


这个跟踪表明我们达到了理想的效果，寄存器%rbx设成了12，三个条件码都设成了0，而PC加了2。

执行 `rrmovq` 指令和执行算术运算类似。不过，不需要取第二个寄存器操作数。我们将ALU的第二个输入设为0，先把它和第一个操作数相加，得到 $valE = valA$ ，然后再把这个值写到寄存器文件。对 `irmovq` 的处理与此类似，除了ALU的第一个输入为常数值 $valC$ 。另外，因为是长指令格式，对于 `irmovq`，程序计数器必须加10。所有这些指令都不改变条件码。

 **练习题 4.13** 填写下表的右边一栏，这个表描述的是图4-17中目标代码第4行上的 `irmovq` 指令的处理情况：

阶段	通用	具体
	<code>irmovq V, rB</code>	<code>irmovq \$128, %rsp</code>
取指	$icode, ifun \leftarrow M_1[PC]$ $rA, rB \leftarrow M_1[PC+1]$ $valC \leftarrow M_8[PC+2]$ $valP \leftarrow PC+10$	
译码		
执行	$valE \leftarrow 0 + valC$	
访存		
写回	$R[rB] \leftarrow valE$	
更新 PC	$PC \leftarrow valP$	

这条指令的执行会怎样改变寄存器和PC呢？

图4-19给出了内存读写指令 `rmmovq` 和 `mrmovq` 所需要的处理。基本流程也和前面的一样，不过是用ALU来加 $valC$ 和 $valB$ ，得到内存操作的有效地址（偏移量与基址寄存器值之和）。在访存阶段，会将寄存器值 $valA$ 写到内存，或者从内存中读出 $valM$ 。

阶段	<code>rmmovq rA, D(rB)</code>	<code>mrmovq D(rB), rA</code>
取指	$icode, ifun \leftarrow M_1[PC]$ $rA, rB \leftarrow M_1[PC+1]$ $valC \leftarrow M_8[PC+2]$ $valP \leftarrow PC+10$	$icode, ifun \leftarrow M_1[PC]$ $rA, rB \leftarrow M_1[PC+1]$ $valC \leftarrow M_8[PC+2]$ $valP \leftarrow PC+10$
译码	$valA \leftarrow R[rA]$ $valB \leftarrow R[rB]$	$valB \leftarrow R[rB]$
执行	$valE \leftarrow valB + valC$	$valE \leftarrow valB + valC$
访存	$M_8[valE] \leftarrow valA$	$valE \leftarrow M_8[valE]$
写回		$R[rA] \leftarrow valM$
更新 PC	$PC \leftarrow valP$	$PC \leftarrow valP$

图4-19 Y86-64指令 `rmmovq` 和 `mrmovq` 在顺序实现中的计算。这些指令读或者写内存

旁注 跟踪 `rmmovq` 指令的执行

让我们来看看图4-17中目标代码的第5行 `rmmovq` 指令的处理情况。可以看到，前面的指令已将寄存器%rsp初始化成了128，而%rbx仍然是 `subq` 指令（第3行）算出来的