



(程序计数器)相对寻址方式。处理器使用 PC 相对寻址方式,分支指令的编码会更简洁,同时这样也能允许代码从内存的一部分复制到另一部分而不需要更新所有的分支目标地址。因为我们更关心描述的简单性,所以就使用了绝对寻址方式。同 IA32 一样,所有整数采用小端法编码。当指令按照反汇编格式书写时,这些字节就以相反的顺序出现。

例如,用十六进制来表示指令 `rmmovq %rsp, 0x123456789abcd(%rdx)` 的字节编码。从图 4-2 我们可以看到, `rmmovq` 的第一个字节为 40。源寄存器 `%rsp` 应该编码放在 rA 字段中,而基址寄存器 `%rdx` 应该编码放在 rB 字段中。根据图 4-4 中的寄存器编号,我们得到寄存器指示符字节 42。最后,偏移量编码放在 8 字节的常数字中。首先在 `0x123456789abcd` 的前面填充上 0 变成 8 个字节,变成字节序列 `00 01 23 45 67 89 ab cd`。写成按字节反序就是 `cd ab 89 67 45 23 01 00`。将它们都连接起来就得到指令的编码 `4042cdab896745230100`。

指令集的一个重要性质就是字节编码必须有唯一的解释。任意一个字节序列要么是一个唯一的指令序列的编码,要么就不是一个合法的字节序列。Y86-64 就具有这个性质,因为每条指令的第一个字节有唯一的代码和功能组合,给定这个字节,我们就可以决定所有其他附加字节的长度和含义。这个性质保证了处理器可以无二义性地执行目标代码程序。即使代码嵌入在程序的其他字节中,只要从序列的第一个字节开始处理,我们仍然可以很容易地确定指令序列。反过来说,如果不知道一段代码序列的起始位置,我们就不能准确地确定怎样将序列划分成单独的指令。对于试图直接从目标代码字节序列中抽取出机器级程序的反汇编程序和其他一些工具来说,这就带来了问题。

 **练习题 4.1** 确定下面的 Y86-64 指令序列的字节编码。“`.pos 0x100`”那一行表明这段目标代码的起始地址应该是 `0x100`。

```
.pos 0x100 # Start code at address 0x100
irmovq $15,%rbx
rrmovq %rbx,%rcx
loop:
    rmmovq %rcx,-3(%rbx)
    addq   %rbx,%rcx
    jmp   loop
```

 **练习题 4.2** 确定下列每个字节序列所编码的 Y86-64 指令序列。如果序列中有不合法的字节,指出指令序列中不合法值出现的位置。每个序列都先给出了起始地址,冒号,然后是字节序列。

- A. `0x100: 30f3fcffffffffffff40630008000000000000`
- B. `0x200: a06f800c020000000000000030f30a000000000000090`
- C. `0x300: 50540700000000000000010f0b01f`
- D. `0x400: 611373000400000000000000`
- E. `0x500: 6362a0f0`

旁注 比较 x86-64 和 Y86-64 的指令编码

同 x86-64 中的指令编码相比, Y86-64 的编码简单得多,但是没那么紧凑。在所有的 Y86-64 指令中,寄存器字段的位置都是固定的,而在不同的 x86-64 指令中,它们的位置是不一样的。x86-64 可以将常数值编码成 1、2、4 或 8 个字节,而 Y86-64 总是将常数值编码成 8 个字节。