暂停技术就是让一组指令阻塞在它们所处的阶段，而允许其他指令继续通过流水线。那么在本该正常处理 addq 指令的阶段中，我们该做些什么呢？我们使用的处理方法是：每次要把一条指令阻塞在译码阶段，就在执行阶段插入一个气泡。气泡就像一个自动产生的 nop 指令——它不会改变寄存器、内存、条件码或程序状态。在图 4-47 和图 4-48 的流水线图中，白色方框表示的就是气泡。在这些图中，我们用一个 addq 指令的标号为“D”的方框到标号为“E”的方框之间的箭头来表示一个流水线气泡，这些箭头表明，在执行阶段中插入气泡是为了替代 addq 指令，它本来应该经过译码阶段进入执行阶段。在 4.5.8 节中，我们将看到使流水线暂停以及插入气泡的详细机制。

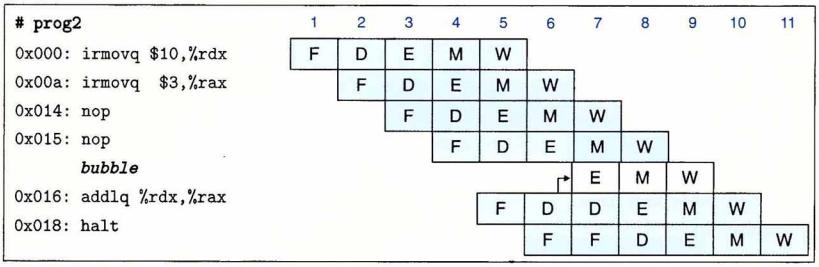


图 4-47　prog2 使用暂停的流水线化的执行。在周期 6 中对 addq 指令译码之后，暂停控制逻辑发现一个数据冒险，它是由写回阶段中对寄存器%rax 未进行的写造成的。它在执行阶段中插入一个气泡，并在周期 7 中重复对指令 addq 的译码。实际上，机器是动态地插入一条 nop 指令，得到的执行流类似于 prog1 的执行流(图 4-43)

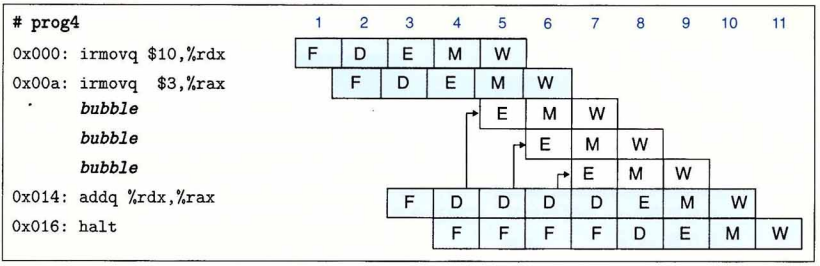


图 4-48　prog4 使用暂停的流水线化的执行。在周期 4 中对 addq 指令译码之后，暂停控制逻辑发现了对两个源寄存器的数据冒险。它在执行阶段中插入一个气泡，并在周期 5 中重复对指令 addq 的译码。它再次发现对两个源寄存器的冒险，就在执行阶段中插入一个气泡，并在周期 6 中重复对指令 addq 的译码。它再次发现对寄存器%rax 的冒险，就在执行阶段中插入一个气泡，并在周期 7 中重复对指令 addq 的译码。实际上，机器是动态地插入三条 nop 指令，得到的执行流类似于 prog1 的执行流(图 4-43)

在使用暂停技术来解决数据冒险的过程中，我们通过动态地产生和 prog1 流(图 4-43)一样的流水线流，有效地执行了程序 prog2 和 prog4。为 prog2 插入 1 个气泡，为 prog4 插入 3 个气泡，与在第 2 条 irmovq 指令和 addq 指令之间有 3 条 nop 指令，有相同的效果。虽然实现这一机制相当容易(参考家庭作业 4.53)，但是得到的性能并不很好。一条指令更新一个寄存器，紧跟其后的指令就使用被更新的寄存器，像这样的情况不胜枚举。这会导致流水线暂停长达三个周期，严重降低了整体的吞吐量。