图 4-21 表明了三类控制转移指令的处理：各种跳转、call 和 ret。可以看到，我们能用同前面指令一样的整体流程来实现这些指令。

阶段	jXX Dest	call Dest	ret
取指	icode:ifun ← M_1[PC] valC ← M_8[PC+1] valP ← PC+9	icode:ifun ← M_1[PC] valC ← M_8[PC+1] valP ← PC+9	icode:ifun ← M_1[PC] valP ← PC+1
译码		 valB ← R[%rsp]	valA ← R[%rsp] valB ← R[%rsp]
执行	 Cnd ← Cond(CC, ifun)	valE ← valB+(−8)	valE ← valB+8
访存		M_8[valE] ← valP	valM ← M_8[valA]
写回		R[%rsp] ← valE	R[%rsp] ← valE
更新 PC	PC ← Cnd?valC:valP	PC ← valC	PC ← valM

图 4-21　Y86-64 指令 jXX、call 和 ret 在顺序实现中的计算。这些指令导致控制转移

同对整数操作一样，我们能够以一种统一的方式处理所有的跳转指令，因为它们的不同只在于判断是否要选择分支的时候。除了不需要一个寄存器指示符字节以外，跳转指令在取指和译码阶段都和前面讲的其他指令类似。在执行阶段，检查条件码和跳转条件来确定是否要选择分支，产生出一个一位信号 Cnd。在更新 PC 阶段，检查这个标志，如果这个标志为 1，就将 PC 设为 valC(跳转目标)，如果为 0，就设为 valP(下一条指令的地址)。我们的表示法 $x?a:b$ 类似于 C 语句中的条件表达式——当 x 非零时，它等于 a，当 x 为零时，等于 b。

旁注　跟踪 je 指令的执行

让我们来看看图 4-17 中目标代码的第 8 行 je 指令的处理情况。subq 指令(第 3 行)已经将所有的条件码都置为了 0，所以不会选择分支。该指令位于地址 0x02e，有 9 个字节。第一个字节的值为 0x73，而剩下的 8 个字节是数字 0x0000000000000040 按字节反过来得到的数，也就是跳转的目标。各个阶段的处理如下：

阶段	通用	具体
	jXX Dest	je 0x040
取指	icode:ifun ← M_1[PC] valC ← M_8[PC+1] valP ← PC+9	icode:ifun ← M_1[0x02e] = 7:3 valC ← M_8[0x02f] = 0x040 valP ← 0x02e+9 = 0x037
译码		
执行	 Cnd ← Cond(CC, ifun)	 Cnd ← Cond((0, 0, 0), 3) = 0
访存		
写回		
更新 PC	PC ← Cnd?valC:valP	PC ← 0? 0x040:0x037 = 0x037

就像这个跟踪记录表明的那样，这条指令的效果就是将 PC 加 9。

练习题 4.17　从指令编码(图 4-2 和图 4-3)我们可以看出，rmmovq 指令是一类更通用的、包括条件转移在内的指令的无条件版本。请给出你要如何修改下面 rrmovq 指令