阶段	OPq rA,rB	rrmovq rA,rB	irmovqV, rB
取指	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_8[PC+2] valP ← PC+10
译码	valA ← R[rA] valB ← R[rB]	valA ← R[rA]	
执行	valE ← valB OP valA Set CC	valE ← 0+valA	valE ← 0+valC
访存			
写回	R[rB]← valE	R[rB]← valE	R[rB]← valE
更新 PC	PC ← valP	PC ← valP	PC ← valP

图 4-18　Y86-64 指令 OPq、rrmovq 和 irmovq 在顺序实现中的计算。这些指令计算了一个值，并将结果存放在寄存器中。符号 icode:ifun 表明指令字节的两个组成部分，而 rA:rB 表明寄存器指示符字节的两个组成部分。符号 M_1[x]表示访问(读或者写)内存位置 x 处的一个字节，而 M_8[x]表示访问八个字节

整数操作指令的处理遵循上面列出的通用模式。在取指阶段，我们不需要常数字，所以 valP 就计算为 PC+2。在译码阶段，我们要读两个操作数。在执行阶段，它们和功能指示符 ifun 一起再提供给 ALU，这样一来 valE 就成为了指令结果。这个计算是用表达式 valB OP valA 来表达的，这里 OP 代表 ifun 指定的操作。要注意两个参数的顺序——这个顺序与 Y86-64(和 x86-64)的习惯是一致的。例如，指令 subq %rax, %rdx 计算的是 R[%rdx]-R[%rax]的值。这些指令在访存阶段什么也不做，而在写回阶段，valE 被写入寄存器 rB，然后 PC 设为 valP，整个指令的执行就结束了。

旁注　跟踪 subq 指令的执行

作为一个例子，让我们来看看一条 subq 指令的处理过程，这条指令是图 4-17 所示目标代码的第 3 行中的 subq 指令。可以看到前面两条指令分别将寄存器%rdx 和%rbx 初始化成 9 和 21。我们还能看到指令位于地址 0x014，由两个字节组成，值分别为 0x61 和 0x23。这条指令处理的各个阶段如下表所示，左边列出了处理一个 OPq 指令的通用的规则(图 4-18)，而右边列出的是对这条具体指令的计算。

阶段	OPq rA,rB	subq %rdx, %rbx
取指	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	icode:ifun ← M_1[0x014]=6:1 rA:rB ← M_1[0x015]=2:3 valP ← 0x014+2=0x016
译码	valA ← R[rA] valB ← R[rB]	valA ← R[%rdx]=9 valB ← R[%rbx]=21
执行	valE ← valB OP valA Set CC	valE ← 21−9=12 ZF ← 0, SF ← 0, OF ← 0
访存		
写回	R[rB]← valE	R[%rbx]← valE=12
更新 PC	PC ← valP	PC ← valP=0x016