


阶段	通用	具体
	pushq rA	pushq %rdx
取指	icode:ifun $\leftarrow M_1[PC]$ rA:rB $\leftarrow M_1[PC+1]$	icode:ifun $\leftarrow M_1[0x02a]=a:0$ rA:rB $\leftarrow M_1[0x02b]=2:f$
	valP $\leftarrow PC+2$	valP $\leftarrow 0x02a+2=0x02c$
译码	valA $\leftarrow R[rA]$ valB $\leftarrow R[\%rsp]$	valA $\leftarrow R[\%rdx]=9$ valB $\leftarrow R[\%rsp]=128$
执行	valE $\leftarrow valB+(-8)$	valE $\leftarrow 128+(-8)=120$
访存	M <sub>8</sub> [valE] $\leftarrow valA$	M <sub>8</sub> [120] $\leftarrow 9$
写回	R[%rsp] $\leftarrow valE$	R[%rsp] $\leftarrow 120$
更新 PC	PC $\leftarrow valP$	PC $\leftarrow 0x02c$


跟踪记录表明这条指令的效果就是将%rsp 设为 120, 将 9 写入地址 120, 并将 PC 加 2。


popq 指令的执行与 pushq 的执行类似, 除了在译码阶段要读两次栈指针以外。这样做看上去很多余, 但是我们会看到让 valA 和 valB 都存放栈指针的值, 会使后面的流程跟其他的指令更相似, 增强设计的整体一致性。在执行阶段, 用 ALU 给栈指针加 8, 但是用没加过 8 的原始值作为内存操作的地址。在写回阶段, 要用加过 8 的栈指针更新栈指针寄存器, 还要将寄存器 rA 更新为从内存中读出的值。用没加过 8 的值作为内存读地址, 保持了 Y86-64(和 x86-64)的惯例, popq 应该首先读内存, 然后再增加栈指针。

 **练习题 4.14** 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 7 行 popq 指令的处理情况:

阶段	通用	具体
	popq rA	popq %rax
取指	icode:ifun $\leftarrow M_1[PC]$ rA:rB $\leftarrow M_1[PC+1]$	
	valP $\leftarrow PC+2$	
译码	valA $\leftarrow R[\%rsp]$ valB $\leftarrow R[\%rsp]$	
执行	valE $\leftarrow valB+8$	
访存	valM $\leftarrow M_8[valA]$	
写回	R[%rsp] $\leftarrow valE$ R[rA] $\leftarrow valM$	
更新 PC	PC $\leftarrow valP$	

这条指令的执行会怎样改变寄存器和 PC 呢?

 **练习题 4.15** 根据图 4-20 中列出的步骤, 指令 pushq %rsp 会有什么样的效果? 这与练习题 4.7 中确定的 Y86-64 期望的行为一致吗?

 **练习题 4.16** 假设 popq 在写回阶段中的两个寄存器写操作按照图 4-20 列出的顺序进行。popq %rsp 执行的效果会是怎样的? 这与练习题 4.8 中确定的 Y86-64 期望的行为一致吗?