# 处理器设计

## 流水线中的控制冒险

主讲人: 邓倩妮

上海交通大学

部分内容来自：

Computer Organization and Design, 4th Edition, Patterson & Hennessy

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

# 本节主要内容

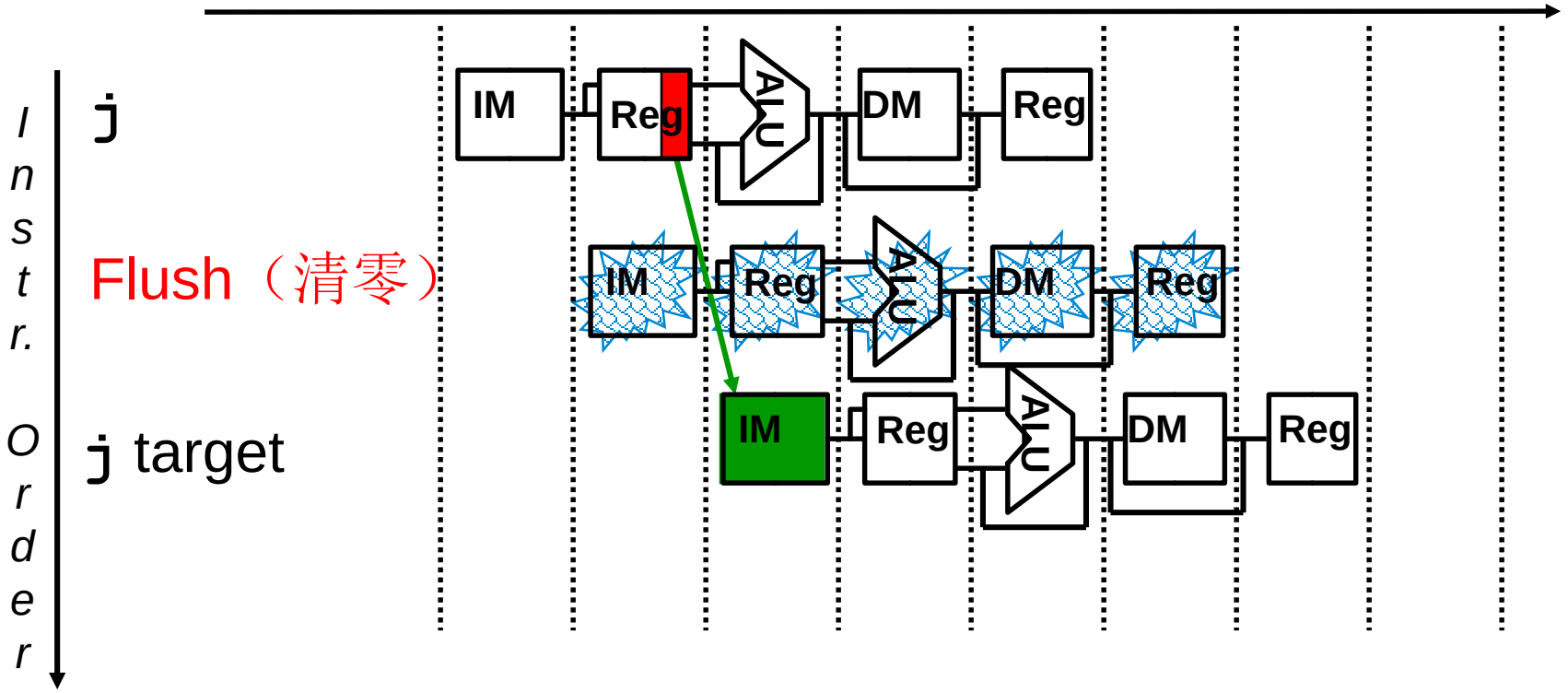

- 控制冒险
- 控制冒险的解决方案

# 控制冒险

- 控制冒险（control hazards）
- 当流水线遇到分支指令和其他改变 PC值（不再是PC+4）的指令时，就会发生控制冒险。
- 例如 ：
  - 无条件转移 (j, jal, jr)：跳转到指定位置
  - 条件转移 (beq, bne)
    - 转移失败：PC值加4
    - 转移成功：将PC值改变为转移目标地址
- 处理转移指令最简单的方法
  - 一旦检测到转移指令(在ID段)，就停顿执行其后的指令，直到分支指令到达MEM段，确定出新的PC值为止。

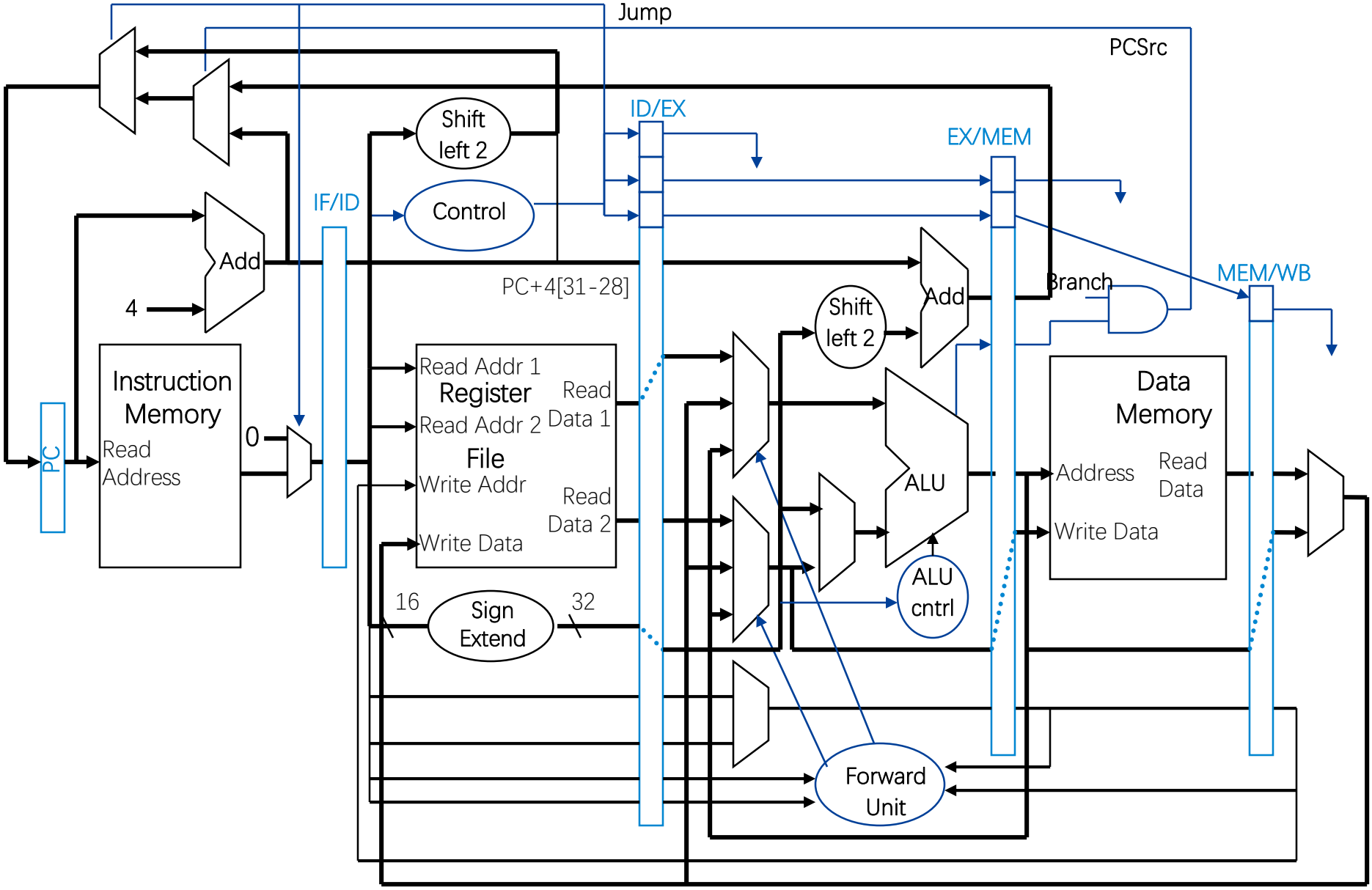
# 无条件转移（Jumps） 引发控制冒险

- Jumps指令
  - 在 ID 段译码，此时IF段已经取了它后面的指令
  - 需要将IF/ID 段中的指令清零（ flush）




- Jump 指令出现得不算频繁，在SPECint 中的jump指令仅占比 3%

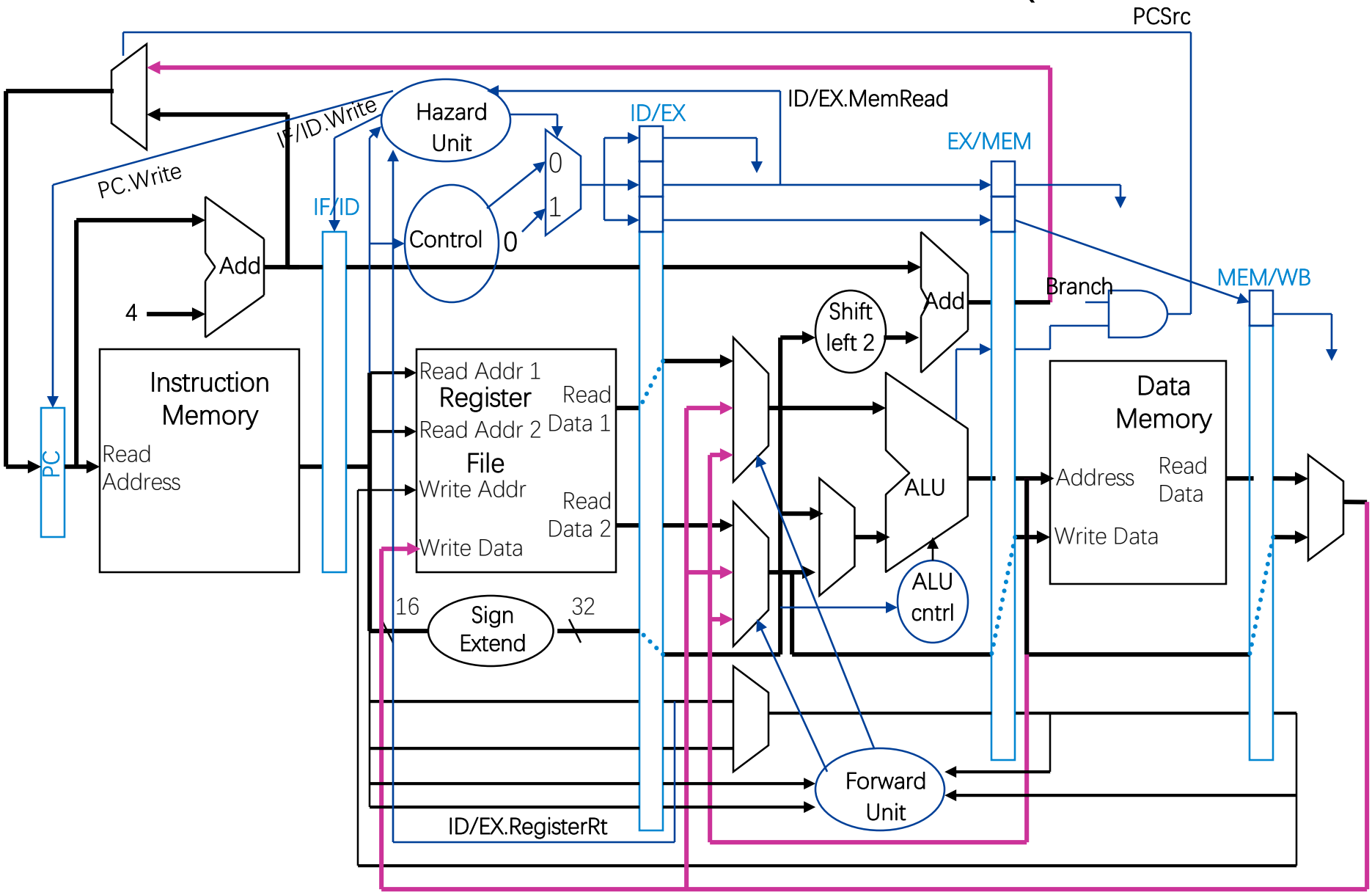
# Jumps 指令的支持- 如何停顿（清零）？



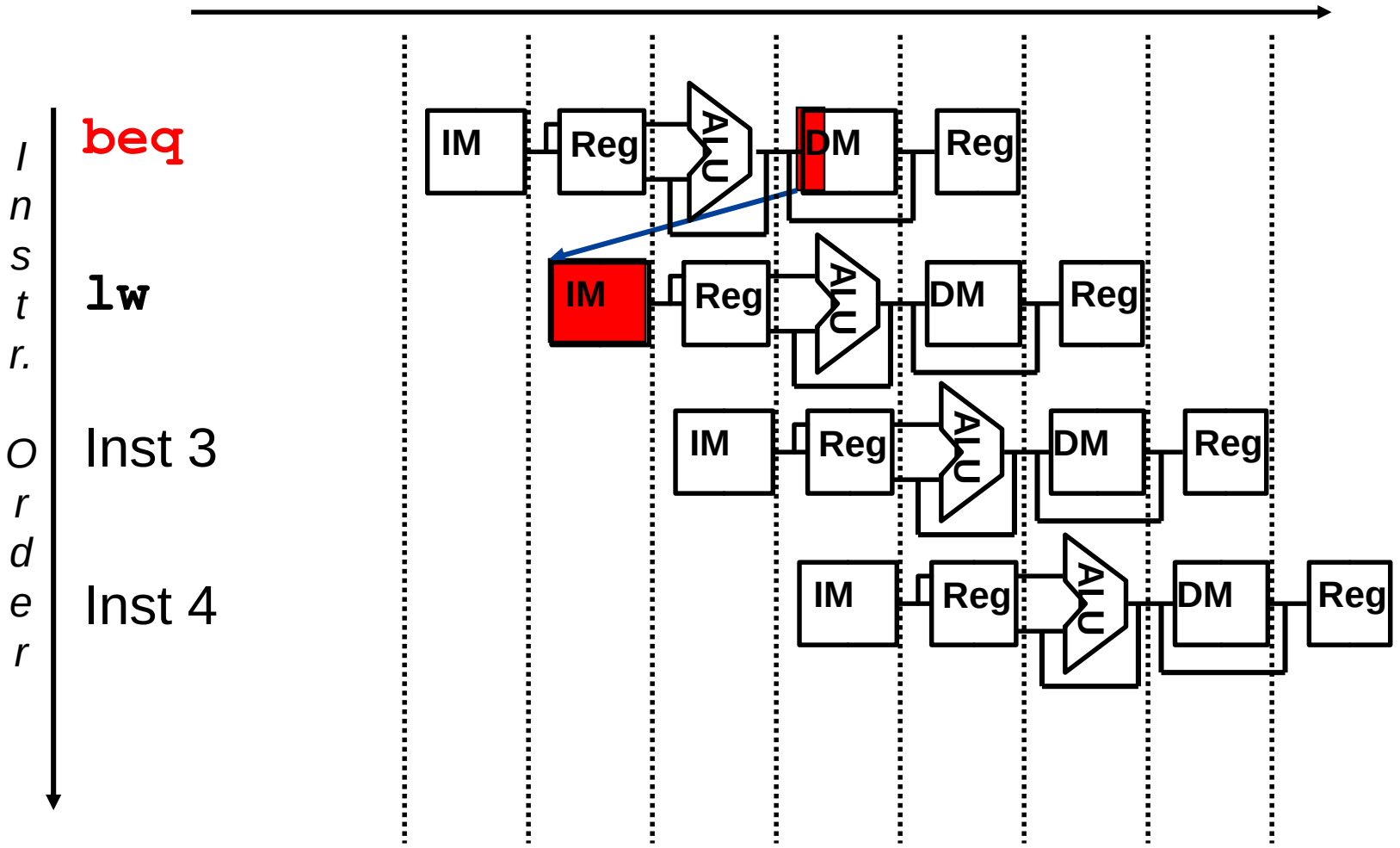
# 两“种”停顿（stall）

- 清零（Flushes）
  - 例如：代码顺序跟在 j 和 beq 后的指令
  - 被替换为空（noop）指令
    - 转移指令后发射的指令，所在的流水段寄存器全部 置“0”（flush）
- 气泡 (bubble) ：插入在两条指令之间
  - 例如：针对“访存-使用”冒险（load-use hazards）
  - 当一条指令被暂停时，暂停在其后发射的指令，
    - 将适当的流水段寄存器中的控制信号置“0”，以插入气泡
    - 被暂停的指令及其后发射的指令，维持在原流水段不动
    - 继续执行在其前发射的 指令，让它们继续沿流水线执行

# 对比：冲突检测的支持：添加停顿(气泡)

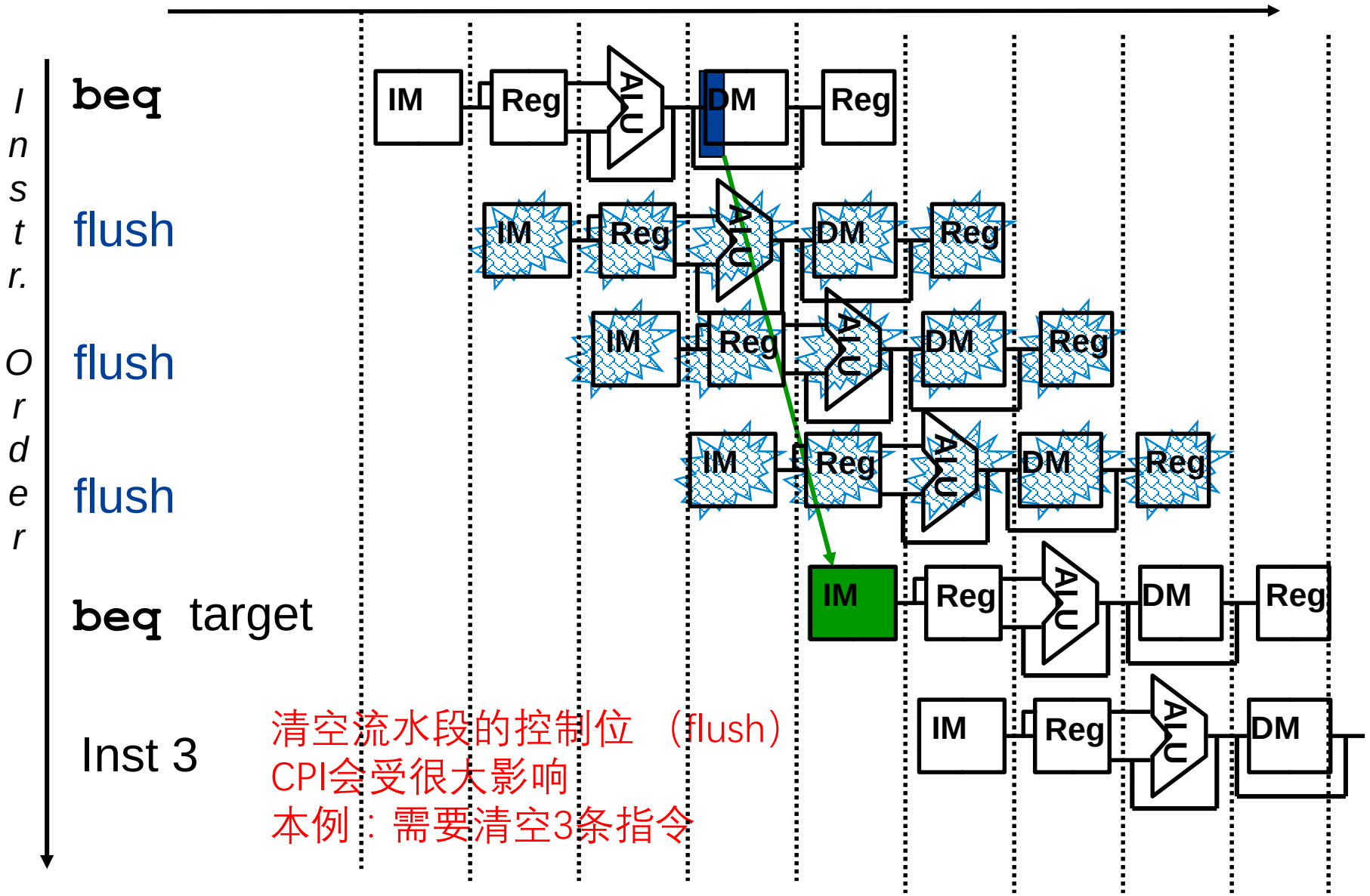


# 条件转移指令引发控制冒险




- 新 PC 的值要在MEM 段才计算， 引发了控制冒险

# 解决控制冒险最简单的方法：停顿（清零）



# 控制冒险对CPI的影响

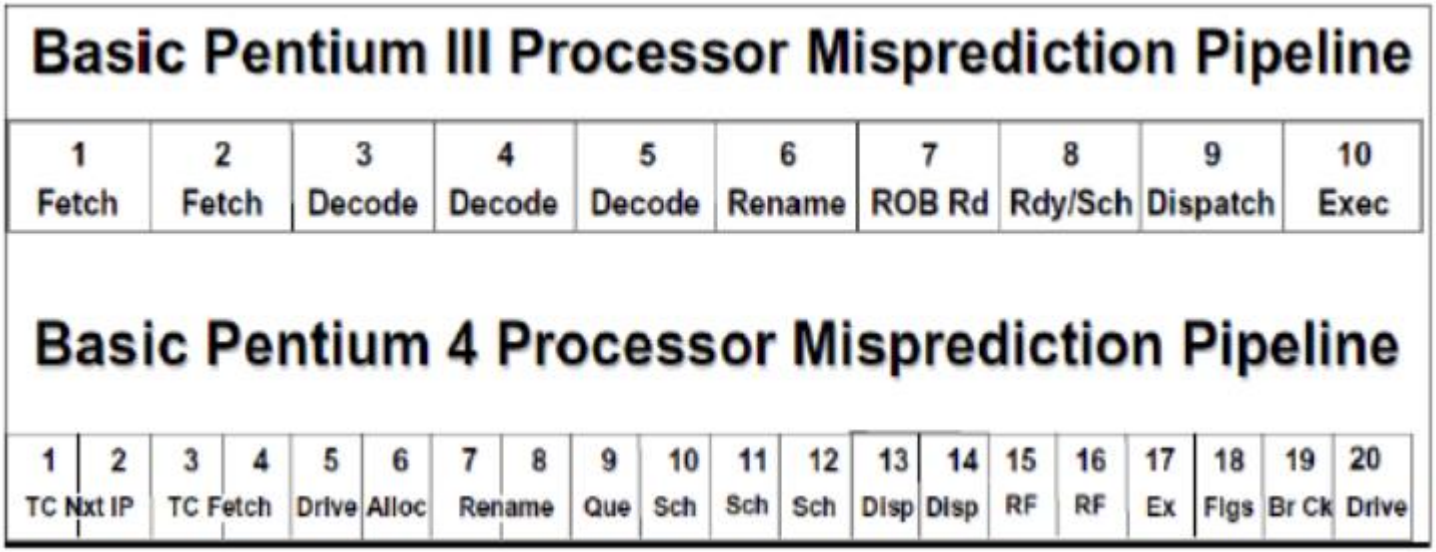
- 条件转移（分支转移）成功导致暂停3个时钟周期。
- 若分支指令的频度为30%，理想*CPI*=1，

  则 实际CPI（cycle per instr.）=1+30%×3≈ 2

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 分支指令 | IF | ID | EX | MEM | WB | | | | |
| 分支后继指令 | | IF | stall | stall | IF | ID | EX | MEM | WB |
| 分支后继指令+1 | | | | | IF | ID | EX | MEM | WB |
| 分支后继指令+2 | | | | | | IF | ID | EX | MEM |
| 分支后继指令+3 | | | | | | | IF | ID | EX |
| 分支后继指令+4 | | | | | | | | IF | ID |
| 分支后继指令+5 | | | | | | | | | IF |

# 流水段数越深，控制冒险的开销越大

- 举例：
- Pentium 3：转移开销 10周期
- Pentium 4： 转移开销 20周期

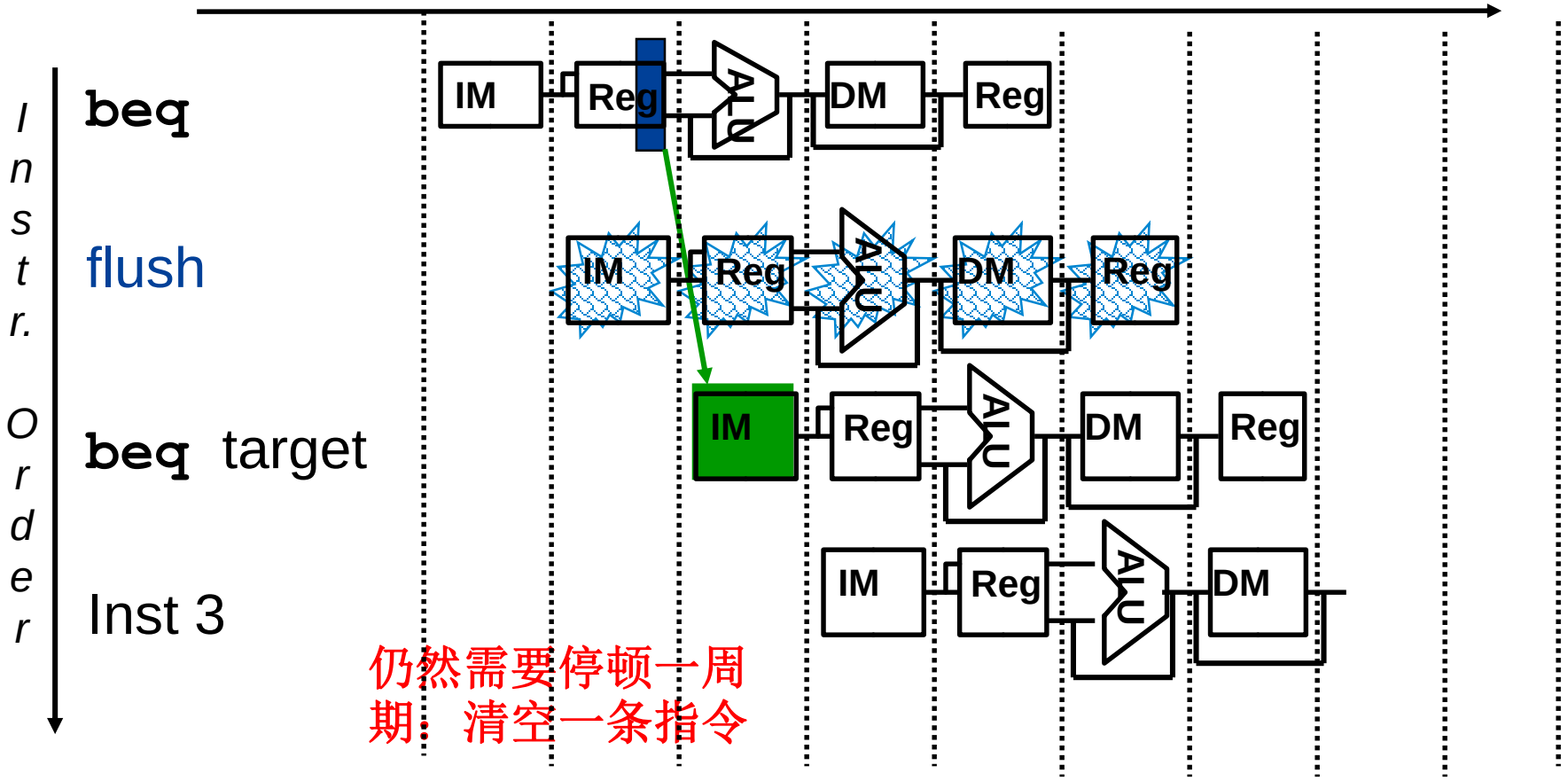


# 控制冒险的解决方案


- 相比数据冒险，控制冒险（control hazards）的出现频率小，但仍需要重视
- 控制冒险会严重影响CPI
  - 流水段越长，影响越大
- 可能的解决方案
  - 停顿 (对性能影响大)
  - 让branch决策尽量早开始
  - 设置转移延迟槽(需要编译器支持)
  - 转移预测：静态、动态

# 另一个解决控制冒险的方案：提前决策

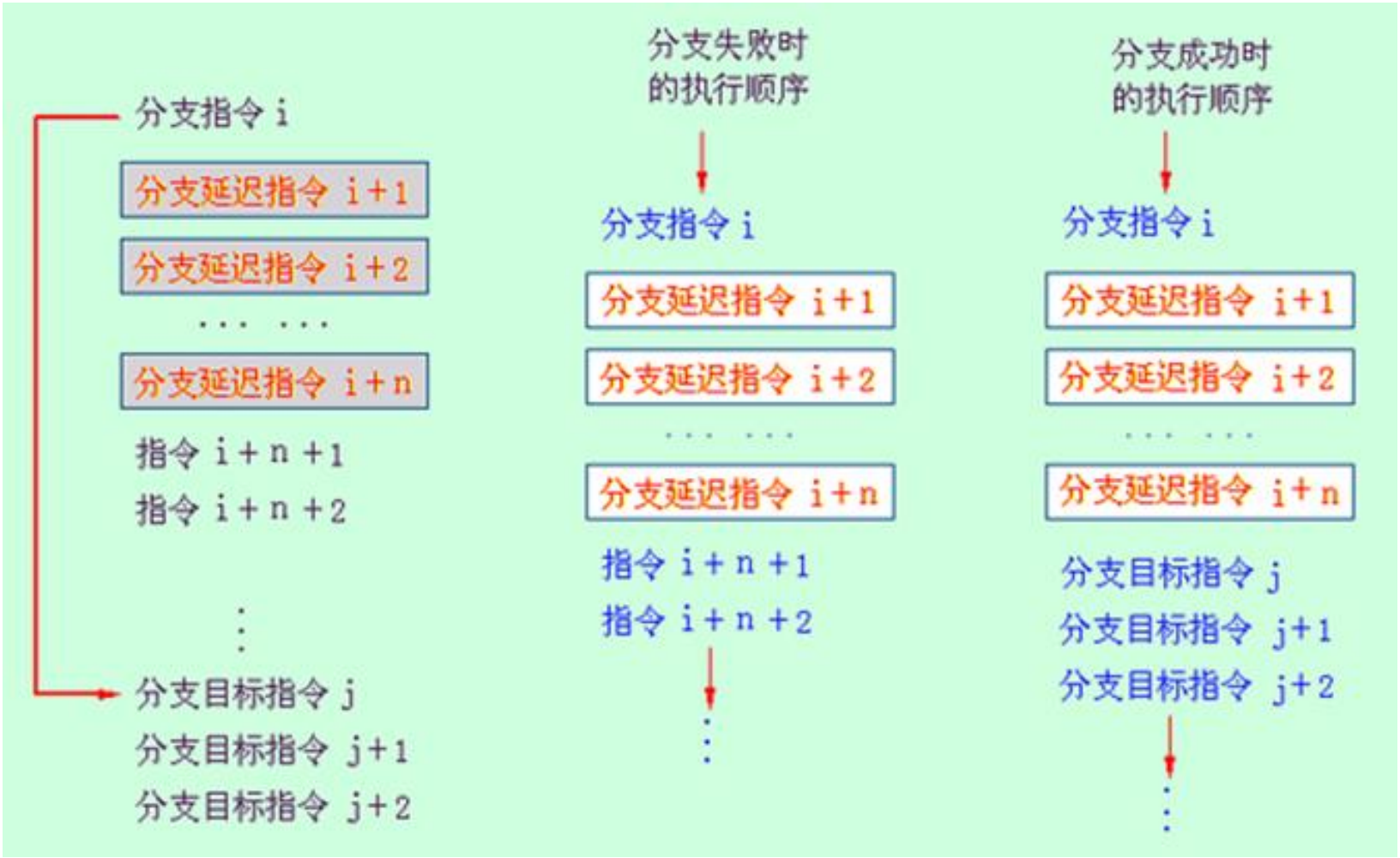
- 让branch决策尽量早开始
- 在译码阶段完成条件的比较（比较单元）、计算新的地址（加法器）、PC更新



# 进一步优化：设置转移延迟槽（Branch Slot）

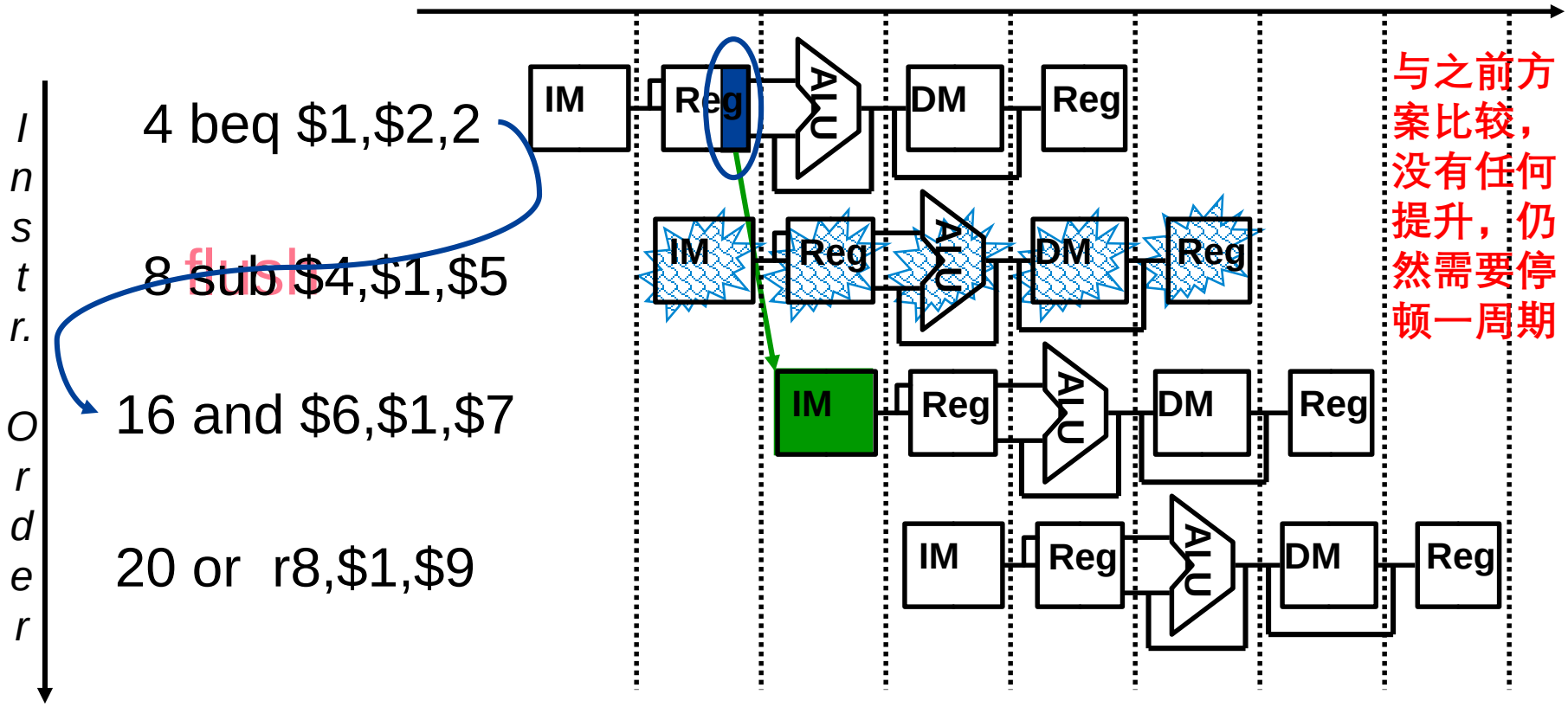
- 基本思想：
  - 编译器将一条（或几条）肯定会执行的指令（无论转移或是不转移时）移到branch指令后面，隐藏转移指令的延迟；
  - 需要编译器支持
- 转移延迟槽（branch slot）的大小？
  - 五阶段流水，在译码阶段完成转移决策，转移延迟槽（branch slot）大小为1
  - 延迟开销为n 的转移指令，其后紧跟有n 个延迟槽
  - 流水段级数多的情况下，转移延迟槽（branch slot）大小可能要远远超过1，这对编译器要求高。

# 转移延迟槽（大小为n）


分支指令 i
分支延迟指令 i+1
分支延迟指令 i+2
分支延迟指令 i+n
指令 i+n+1
指令 i+n+2
分支目标指令 j
分支目标指令 j+1
分支目标指令 j+2
分支失败时
的执行顺序
分支指令 i
分支延迟指令 i+1
分支延迟指令 i+2
分支延迟指令 i+n
指令 i+n+1
指令 i+n+2
分支成功时
的执行顺序
分支指令 i
分支延迟指令 i+1
分支延迟指令 i+2
分支延迟指令 i+n
分支目标指令 j
分支目标指令 j+1
分支目标指令 j+2

# 另一个解决控制冒险的方案：转移预测

- 静态预测：预测转移指令不转移(not taken)




- branch决策在译码段完成
- 清空IF/ID 流水段寄存器(将其变为一条空（ noop ）指令)

# 静态转移（分支）预测（续）

- 静态预测：预测转移指令总是转移( taken)
  - 总会发生一周期停顿
  - 因为：在ID段，才知道它是转移指令，并计算出转移地址
- 静态转移预测总是会影响性能
  - 流水段数越深，影响越大
  - 解决方法： 动态转移预测

# 动态转移（分支）预测

- 现代处理器一般都采用的隐藏转移延迟的方法
- 需要预测两部分信息：
  1. 预测分支结果：（Predict Branch Outcome
     - 转 or 不转？
  2. 预测分支目标地址：（Predict Branch/Jump Address）
     - 转到什么位置

# 动态转移预测

## 1. 预测：转 / 不转 ？

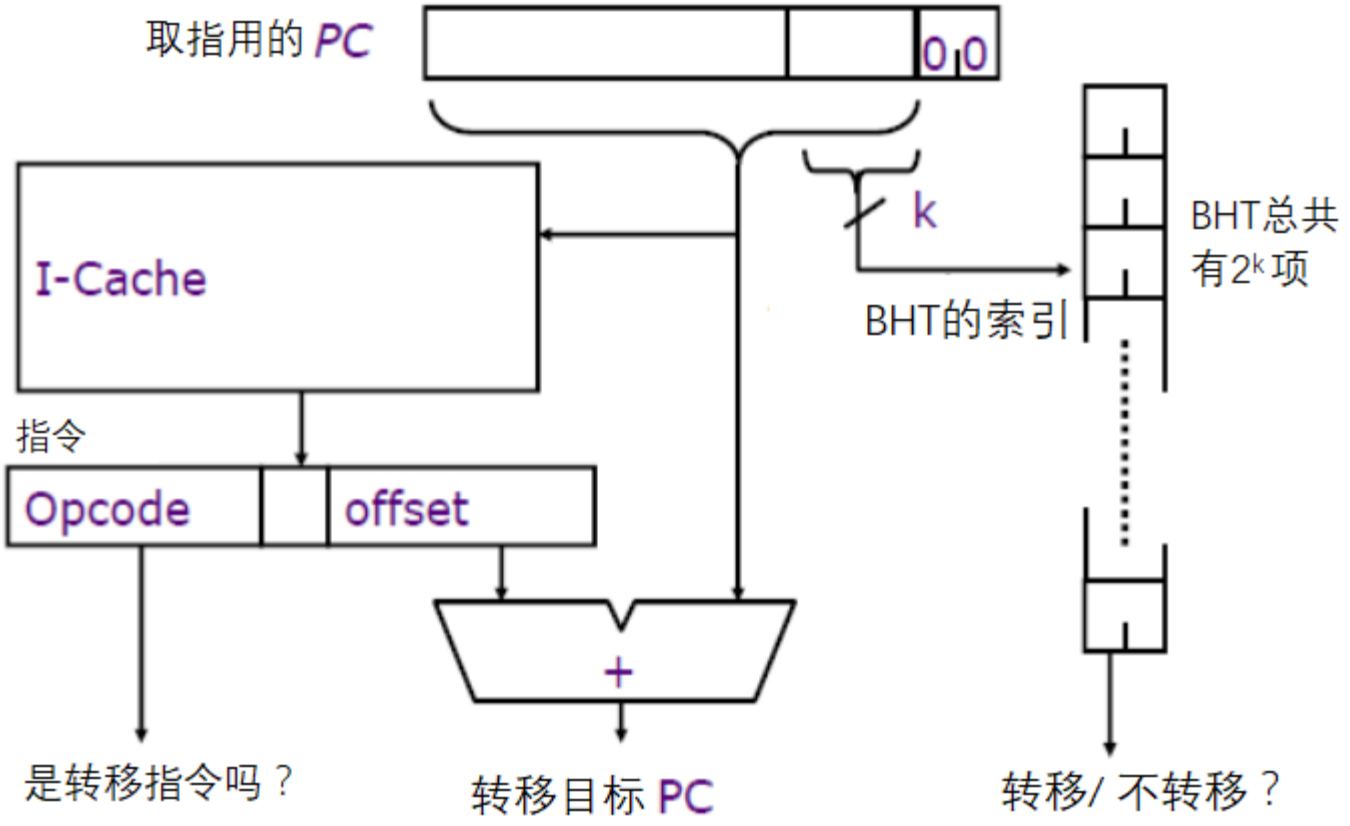
- 转移历史表 Branch History Table，简写为BHT：
  使用一片存储区域，记录最近一次或几次分支特征的历史。

## 2.预测：转到什么位置？

- Branch Target Buffer: 简写为BTB

将转移指令的地址、该转移指令的目标地址都放到一个缓冲区中保存起来

# 转移历史表 Branch History Table (BHT)




- 索引：

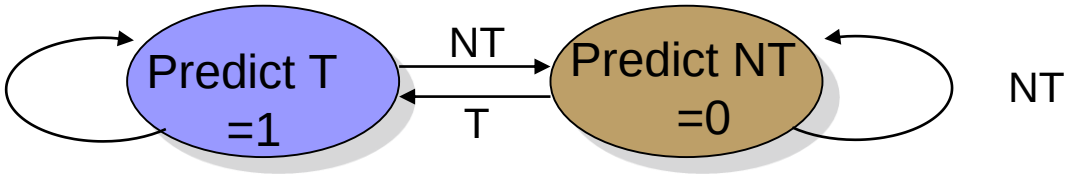
分支指令地址的低位。

- 存储区：

1位或2位的分支历史记录位，又称为预测位。

一个4096 位的 BHT 预测错误率 可以从1% (nasa7, tomcatv) 到 18% (eqntott), 平均预测准确率 80%-99%

# 1 位预测器




```
Loop: 1st loop instr
      2nd loop instr
            .
            .
            .
      last loop instr
      bne $1,$2,Loop
      fall out instr
```

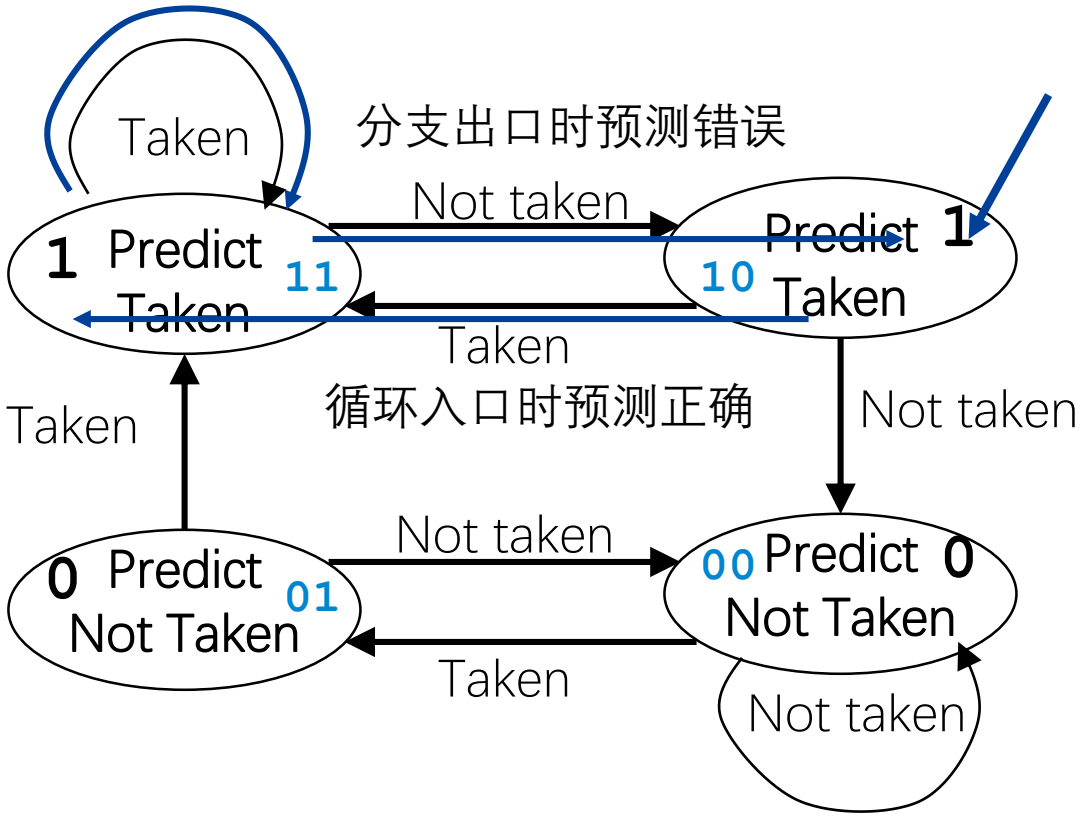
假设从预测位 0 开始 (预测不转移)

进入第一轮循环：预测错误；预测位翻转 ( = 1)，预测转移

- 只要循环转移，预测正确
- 分支出口，循环不再转移，预测错误，预测位翻转（ = 0），预测不转移
- 预测错误两次

- 循环10次，预测正确率 80% prediction accuracy

# 2 位预测器



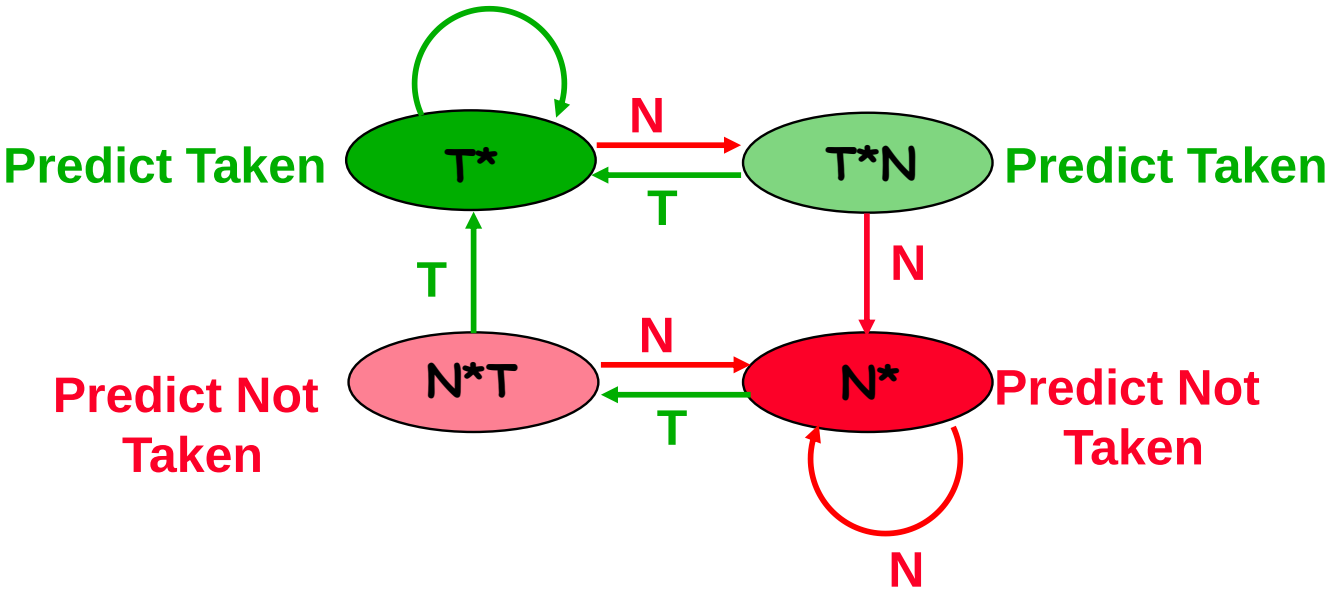

```
Loop: 1st loop instr
      2nd loop instr
            .
            .
            .
      last loop instr
      bne $1,$2,Loop
      fall out instr
```

- BHT 的每一项表示一个两位的有限状态机

**循环10次，预测正确9次，预测正确率90%**

# 举例：双重循环的两位动态预测

```
into sum (int N)
{
int i, j, sum=0;
for (i=0; i < N; i++)
    for (j=0; j <N; j++)
        sum=sum+1;
return sum;
}
```

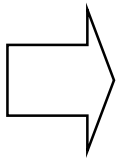



两位预测器，初始状态为：T*N.

当 N=10 或 100， 转移预测正确率分别为多少？

# 举例：双重循环的两位动态预测（续）

```
into sum (int N)
{
int i, j, sum=0;
for (i=0; i < N; i++)
    for (j=0; j <N; j++)
        sum=sum+1;
return sum;
}
```



```
… …
Loop-i:  beq $t1,$a0, exit-i    # 若( i=N)则跳出外循环
         add $t2, $zero, $zero  #j=0
Loop-j:  beq $t2, $a0, exit-j   # 若(j=N)则跳出内循环
         addi $t2, $t2, 1       # j=j+1
         addi $t0, $t0, 1       #sum=sum+1
         j  Loop-j
exit-j:  addi $t1, $t1, 1       # i=i +1
         j  Loop-i
exit-i:  … …
```

外循环中的分支指令共执行N+1次，
内循环中的分支指令共执行N×(N+1)次。

**N=10, 分别90.9%和90.9%**

**N=100, 分别99%和99%**

预测位初始为T*N，外循环只有最后一次预测错误；跳出内循环时预测位变为01，再进入内循环时，第一次预测正确，只有最后一次预测错误，因此，总共有N次预测错误。

**N越大准确率越高！**

# 动态转移预测

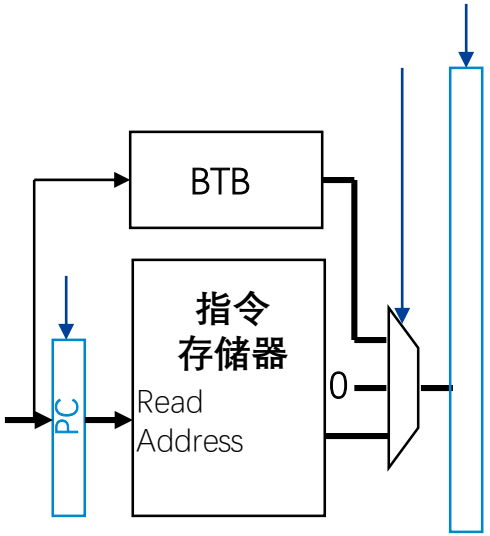
## 1. 预测：转 / 不转 ？

- Branch History Table，简写为BHT：
  使用一片存储区域，记录最近一次或几次分支特征的历史。

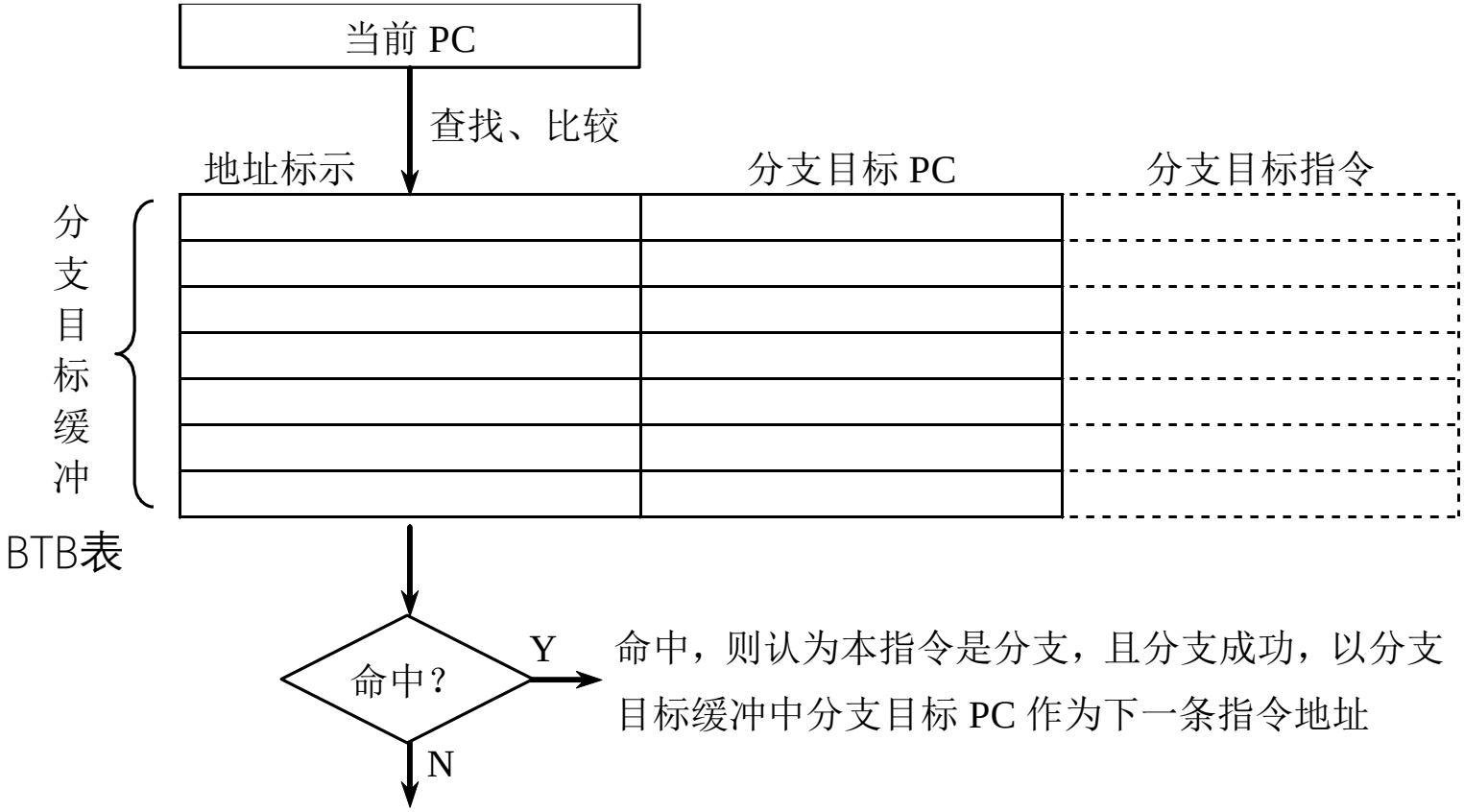
## 2.预测：转到什么位置？

- Branch Target Buffer：简写为BTB

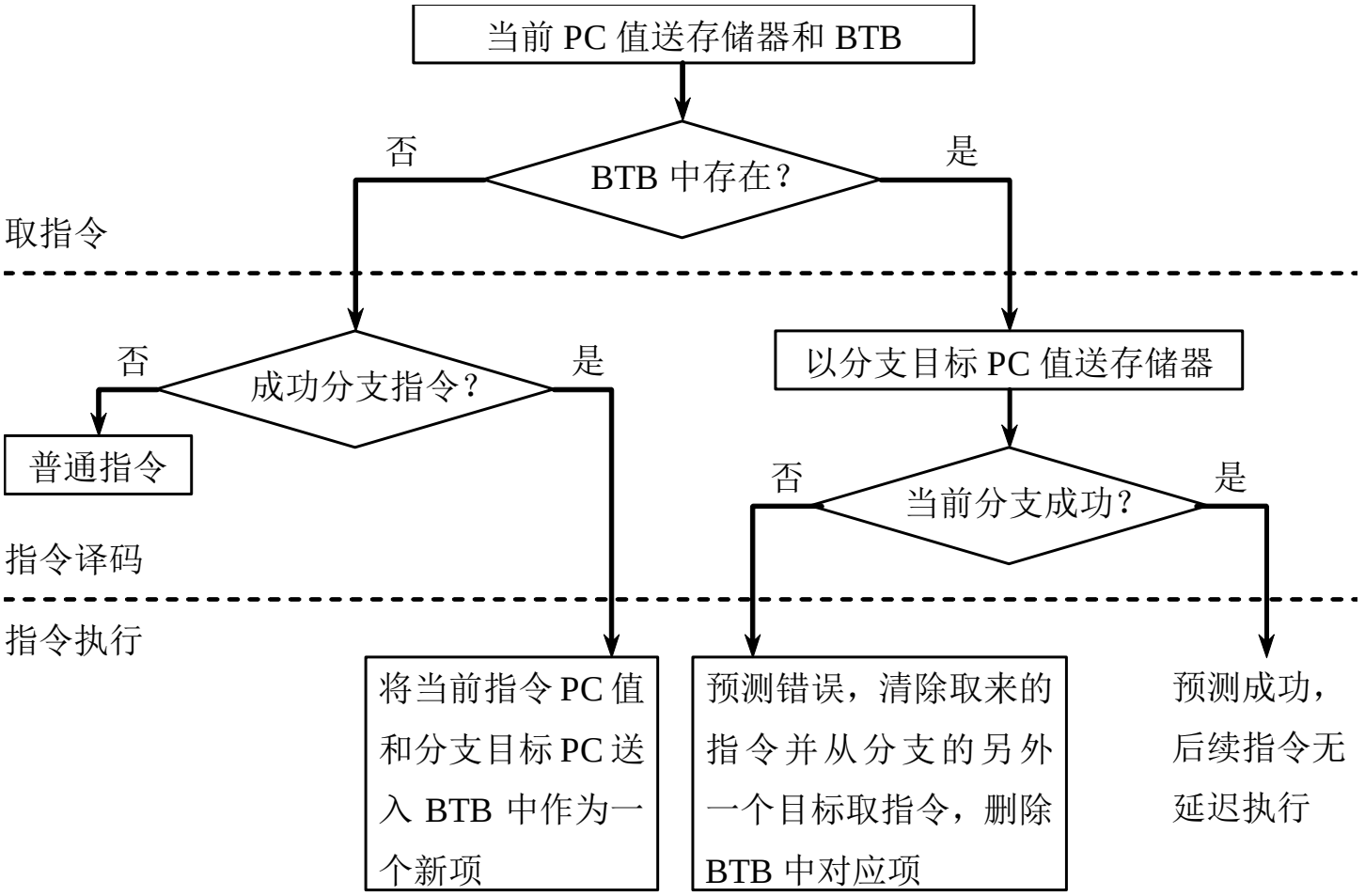
将转移指令的地址、该转移指令的目标地址都放到一个缓冲区中保存起来



# 转移（分支）目标缓冲结构

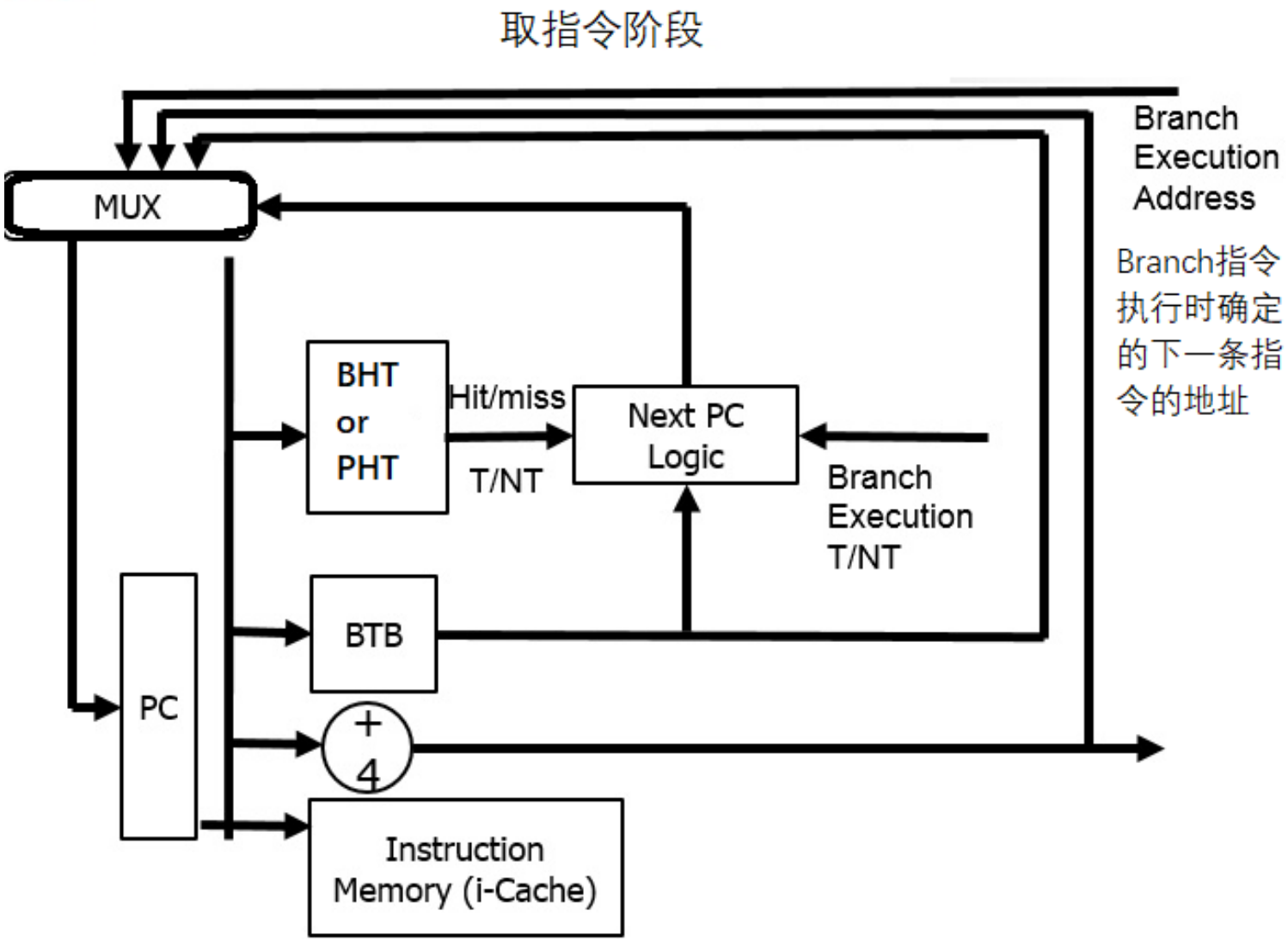

当前 PC
查找、比较
地址标示
分支目标 PC
分支目标指令
分支目标缓冲
BTB表
命中？
Y
命中，则认为本指令是分支，且分支成功，以分支目标缓冲中分支目标 PC 作为下一条指令地址
N
没有命中，则认为本指令不是分支，按普通指令执行

# BTB表工作过程




**问题：解决预测错误或BTB表不命中的延迟。**

# 综合BTB和BHT表的取指令阶段



例：根据下面的假设，计算引入BTB后，分支转移的总延迟。

（1）对于BTB中的指令，预测准确率90%

（2）BTB 缓冲区命中率90%

（3）不在BTB中分支转移成功的比例为60%

| 指令在BTB中？ | 预测结果 | 实际的分支 | 延迟周期 |
|---|---|---|---|
| 是 | 成功 | 成功 | 0 |
| 是 | 成功 | 不成功 | 3 |
| 不是 | | 成功 | 3 |
| 不是 | | 不成功 | 0 |

采用BTB技术时指令在各种情况下的转移延迟

包括4个部分：

1）在BTB中，预测转移，实际成功转移，延迟为0。

2）在BTB中，预测转移，实际转移不成功，延迟为：

BTB命中率 × 预测错误率 × 2

= 90% × 10% × 3

= 0.27（时钟周期）

3）不在BTB中，实际转移成功，延迟为：

（1-BTB命中率） × 不在BTB中分支转移成功率 × 3

= 10% × 60% × 3

= 0.18（时钟周期）

4）不在BTB中，实际转移也不成功，延迟为0。

# 小结

- 控制冒险
- 控制冒险的解决方案
  - 停顿 (对性能影响大)
  - 转移决策尽量早开始
  - 转移延迟槽(需要编译器支持)
  - 转移预测：
    - 静态：
    - 动态：用存储空间换取动态转移预测的正确性
    - 转移预测进一步的工作，已经成熟应用于现代处理器

# 谢谢！