对一条跳转指令来说，这个阶段会决定是不是应该选择分支。

- **访存**(memory)：访存阶段可以将数据写入内存，或者从内存读出数据。读出的值为 valM。
- **写回**(write back)：写回阶段最多可以写两个结果到寄存器文件。
- **更新 PC**(PC update)：将 PC 设置成下一条指令的地址。

处理器无限循环，执行这些阶段。在我们简化的实现中，发生任何异常时，处理器就会停止：它执行 halt 指令或非法指令，或它试图读或者写非法地址。在更完整的设计中，处理器会进入异常处理模式，开始执行由异常的类型决定的特殊代码。

从前面的讲述可以看出，执行一条指令是需要进行很多处理的。我们不仅必须执行指令所表明的操作，还必须计算地址、更新栈指针，以及确定下一条指令的地址。幸好每条指令的整个流程都比较相似。因为我们想使硬件数量尽可能少，并且最终将把它映射到一个二维的集成电路芯片的表面，在设计硬件时，一个非常简单而一致的结构是非常重要的。降低复杂度的一种方法是让不同的指令共享尽量多的硬件。例如，我们的每个处理器设计都只含有一个算术/逻辑单元，根据所执行的指令类型的不同，它的使用方式也不同。在硬件上复制逻辑块的成本比软件中有重复代码的成本大得多。而且在硬件系统中处理许多特殊情况和特性要比用软件来处理困难得多。

我们面临的一个挑战是将每条不同指令所需要的计算放入到上述那个通用框架中。我们会使用图 4-17 中所示的代码来描述不同 Y86-64 指令的处理。图 4-18～图 4-21 中的表描述了不同 Y86-64 指令在各个阶段是怎样处理的。很值得仔细研究一下这些表。表中的这种格式很容易映射到硬件。表中的每一行都描述了一个信号或存储状态的分配(用分配操作←来表示)。阅读时可以把它看成是从上至下的顺序求值。当我们将这些计算映射到硬件时，会发现其实并不需要严格按照顺序来执行这些求值。

```
1   0x000: 30f20900000000000000 |     irmovq $9,  %rdx
2   0x00a: 30f31500000000000000 |     irmovq $21, %rbx
3   0x014: 6123                 |     subq %rdx, %rbx              # subtract
4   0x016: 30f48000000000000000 |     irmovq $128,%rsp             # Problem 4.13
5   0x020: 4043640000000000000  |     rmmovq %rsp, 100(%rbx)       # store
6   0x02a: a02f                 |     pushq %rdx                   # push
7   0x02c: b00f                 |     popq  %rax                   # Problem 4.14
8   0x02e: 734000000000000000   |     je done                      # Not taken
9   0x037: 804100000000000000   |     call proc                    # Problem 4.18
10  0x040:                      | done:
11  0x040: 00                   |     halt
12  0x041:                      | proc:
13  0x041: 90                   |     ret                          # Return
14                              |
```

图 4-17　Y86-64 指令序列示例。我们会跟踪这些指令通过各个阶段的处理

图 4-18 给出了对 OPq(整数和逻辑运算)、rrmovq(寄存器-寄存器传送)和 irmovq(立即数-寄存器传送)类型的指令所需的处理。让我们先来考虑一下整数操作。回顾图 4-2，可以看到我们小心地选择了指令编码，这样四个整数操作(addq、subq、andq 和 xorq)都有相同的 icode 值。我们可以以相同的步骤顺序来处理它们，除了 ALU 计算必须根据 ifun 中编码的具体的指令操作来设定。