我们看到 prog1 通过流水线并得到正确的结果，因为 3 条 nop 指令在有数据相关的指令之间创造了一些延迟。让我们来看看如果去掉这些 nop 指令会发生些什么。图 4-44 描述的是 prog2 程序的流水线流程，在两条产生寄存器%rdx 和%rax 值的 irmovq 指令和以这两个寄存器作为操作数的 addq 指令之间有两条 nop 指令。在这种情况下，关键步骤发生在周期 6，此时 addq 指令从寄存器文件中读取它的操作数。该图底部是这个周期内流水线活动的扩展描述。第一个 irmovq 指令已经通过了写回阶段，因此程序寄存器%rdx 已经在寄存器文件中更新过了。在该周期内，第二个 irmovq 指令处于写回阶段，因此对程序寄存器%rax 的写要到周期 7 开始，时钟上升时，才会发生。结果，会读出%rax 的错误值(回想一下，我们假设所有的寄存器的初始值为 0)，因为对该寄存器的写还未发生。很明显，我们必须改进流水线让它能够正确处理这样的冒险。

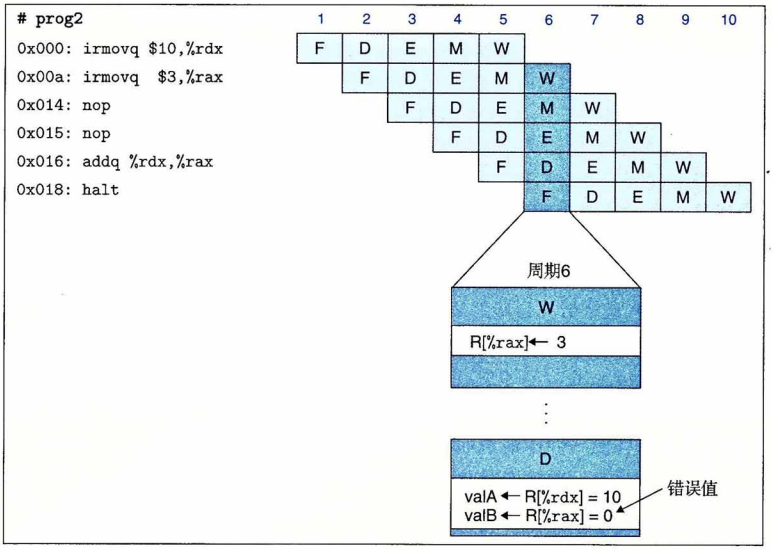

图 4-44　prog2 的流水线化的执行，没有特殊的流水线控制。直到周期 7 结束时，对寄存器%rax 的写才发生，所以 addq 指令在译码阶段读出的是该寄存器的错误值

图 4-45 是当 irmovq 指令和 addq 指令之间只有一条 nop 指令，即为程序 prog3 时，发生的情况。现在我们必须检查周期 5 内流水线的行为，此时 addq 指令通过译码阶段。不幸的是，对寄存器%rdx 的写仍处在写回阶段，而对寄存器%rax 的写还处在访存阶段。因此，addq 指令会得到两个错误的操作数。

图 4-46 是当去掉 irmovq 指令和 addq 指令间的所有 nop 指令，即为程序 prog4 时，发生的情况。现在我们必须检查周期 4 内流水线的行为，此时 addq 指令通过译码阶段。不幸的是，对寄存器%rdx 的写仍处在访存阶段，而执行阶段正在计算寄存器%rax 的新值。因此，addq 指令的两个操作数都是不正确的。

这些例子说明，如果一条指令的操作数被它前面三条指令中的任意一条改变的话，都会出现数据冒险。之所以会出现这些冒险，是因为我们的流水线化的处理器是在译码阶段从寄存器文件中读取指令的操作数，而要到三个周期以后，指令经过写回阶段时，才会将指令的结果写到寄存器文件。