钟变化会引发一个经过组合逻辑的流，来执行整个指令。让我们来看看这些硬件怎样实现表中列出的这一行为。

SEQ 的实现包括组合逻辑和两种存储器设备：时钟寄存器(程序计数器和条件码寄存器)，随机访问存储器(寄存器文件、指令内存和数据内存)。组合逻辑不需要任何时序或控制——只要输入变化了，值就通过逻辑门网络传播。正如提到过的那样，我们也将读随机访问存储器看成和组合逻辑一样的操作，根据地址输入产生输出字。对于较小的存储器来说(例如寄存器文件)，这是一个合理的假设，而对于较大的电路来说，可以用特殊的时钟电路来模拟这个效果。由于指令内存只用来读指令，因此我们可以将这个单元看成是组合逻辑。

现在还剩四个硬件单元需要对它们的时序进行明确的控制——程序计数器、条件码寄存器、数据内存和寄存器文件。这些单元通过一个时钟信号来控制，它触发将新值装载到寄存器以及将值写到随机访问存储器。每个时钟周期，程序计数器都会装载新的指令地址。只有在执行整数运算指令时，才会装载条件码寄存器。只有在执行 rmmovq、pushq 或 call 指令时，才会写数据内存。寄存器文件的两个写端口允许每个时钟周期更新两个程序寄存器，不过我们可以用特殊的寄存器 ID 0xF 作为端口地址，来表明在此端口不应该执行写操作。

要控制处理器中活动的时序，只需要寄存器和内存的时钟控制。硬件获得了如图 4-18～图 4-21 的表中所示的那些赋值顺序执行一样的效果，即使所有的状态更新实际上同时发生，且只在时钟上升开始下一个周期时。之所以能保持这样的等价性，是由于 Y86-64 指令集的本质，因为我们遵循以下原则组织计算：

**原则**：*从不回读*

*处理器从来不需要为了完成一条指令的执行而去读由该指令更新了的状态。*

这条原则对实现的成功来说至关重要。为了说明问题，假设我们对 pushq 指令的实现是先将%rsp 减 8，再将更新后的%rsp 值作为写操作的地址。这种方法同前面所说的那个原则相违背。为了执行内存操作，它需要先从寄存器文件中读更新过的栈指针。然而，我们的实现(图 4-20)产生出减后的栈指针值，作为信号 valE，然后再用这个信号既作为寄存器写的数据，也作为内存写的地址。因此，在时钟上升开始下一个周期时，处理器就可以同时执行寄存器写和内存写了。

再举个例子来说明这条原则，我们可以看到有些指令(整数运算)会设置条件码，有些指令(跳转指令)会读取条件码，但没有指令必须既设置又读取条件码。虽然要到时钟上升开始下一个周期时，才会设置条件码，但是在任何指令试图读之前，它们都会更新。

以下是汇编代码，左边列出的是指令地址，图 4-25 给出了 SEQ 硬件如何处理其中第 3 和第 4 行指令：

```
1      0x000:   irmovq $0x100,%rbx   # %rbx <-- 0x100
2      0x00a:   irmovq $0x200,%rdx   # %rdx <-- 0x200
3      0x014:   addq %rdx,%rbx       # %rbx <-- 0x300 CC <-- 000
4      0x016:   je dest              # Not taken
5      0x01f:   rmmovq %rbx,0(%rdx)  # M[0x200] <-- 0x300
6      0x029: dest: halt
```

标号为 1～4 的各个图给出了 4 个状态单元，还有组合逻辑，以及状态单元之间的连接。组合逻辑被条件码寄存器环绕着，因为有的组合逻辑(例如 ALU)产生输入到条件码寄存器，而其他部分(例如分支计算和 PC 选择逻辑)又将条件码寄存器作为输入。图中寄