分，其详细实现将在 4.5.8 节给出。

对于 ret 指令，考虑下面的示例程序。这个程序是用汇编代码表示的，左边是各个指令的地址，以供参考：

```
0x000:     irmovq stack,%rsp  #   Initialize stack pointer
0x00a:     call proc          #   Procedure call
0x013:     irmovq $10,%rdx    #   Return point
0x01d:     halt
0x020: .pos 0x20
0x020: proc:                  # proc:
0x020:     ret                #   Return immediately
0x021:     rrmovq %rdx,%rbx   #   Not executed
0x030: .pos 0x30
0x030: stack:                 # stack: Stack pointer
```

图 4-55 给出了我们希望流水线如何来处理 ret 指令。同前面的流水线图一样，这幅图展示了流水线的活动，时间从左向右增加。与前面不同的是，指令列出的顺序与它们在程序中出现的顺序并不相同，这是因为这个程序的控制流中指令并不是按线性顺序执行的。看看指令的地址就能看出它们在程序中的位置。

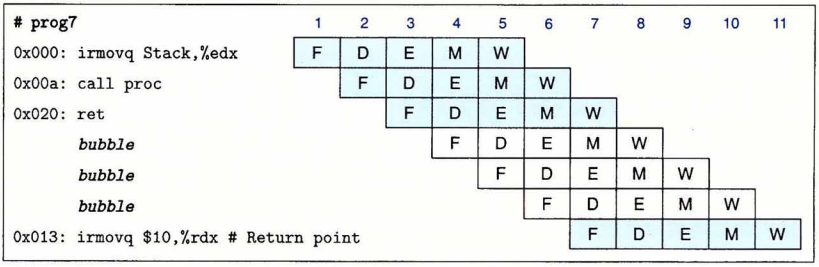

图 4-55　ret 指令处理的简化视图。当 ret 经过译码、执行和访存阶段时，流水线应该暂停，在处理过程中插入三个气泡。一旦 ret 指令到达写回阶段(周期 7)，PC 选择逻辑就会选择返回地址作为指令的取指地址

如这张图所示，在周期 3 中取出 ret 指令，并沿着流水线前进，在周期 7 进入写回阶段。在它经过译码、执行和访存阶段时，流水线不能做任何有用的活动。我们只能在流水线中插入三个气泡。一旦 ret 指令到达写回阶段，PC 选择逻辑就会将程序计数器设为返回地址，然后取指阶段就会取出位于返回点(地址 0x013)处的 irmovq 指令。

要处理预测错误的分支，考虑下面这个用汇编代码表示的程序，左边是各个指令的地址，以供参考：

```
0x000:     xorq %rax,%rax
0x002:     jne  target        # Not taken
0x00b:     irmovq $1, %rax    # Fall through
0x015:     halt
0x016: target:
0x016:     irmovq $2, %rdx    # Target
0x020:     irmovq $3, %rbx    # Target+1
0x02a:     halt
```

图 4-56 表明是如何处理这些指令的。同前面一样，指令是按照它们进入流水线的顺