**反汇编程序的工作原理：**

反汇编程序（disassembler）会根据最终生成的二进制执行文件（如可执行文件或动态链接库）来反向解析并生成对应的汇编代码。它的工作原理是将机器码（二进制指令）转换回人类可读的汇编语言。由于编译器在编译高层语言时进行了大量优化和转换，反汇编得到的汇编代码通常不会完全匹配源代码的结构，但它仍然能够体现程序的执行逻辑。

不过，反汇编得到的汇编代码通常不包括高级语言中的变量名、函数名或注释，这些信息在编译过程中可能已经被剥离或替换为了内存地址或其他标识符。因此，反汇编出的代码相对难以理解和分析。

反汇编程序通常用于以下场景：

-\*\*调试程序\*\*：帮助开发者理解程序的底层行为，特别是在编译器优化后。

-\*\*逆向工程\*\*：分析已编译的程序以理解其逻辑、功能，或发现漏洞。

-\*\*安全审计\*\*：检查是否存在安全隐患，如二进制中的恶意代码。

**对于寄存器的名，基本不需要下太多注意**

在现在的系统中，基本对于寄存器的功能都进行了统一，原本单一功能的寄存器现在能实现更多的功能，但还是有几个寄存器需要注意，就比如**%rsp**寄存器，这个寄存器是用来管理函数的，储存的是栈帧

**对于这个rsp的额外分析（一时兴起查了下）**

**对于所有的寄存器，每时每刻都只能储存一个元素，rsp也不例外**

**这就意味着要想实现对于函数的管理，仅有这个寄存器是不够的， 这时就需要一段额外的内存来辅助了，这段内存可以看做栈，里 面储存的是最近的函数调用，越近调用的函数越在上层，而这个 寄存器储存的就是这个最上层的栈的地址**

**底层mov指令能实现的功能：**

将一个立即数立即加载到寄存器，注意，这里的立即数是指的常数，处于硬件设计等方面，底层不允许从内存中复制一个地址空间到另一个地址空间

将一个立即数加载到寄存器：

MOV %rax, 5 ; 将数字 5 移动到 rax 寄存器

将一个寄存器中的值复制到另一个寄存器：

MOV %rbx, %rcx ; 将 rbx 的值复制到 rcx

从内存中读取数据到寄存器中：

MOV %rax, 0x1000 ; 将内存地址 0x1000 的数据移动到 rax

将寄存器中的值写入内存：

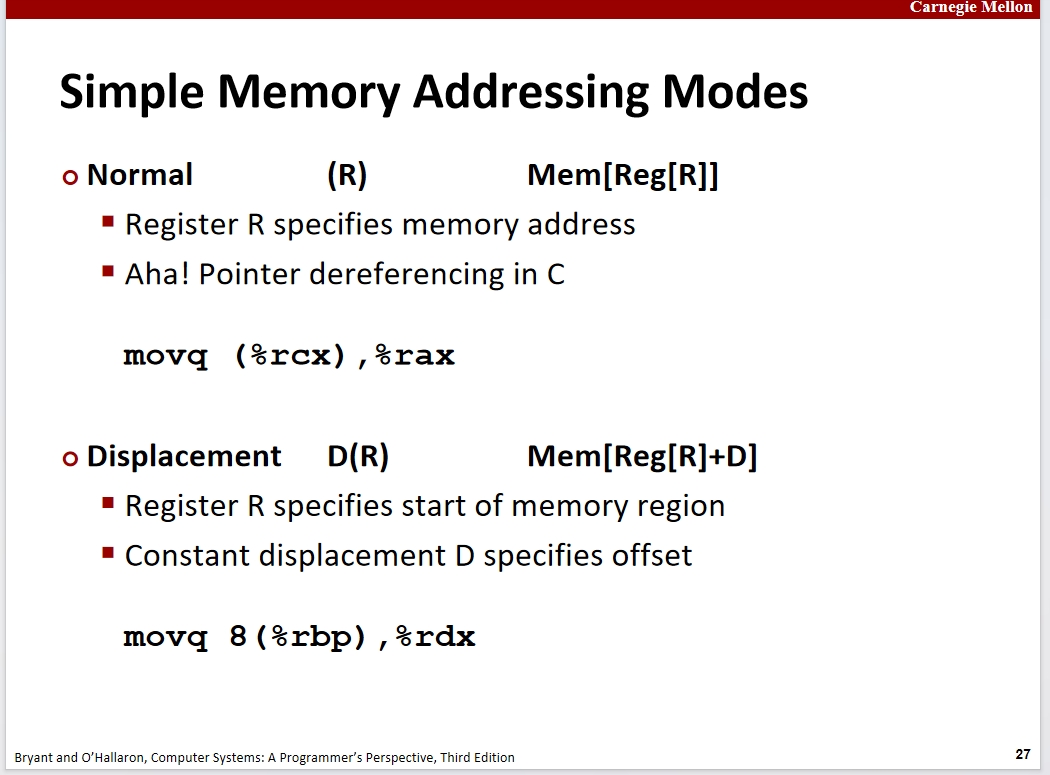
MOV 0x2000, %rbx ; 将 rbx 的值写入到内存地址 0x2000

还需要对括号加不加进行一些解释：

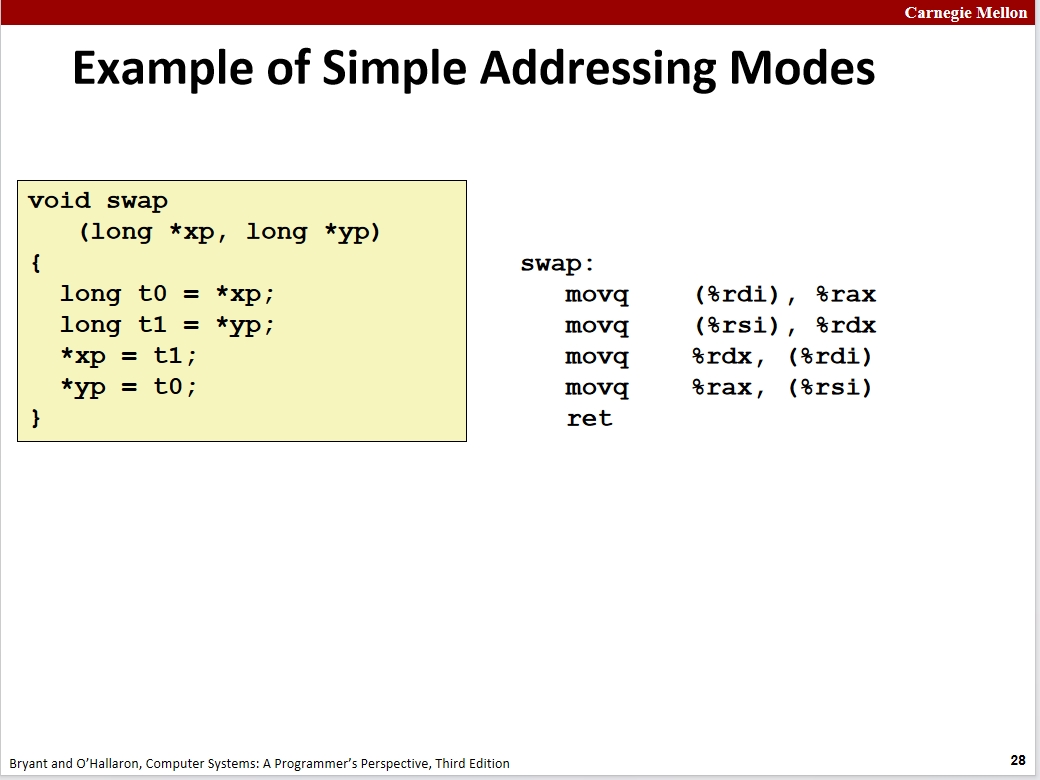
在这种汇编语句中，对寄存器的名加括号代表的是去找到寄存器 所指向的内存空间中的01串，而不加括号指的是该寄存器中01 串所代表的内存空间中的01串

MOV 8 (%rbx),%rax

需要注意，这里的括号是不能改变位置的，这里可以说是一个语 法。这条指令的意思是从 %rbx 指向的地址偏移 8 字节的位置读 取数据， 然后将其复制到 %rax 寄存器中。



接下来来分析一个例子



在这个例子中，我们来分析这里的内存逻辑

首先我们其实可以了解到一点，在内存中对一个数的解引用，其实可以看做是通过寄存器(储存该地址)去找到内存空间中对应的地址中储存的01串，并将这个储存到另一个寄存器中的过程，这里的前俩条C语句就对应了前俩条汇编语句，正好对应我们上面讲的。这里的%rdi储存的是xp储存的地址，%rsi储存的是yp的地址，而这里直接将这俩个操作后储存对应值的寄存器当做了我们储存这个值的位置，也就是说，%rax储存t0,%rdx储存t1

后面俩条语句与上面的逻辑相同，不进行赘述

在这里可以看出编译器对于临时变量的处理