作指令和浮点数操作指令是不相同的。
- 当把一个值从一个小寄存器move到一个大寄存器中时，存在0扩展和符号位扩展。这里说的小寄存器和大寄存器是指的寄存器的使用情况。
  - 0扩展：存在从单字到字长，双字，四字；字长到双字，四字；注意这里没有双字到四字，0扩展会自动在寄存器首16位添加0；
  - 符号位扩展：存在从单字到字长，双字，四字；字长到双字，四字；双字到四字；

大纲：

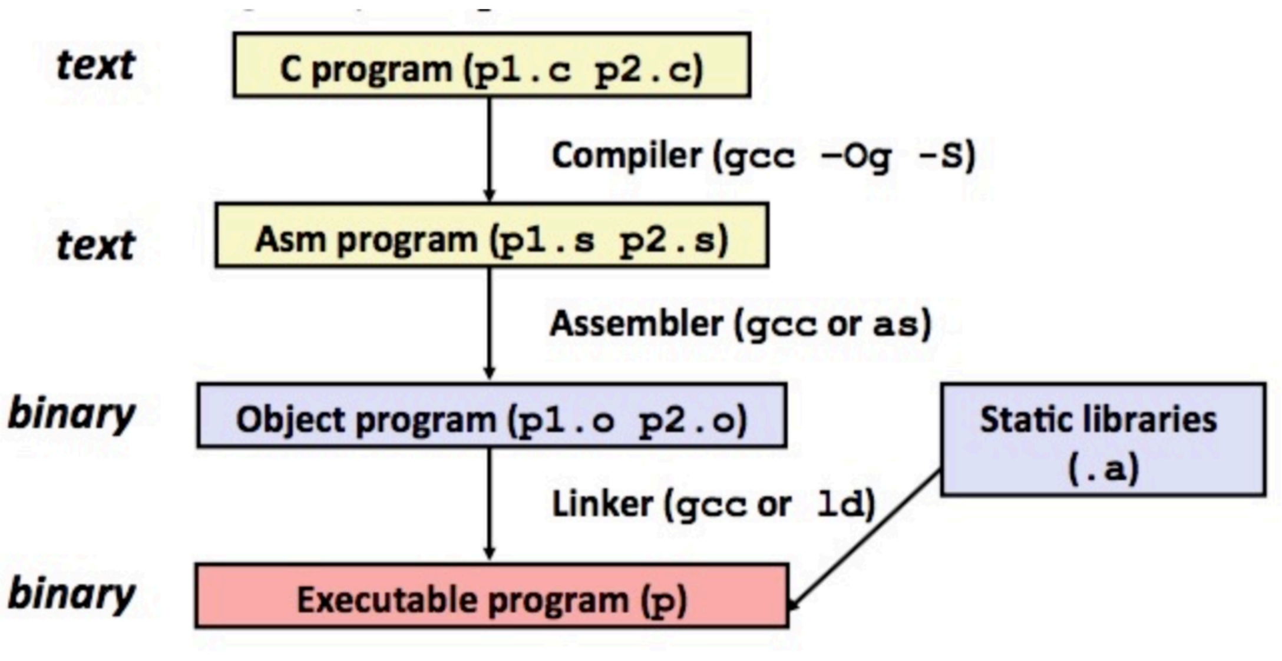
本章主要讲解c语言，汇编和机器代码以及在编程过程中常用操作的汇编指令，比如循环，条件等等。

程序编码：一个c文件从编译到执行总共会牵扯到四个步骤，四种类型文件：

- c文件 (文件名以.c结尾) (文本文件)
- 汇编文件 (文件名以.s结尾) (文本文件)
- 目标文件 (文件名以.o结尾) (二进制文件)
- 可执行文件 (二进制文件)

`gcc -Og -o p1.c p2.c`

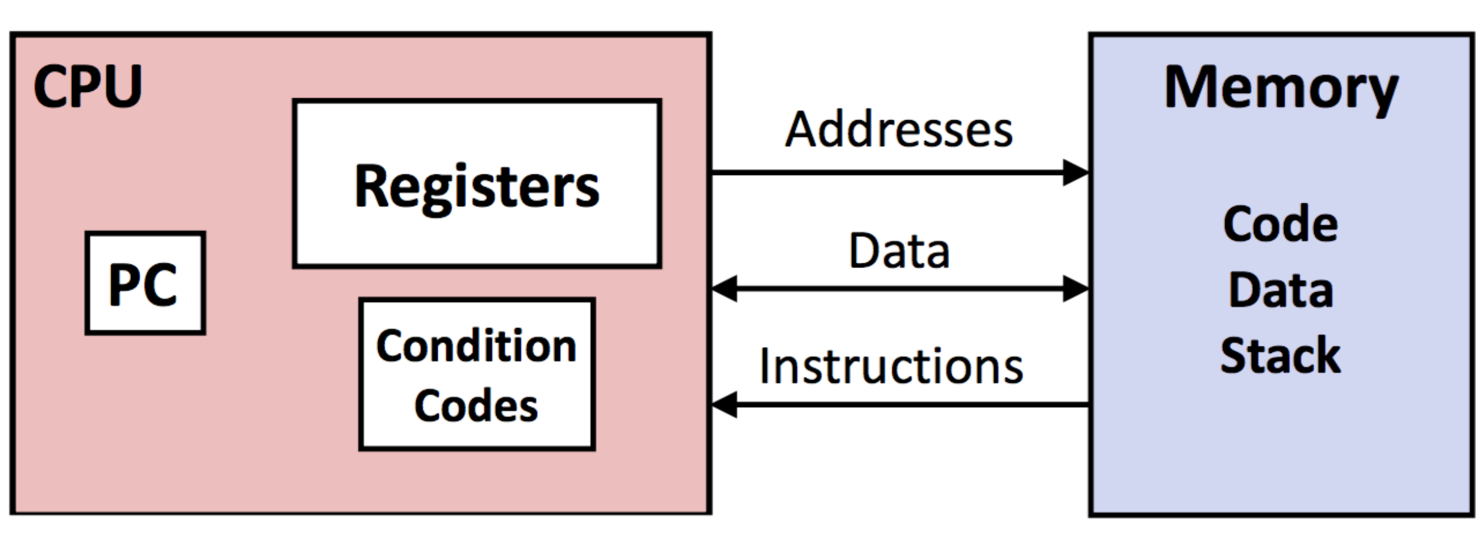
- 首先是*预处理器(preprocessor)*将源代码中的include的外库和代码中定义的宏变量(macros)引入，生成一个新的c代码文件；
- 然后*编译器(compiler)*开始发挥作用，将扩展后的c代码编译成汇编语言形式，这也是这张的主要讲解部分；
- 接着是*汇编器(Assembler)*将汇编文件转为二进制程序目标文件，此时文件格式已经发生变化；
- 最后是*连接器(linker)*将统一程序中的不同目标执行文件连接在一起，并且将全局变量地址以及一些静态库导入进来，最终形成可执行文件；



一些重要的定义：

> Architecture: (Instruction Set Architecture) The parts of a precessor design that one needs to understand or write assembly / machine code. 在我理解来就是指的是指令集
> Microarchitecture: Implementation of the architecture. 这个就是从更加微观的角度去看具体怎么实现指令之间的操作；

下图一个简化版的计算机结构：

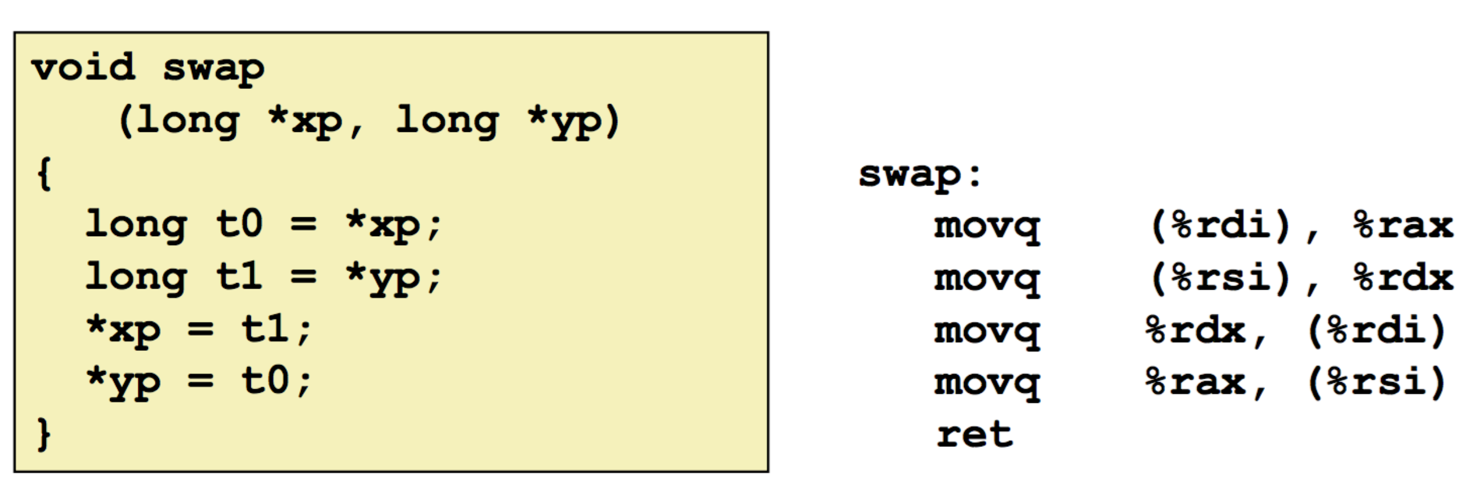


还有系统I/O总线，网络以及外部设备等等。

从上图可以看出几个重要的部分：

- PC: Program couner, 里面存放着下一条指令的地址；called 'RIP'；
- 整数寄存器：有16个命名的64位寄存器，存放的是整数；这些整数可以解释为即时操作数(immediate operand)以及地址(address);
- Condition codes: (条件码，存放着状态信息)，通常是用来保存最近的算术或逻辑操作信息，用来作为条件跳转；
- Memory: 内存，存放着数据，代码，地址以及系统栈；

看一个从c语言通过编译器产生的编译文件例子，在终端敲 `gcc -Og -S swap.c` ， `-Og` 代表进行不彻底优化；



```
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
```

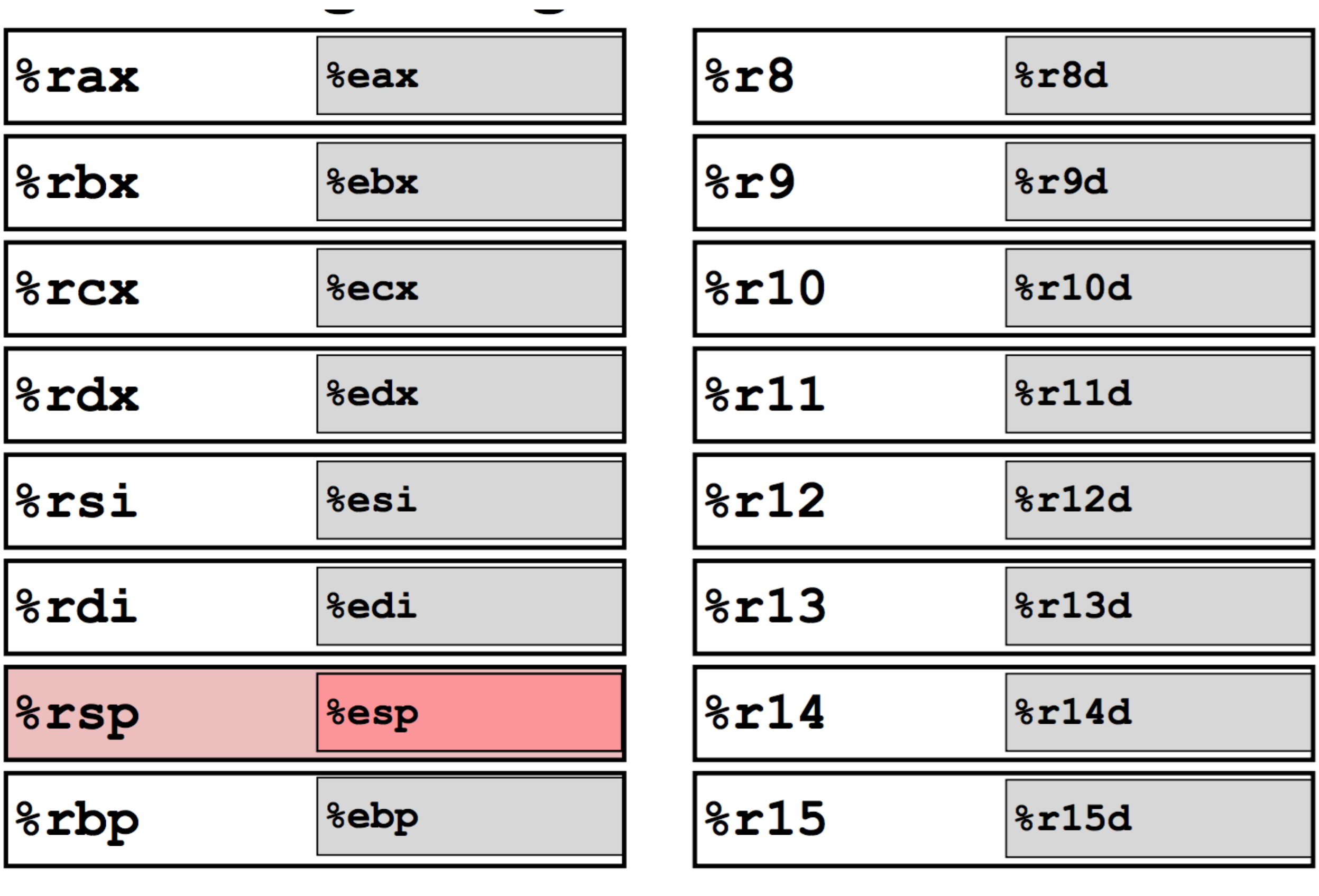
左边是c代码，右边是汇编文件。先来看看在汇编语言中数据类型：

- 整数：1，2，4，8 bytes，可以表示数据，也可以表示地址，当表示地址时，是无符号型的表达方式；
- 浮点数：4，8，10bytes，不用担心混淆，浮点数和整数存放的寄存器不一样；
- 代码：由一系列的指令操作构成；
- 注意，不存在数组和结构体这样的代码，本质上来说都是存放在内存中一连串的字节；

能够进行的操作也是非常简单：

- 从内存load数据到寄存器中；
- 从寄存器store数据到内存中；
- 进行一些数学操作，比如加减乘除；
- 还有就是一些条件跳转

下图就是常见的16个整数寄存器：



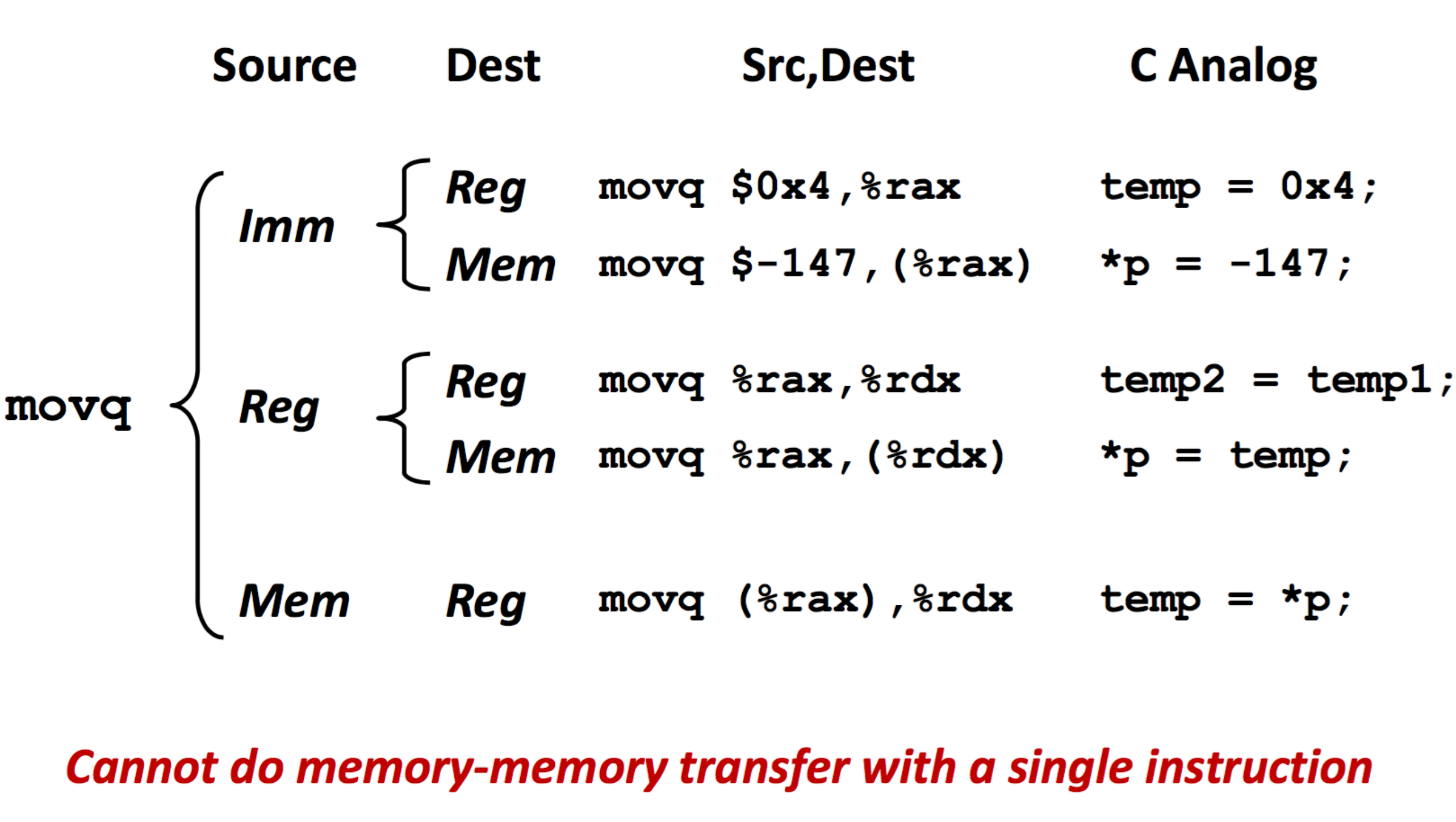
%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

这些寄存器的命名和使用已经和刚开始发明时已经有了很大的灵活性，但是存在一些约定俗成的使用还有一些固定使用；
比如说 `%rdi` 存放第一个操作数， `%rax` 存放返回值， `%rsp` 是存放栈指针等等。

现在来看看汇编中整数寄存器的一些基本操作：

- 移动数据
  整数寄存器进行移动数组操作时操作数的类型主要有三种：
  - 立即数(immediate data)
  - 寄存器(Register)
  - 内存(Memory)

下图给出了整数寄存器能够支持的操作：



	Source	Dest	Src,Dest	C Analog
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;`

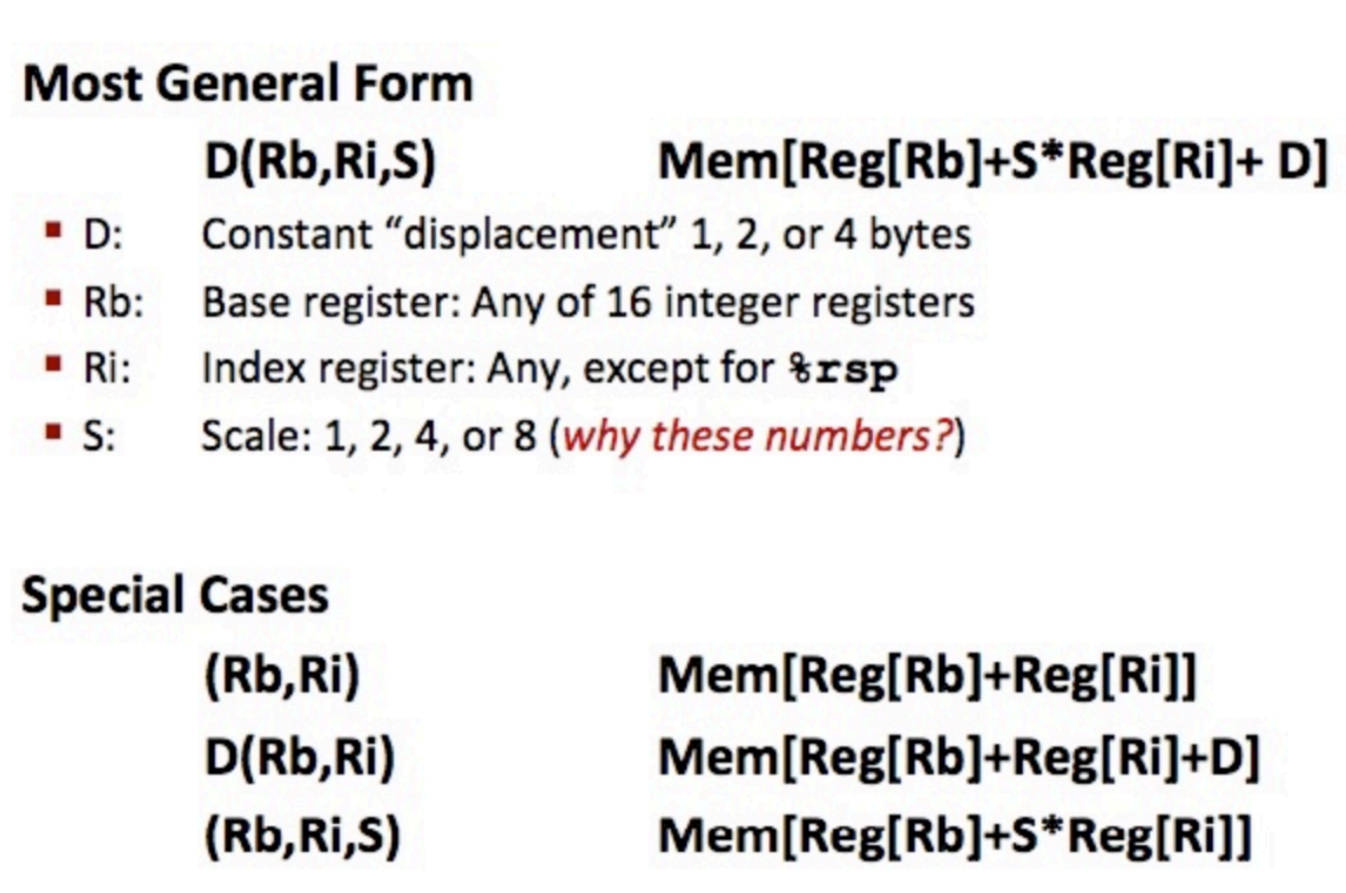
Cannot do memory-memory transfer with a single instruction

注意，不能从内存直接到内存，且操作数不能作为目的(这点是很明显的)

寻址计算：寻址计算有两种方式， 一是普通模式，也就是直接取址；二是偏移计算方式；

- 直接取址： `(R)` ，相当于 `Mem[Reg[R]]` ,也就是说寄存器R里面存放的是内存地址；
- 偏移计算取址： `D(R)` ,相当于 `Mem[Reg[R]+D]` ，寄存器存放的内存地址，然后加上偏移量；

偏移计算取址的一般模式是：



Most General Form

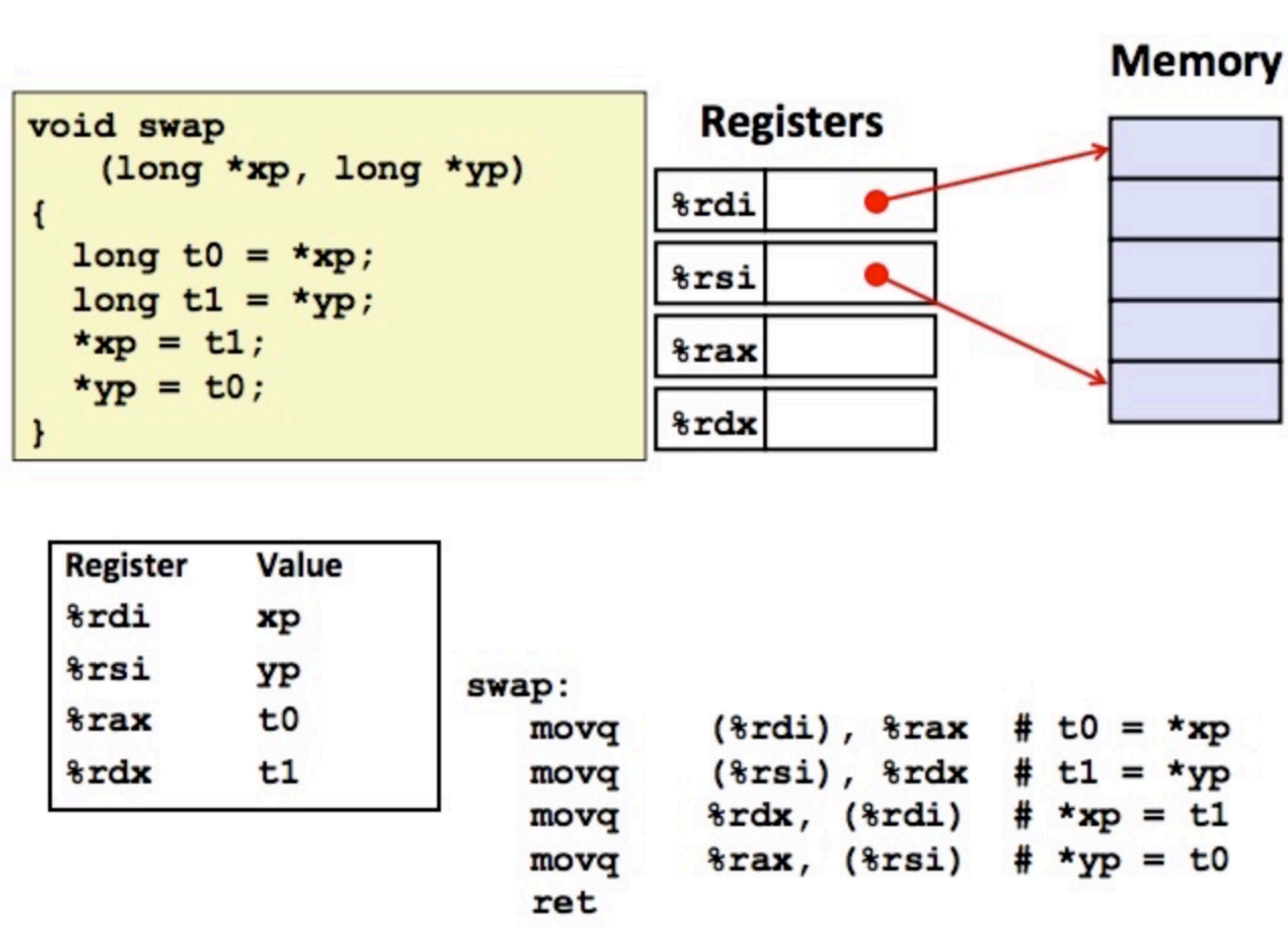
D(Rb,Ri,S) **Mem[Reg[Rb]+S*Reg[Ri]+ D]**

- 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?*)

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]]**

看个例子，一切都很明了：

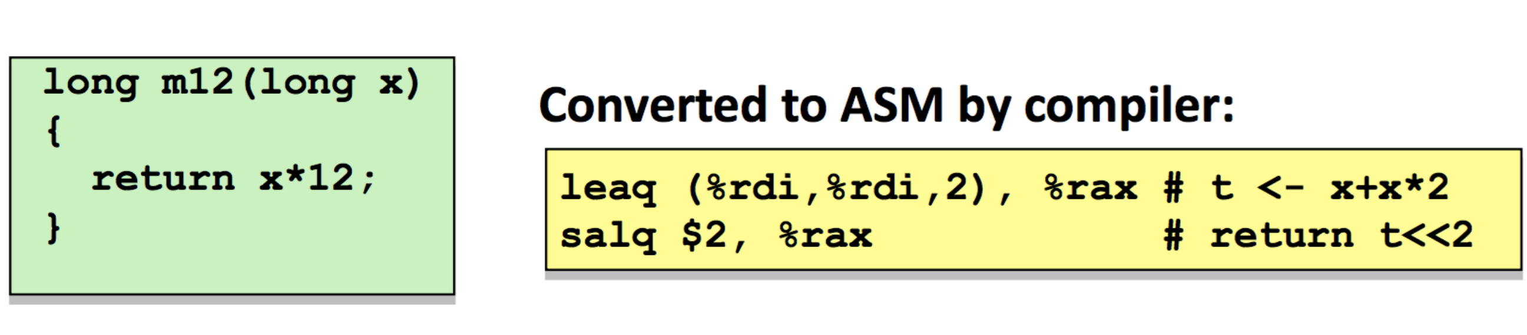


```
void swap
   (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

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
```

- 算术运算以及逻辑运算
  算术运算中一条重要的指令就是 `leaq Src, Dst` ，其中 `Src` 是地址表达式，然后把计算出来的值存入 `Dst` 指定寄存器。 要把这条指令和 `movq` 指令区分开来。
  `leaq` 和 `movq` 的区别在于： `leaq` 指令通过寻址计算方式计算地址，然后把地址存放在目标寄存器里面； `movq` 也是通过寻址计算方式计算地址，然后通过这个计算值去内存中找出地址中存放的值放到目标寄存器或者目标地址中。



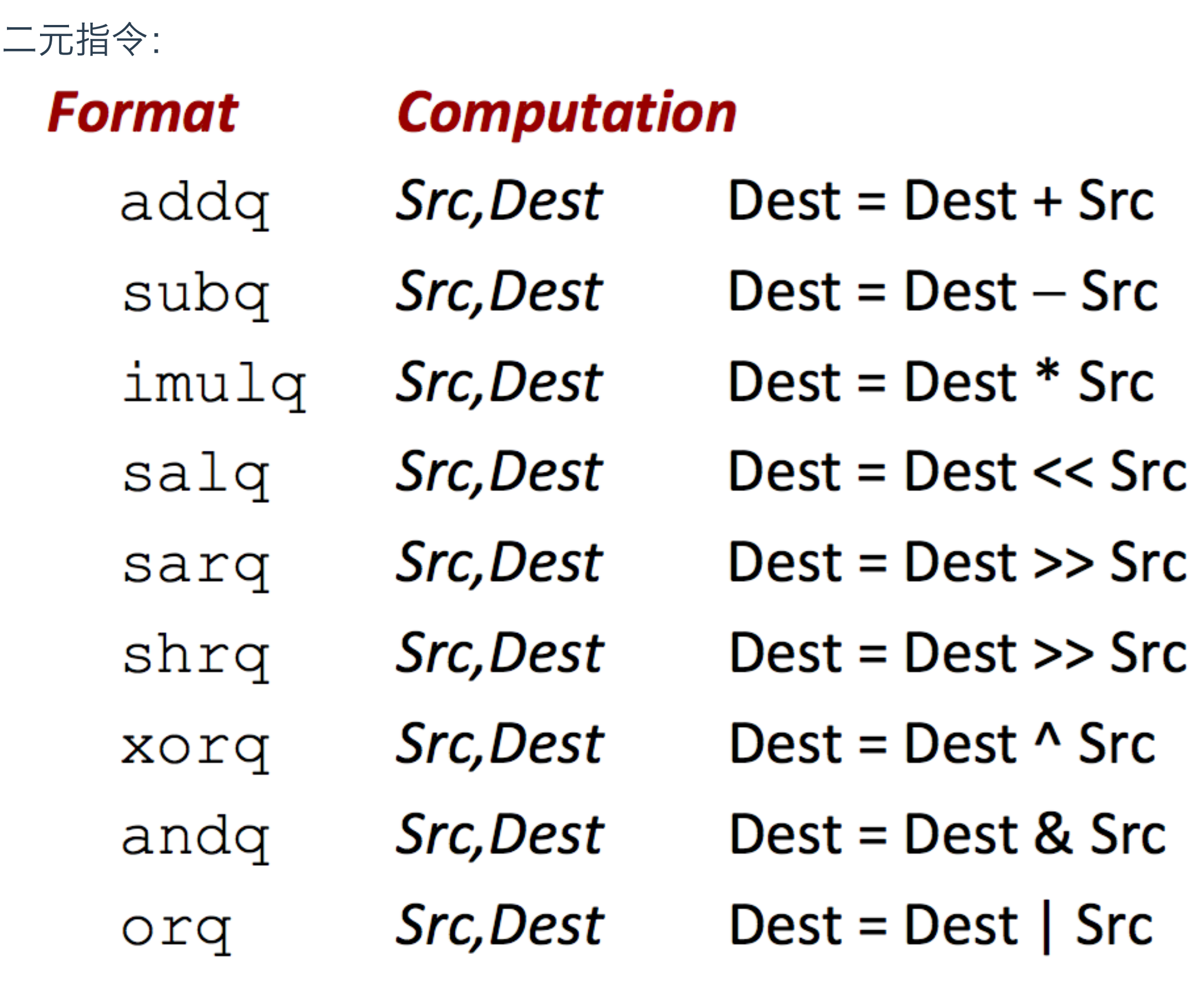
```
long m12(long x)
{
  return x*12;
}
```

Converted to ASM by compiler:

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

如上图中，寄存器 `%rdi` 中存放的 `x` 值本应该解释成地址，通过寻址计算方式之后得到 `3x` ，在这之后操作是直接把 `3x` 存放在目标寄存器中，而并不是通过计算出来的值去内存空间中索引真正的操作数。

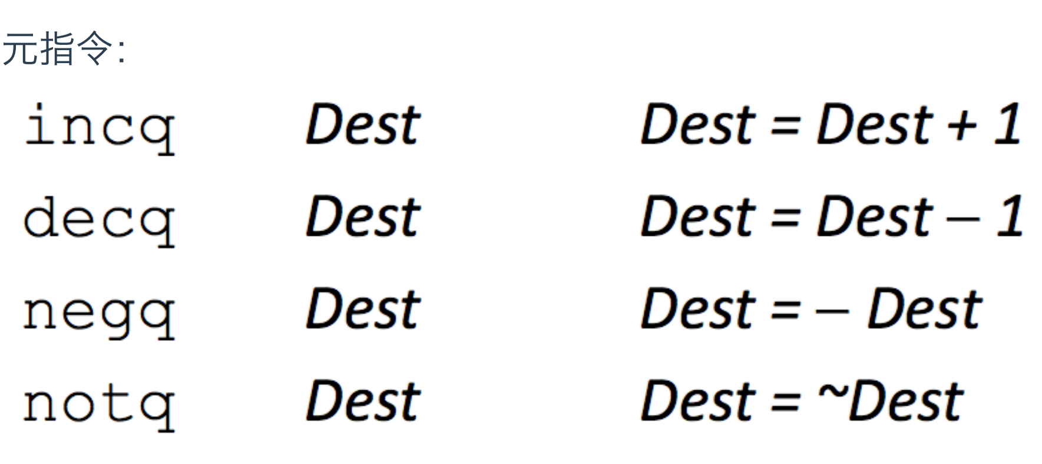
二元操作符和一元操作符：
正如数字计算中有二元计算符和一元计算符，汇编指令也有二元和一元操作指令。
二元指令：



Format	*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
sarq	*Src,Dest*	Dest = Dest >> Src
shrq	*Src,Dest*	Dest = Dest >> Src
xorq	*Src,Dest*	Dest = Dest ^ Src
andq	*Src,Dest*	Dest = Dest & Src
orq	*Src,Dest*	Dest = Dest | Src

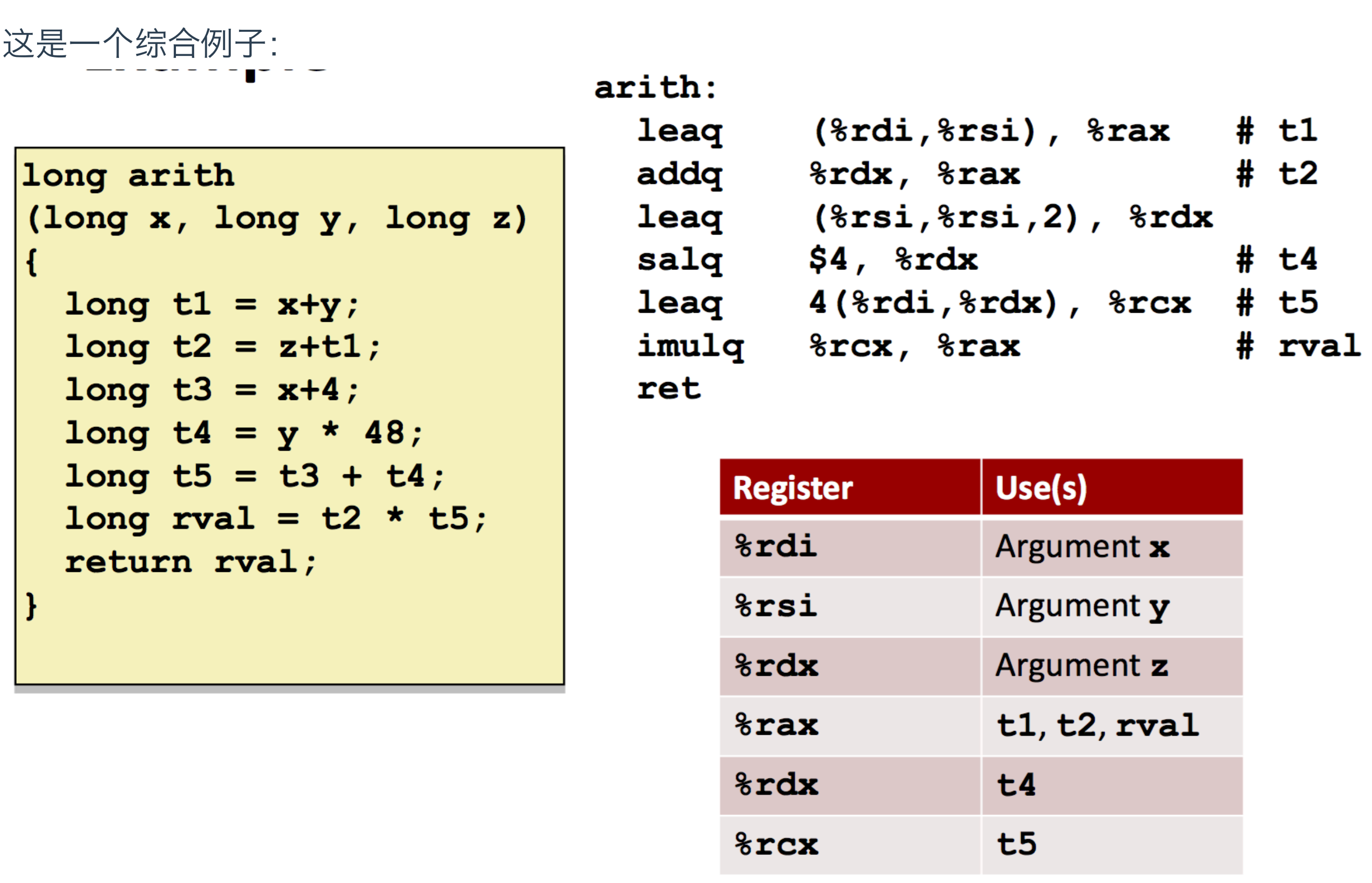
注意目标寄存器和源寄存器计算的顺序。

一元指令：



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

这是一个综合例子：



```
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
```

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