结果 12。我们还可以看到，指令位于地址 0x020，有 10 个字节。前两个的值为 0x40 和 0x43，后 8 个是数字 0x0000000000000064(十进制数 100)按字节反过来得到的数。各个阶段的处理如下：

阶段	通用	具体
	rmmovq rA, D(rB)	rmmovq %rsp, 100(%rbx)
取指	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← M_8[PC+2] valP ← PC+10	icode:ifun ← M_1[0x020]=4:0 rA:rB ← M_1[0x021]=4:3 valC ← M_8[0x022]=100 valP ← 0x020+10=0x02a
译码	valA ← R[rA] valB ← R[rB]	valA ← R[%rsp]=128 valB ← R[%rbx]=12
执行	valE ← valB+valC	valE ← 12+100=112
访存	M_8[valE]← valA	M_8[112]← 128
写回		
更新 PC	PC ← valP	PC ← 0x02a

跟踪记录表明这条指令的效果就是将 128 写入内存地址 112，并将 PC 加 10。

图 4-20 给出了处理 pushq 和 popq 指令所需的步骤。它们可以算是最难实现的 Y86-64 指令了，因为它们既涉及访问内存，又要增加或减少栈指针。虽然这两条指令的流程比较相似，但是它们还是有很重要的区别。

阶段	pushq rA	popq rA
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
译码	valA ← R[rA] valB ← R[%rsp]	valA ← R[%rsp] valB ← R[%rsp]
执行	valE ← valB+(−8)	valE ← valB+8
访存	M_8[valE]← valA	valE ← M_8[valA]
写回	R[%rsp]← valE	R[%rsp]← valE R[rA]← valM
更新 PC	PC ← valP	PC ← valP

图 4-20　Y86-64 指令 pushq 和 popq 在顺序实现中的计算。这些指令将值压入或弹出栈

pushq 指令开始时很像我们前面讲过的指令，但是在译码阶段，用%rsp 作为第二个寄存器操作数的标识符，将栈指针赋值为 valB。在执行阶段，用 ALU 将栈指针减 8。减过 8 的值就是内存写的地址，在写回阶段还会存回到%rsp 中。将 valE 作为写操作的地址，是遵循 Y86-64(和 x86-64)的惯例，也就是在写之前，pushq 应该先将栈指针减去 8，即使栈指针的更新实际上是在内存操作完成之后才进行的。

旁注　跟踪 pushq 指令的执行

让我们来看看图 4-17 中目标代码的第 6 行 pushq 指令的处理情况。此时，寄存器%rdx 的值为 9，而寄存器%rsp 的值为 128。我们还可以看到指令是位于地址 0x02a，有两个字节，值分别为 0xa0 和 0x2f。各个阶段的处理如下：