计算机组成
(2022秋)

计算机组成课程组
（刘旭东、高小鹏、肖利民、栾钟治、万寒）
北京航空航天大学计算机学院中德所
栾钟治

➢1

习题5——单周期处理器

- 已发布
  - Spoc平台
- 11月18日截止
  - 23:55
- 在sopc提交
  - 电子版，可手写

➢2

习题6——流水线处理器设计

- 今晚发布
  - Spoc平台
- 12月02日截止
  - 23:55
- 在sopc提交
  - 电子版，可手写

➢3

回顾：load导致的数据冒险

- 靠旁路/转发不能解决所有问题
  - 必须暂停依赖于load的指令
  - 硬件暂停流水线：“硬件互锁”，相当于插入nop
- 如何插入NOP指令？
  - 检测条件：IF/ID的前序是lw指令，并且lw的rt寄存器与IF/ID的rs或rt相同
  - 执行动作：
    - ①冻结IF/ID：sub继续被保存
    - ②清除ID/EX：指令全为0，等价于插入NOP
    - ③禁止PC：防止PC继续计数，PC应保持PC+8
- Load之后的时间片段称为load延迟时隙（slot）
  - 如果后续指令会用到load回来的结果，硬件互锁机制会暂停该指令一个时钟周期
  - 在延迟时隙由硬件暂停指令等价于在该时隙加nop (除非加nop占用更多的代码空间)
- Idea: 让编译器在该时隙插入一条不相关的指令 → 无需暂停!

➢4

回顾：控制冒险

- 执行流依赖于之前的指令
  - 数据冒险的特例：有关指令指针/程序计数器的数据相关
- 下一个周期从PC里取出来的是什么？
  - 所有的指令都和他们之前的指令存在控制相关，关于下一条指令的PC值
- 如果取到的指令不是一个控制指令：
  - 下一次取的PC是下一条顺序执行的指令
  - 只要我们知道取到的指令尺寸就行了
- 如果取到的指令是控制指令：
  - 我们如何决定下一个要取的PC？
- 怎么知道取的指令是不是一个控制指令？

回顾：处理控制冒险

- 关键在于使流水线保持充满正确的动态指令序列
- 当指令是控制指令时可能的解决方案有：
  - 停顿流水线直到得到下一条指令的取指地址
  - 猜测下一条指令的取指地址（分支预测）
  - 采用延迟分支（分支延迟槽/时隙）
  - 其它(细粒度多线程)
  - 消除控制指令（推断执行）
  - 从所有可能的方向取指（如果知道的话）（多路径执行）
- 简单的解决方案：暂停每一个分支直到获得新的PC值
  - 我们必须暂停多长时间？

回顾：如何比暂停流水线更好——预测分支不发生

- 与其等待关于PC的真相关被解决再行动，不如猜测下一个PC = PC+4，保持每个周期都取指
- *分支预测* – 猜测分支的结果，如果出错需要事后修正
  - 必须取消(*flush*)流水线中所有基于错误猜测执行的指令
  - 最终会取消多少指令？
- 如果预测所有的分支都不执行将得到最简单的硬件

  猜测下一个PC = PC + 4

回顾: 预测分支不发生

- 总是预测下一条按顺序的指令就是下一条要被执行的指令
- 下一条取指地址和分支的预测方式
- 如何能让这种方式更有效率？
- 思路1：使下一条按顺序的指令就是下一条要执行的指令的可能性最大
  - 软件vs. 硬件
- 思路2：去掉控制流指令（或者尽量减少它的发生）
  - 软件vs. 硬件

回顾：性能分析

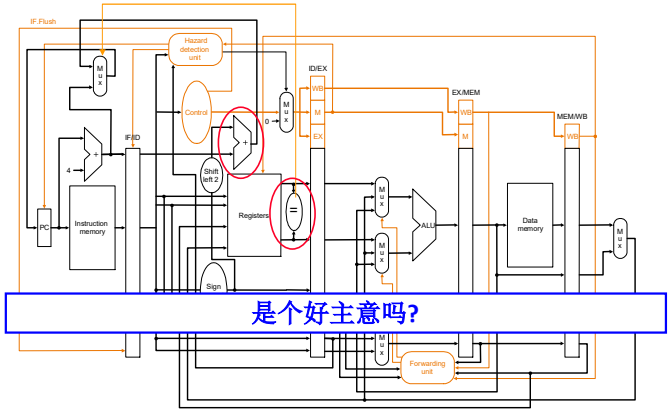
- 大约20%的指令组合是控制流，一般约70%被执行，30%不执行
  - “下一个PC = PC+4” 的期望在~86% 的时间里是对的，但是剩下那14%呢?
- 猜测正确 ⇒ 没有惩罚　约86% 的时间
- 猜测不正确 ⇒ N(比如2)个气泡
- 假设
  - 没有数据相关，20% 的控制流指令，70% 的控制流指令发生转跳
  - CPI = [1 + (0.2*0.7) * 2]
    = [1 + 0.14 * 2] = 1.28

发生错误猜测的可能性　　错误猜测的惩罚

我们有可能减小这两者中的任何一个吗?

回顾：减小分支预测错误的代价——缩短分支延迟

- 提前处理分支条件和获得目标地址（分支判断提前）



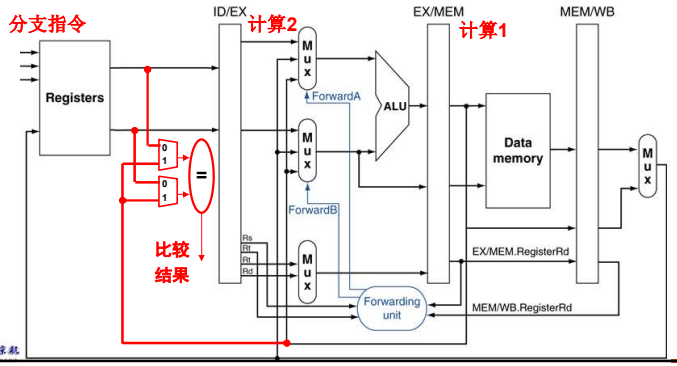
CPI = [1 + (0.2*0.7) * 1] = 1.14

回顾：缩短分支延迟

- 比较器前置后，会产生数据相关
  - 分支指令可能依赖于前条指令的结果
- 依赖计算1：从ALU转发数据
- 依赖计算2：只能暂停

Q：如果依赖MEM/WB的结果，是否需要设置转发?

提示：MEM/WB已经有回写通道了，但RF设计满足吗?

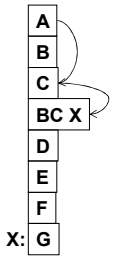


方案3:分支延迟槽

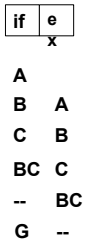
- 改变分支指令的语义
  - N条指令后分支
  - N个周期后分支
- 思路：延迟分支的执行，无论分支的方向如何，总是执行分支指令后紧跟的N条指令（延迟槽）
- 问题：如何找到指令填充延迟槽?
  - 分支必须与延迟槽指令相互独立
- 无条件分支：更容易找到填充延迟槽的指令
- 条件分支：条件计算不应依赖于延迟槽中的指令 → 填充延迟槽有难度

分支延迟(II)

正常的代码:

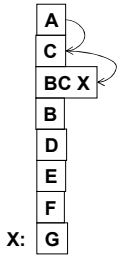


时间线:

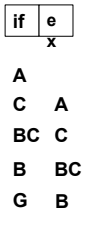


6 个周期

分支延迟的代码:



时间线:



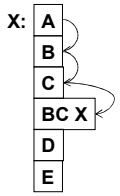
5 个周期

➢13

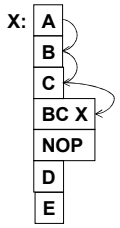
更有想象力的分支延迟(III)

- 带“挤压”的延迟分支
  - SPARC
  - 如果分支不发生，不执行延迟槽中的指令
  - 为什么这样会有好处?

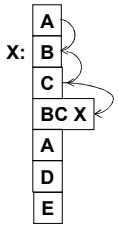
正常代码:



分支延迟代码:



带“挤压”的分支延迟:



➢14

分支延迟(IV)

- 好处:
  - + 使流水线在一种简单的假设下保持充满有用的指令
  1. 延迟槽的数量 == 分支解决之前保持流水线充满的指令条数
  2. 所有的延迟槽可以用有用的指令填充
- 坏处:
  - -- 填充延迟槽不那么容易（即使是2阶段流水线）
  1. 随着流水线深度或者超标量执行的宽度的增加，延迟槽的数量也会增加
  2. 延迟槽的数量会随着操作延迟的变化而变化。为什么?
  - -- ISA 语义与硬件实现的关系
    - -- SPARC, MIPS, HP-PA: 1 个延迟槽
    - -- 如果下一个设计中流水线的实现发生变化了会怎么样?

➢15

延迟分支的例子

无延迟分支

```
or   $8, $9, $10
add  $1, $2, $3
sub  $4, $5, $6
beq  $1, $4, Exit
xor  $10, $1, $11
Exit:
```

延迟分支

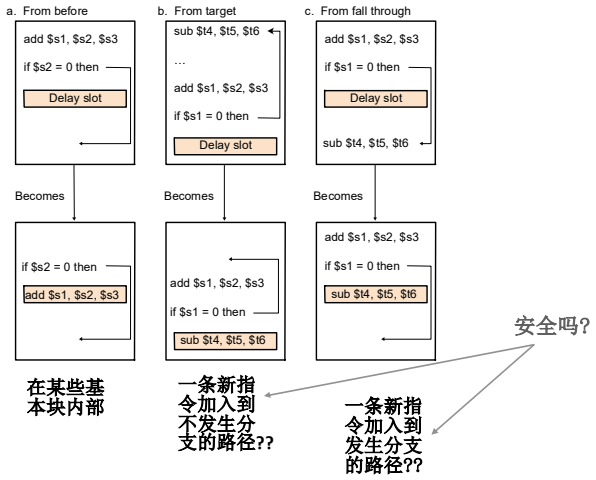
```
add  $1, $2,$3
sub  $4, $5, $6
beq  $1, $4, Exit
or   $8, $9, $10
xor  $10, $1, $11
Exit:
```

为什么不是其它的指令?

➢16

填充延迟槽

重排序数据相关(RAW, WAW, WAR)指令不能够改变程序语义



➢17

分支延迟槽

- MIPS 使用延迟分支这一概念
  - 对指令重排序是加速程序执行的常用方法
  - 编译器选择一条指令放入分支延迟时隙执行大约能节省50%的时间
- Jump指令也有延迟时隙
  - 为什么需要?
- MIPS Green Sheet

```
jal:
R[31]=PC+8;  PC=JumpAddr
```

  - PC+8 就是因为*jump 延迟时隙*!
  - PC+4所指的指令总是在jal转跳向label之前被执行，所以返回PC+8

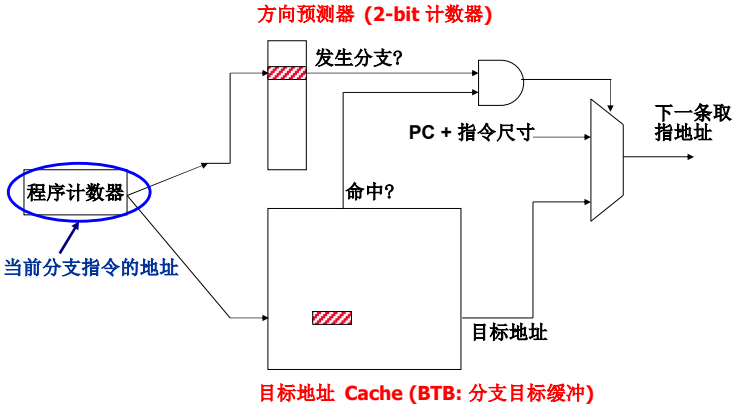
➢18

分支预测(增强版)

- 思路: 预测下一个取指地址（下一个周期会用到）
- 需要在取指阶段预测三件事:
  - 取到的指令是不是一个分支指令
  - (条件) 分支的方向
  - 分支的目标地址（如果分支发生）
- 观察:不同动态实例的条件分支目标地址可能是相同的
  - 思路: 存储以前实例的目标地址，由PC访问它
  - 被称作分支目标缓冲 (BTB) 或者分支目标地址 Cache

➢19

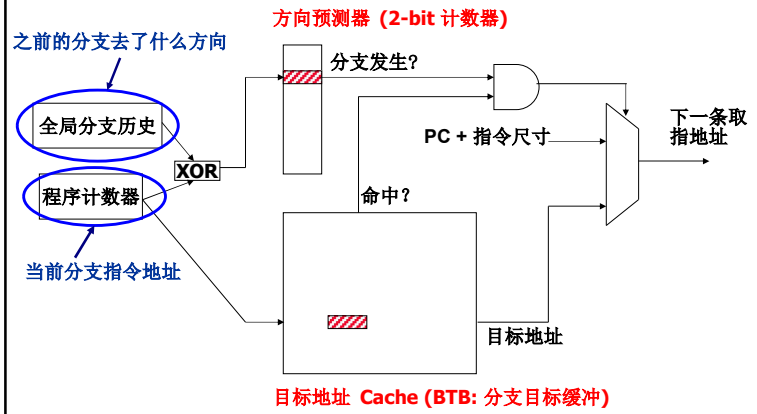
有BTB的取指和方向预测



总是发生的 CPI = [1 + (0.2*0.3) * 2] = 1.12 (70% 的分支会发生)

➢20

更复杂的分支方向预测



简单的分支方向预测方案

- ❖编译时（静态）
  - ➢总是不发生
  - ➢总是发生
  - ➢BTFN（反向发生，正向不发生）
  - ➢基于分析（可能的方向）
- ❖运行时（动态）
  - ➢Last time 预测（1-bit）

更复杂的方向预测

- ❖编译时（静态）
  - ➢总是不发生
  - ➢总是发生
  - ➢BTFN（反向发生，正向不发生）
  - ➢基于剖析（可能的方向）
  - ➢基于程序分析（可能的方向）
- ❖运行时（动态）
  - ➢Last time 预测（1-bit）
  - ➢基于2-bit计数器的预测
  - ➢两层预测(全局vs.局部)
  - ➢混合

静态分支预测(I)

- ❖总是不发生
  - ➢实现简单：不需要BTB，不需要方向预测
  - ➢准确率低：~30-40%
  - ➢编译器可以重新布局代码，这样能够使可能的路径就是“不发生分支”的路径
- ❖总是发生
  - ➢无方向预测
  - ➢更好的准确率：~60-70%
    - 反向分支（ loop分支）通常会发生
    - 反向分支：目标地址比分支指令PC值小
- ❖反向发生，正向不发生（BTFN）
  - ➢预测反向(loop)分支总是发生，其他的不发生

静态分支预测(II)

❖基于剖析（profiling）

➢思路：编译器通过运行分析代码为每个分支决定可能的方向，分支指令格式编码增加一个提示位表示分支方向

+ 逐个分支预测（比前面讲到的方式更准确）→ 如果 分析代码有代表性就有准确率!
- 需要在分支指令格式中加提示位
- 准确性依赖于分支的动态行为:
  TTTTTTTTTTNNNNNNNNNN → 50% 准确率
  TNTNTNTNTNTNTNTNTNTN → 50% 准确率
- 准确与否依赖于分析代码的输入数据集的典型性

静态分支预测(III)

❖基于程序（或者基于程序分析）

➢思路：使用基于程序分析的启发式方法来确定静态预测的分支方向

➢操作码启发式：预测 BLEZ 不发生分支（很多程序中用负整数代表错误值）

➢循环启发式：预测一个分支控制的循环操作会执行分支（执行循环）

➢指针的比较和浮点数的比较：预测不相等

+ 不需要剖析
- 启发式方法可能不具有代表性或者不够好
- 需要编译器分析和ISA支持

静态分支预测(III)

❖基于程序员

➢思路：程序员提供静态预测的方向

➢通过编程语言中的Pragma使分支成为可能发生或可能不发生的分支

+ 不需要剖析或程序分析
+ 相比那些分析技术来说，程序员可能对程序或分支更了解
- 需要编程语言、编译器和ISA支持
- 增加程序员的负担？

Pragma

❖思路：使程序员可以向更低层次的转换转达一些提示的关键词

➢if (likely(x)) { ... }

➢if (unlikely(error)) { ... }

❖很多提示和优化可以通过pragma实现

➢例如，一个循环是否可以并行化：#pragma omp parallel

描述：一个OpenMP并行指令，显式地指示编译器对选定的代码段进行并行化

静态分支预测

❖所有前面讲到的技术都可以组合

➢基于剖析（profile）

➢基于程序（program）

➢基于程序员（programmer）

❖这三种技术有共同的缺陷

➢不能适应分支行为的动态变化

动态分支预测

❖思路：基于动态信息预测分支（运行时采集信息）

❖好处
- + 基于分支执行的历史预测
  - + 可以适应分支行为的动态变化
- + 无需剖析：输入集的典型性问题不复存在

❖坏处
- – 更加复杂（需要额外的硬件）

Last Time 预测器

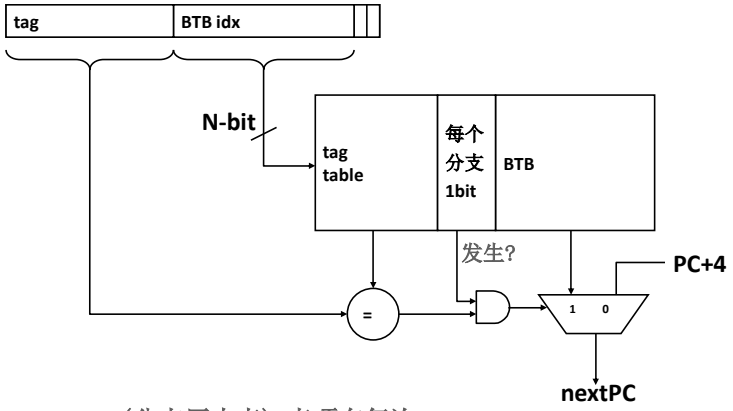
❖Last time 预测器
- ➢每个分支1bit（存在BTB中）
- ➢显示上一次分支执行时的方向
  TTTTTTTTTTNNNNNNNNNN → 90% 准确率

❖对于循环分支总是预测错第一次和最后一次迭代
- ➢对于N次迭代循环的准确率 = (N-2)/N

+ 有大量迭代的循环分支
– 只有少量迭代的循环分支
TNTNTNTNTNTNTNTNTNTN → 0% 准确率

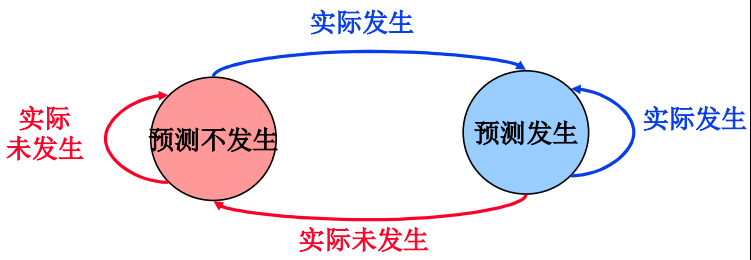
Last-time 预测器 CPI = [1 + (0.20*0.15) * 2] = 1.06 （假设 85% 准确率）

实现Last-Time 预测器



1-bit BHT（分支历史表）表项在每次执行完一个分支后更新正确的结果

Last-Time 预测的状态机



改进的Last Time 预测器

- 问题：last-time 预测器改变预测太快（T→NT或者NT→T）
  - 即使分支可能大部分发生或者大部分不发生
- 解决思路：为预测器增加滞后效果，让预测不要因为出现1次不同的结果就改变
  - 使用2bits而不是1bit跟踪分支预测的历史
  - T或者NT可以分别有2个状态

➢33

基于2-Bit计数器的预测

- 每个分支关联一个2-bit计数器
- 增加的1bit提供一个“滞后”
- 强预测不会因为一次不同结果而改变

- N次迭代循环的准确率= (N-1)/N
  TNTNTNTNTNTNTNTNTNTN → 50% 准确率
  （假设初始时弱发生）

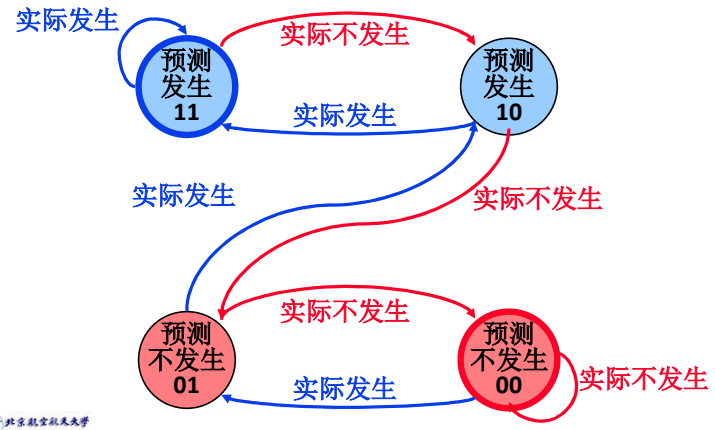
\+ 更好的预测准确率
\- 更多的硬件开销（但是计数器可以是BTB表项的一部分）

2BC 预测器 CPI = [1 + (0.20*0.10) * 2] = 1.04 (90% 准确率)

➢34

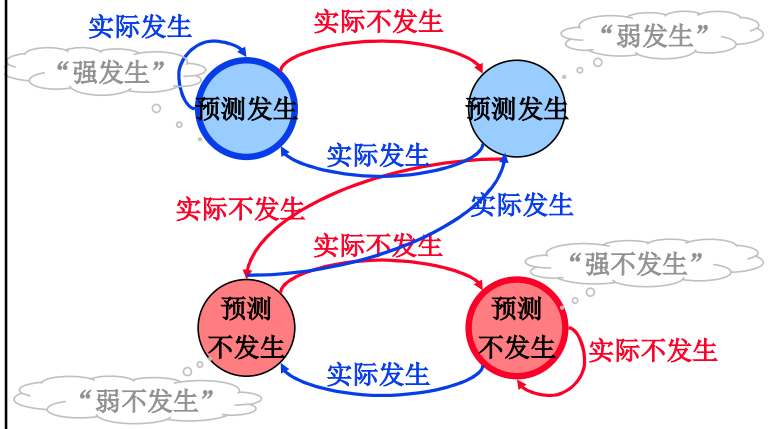
2-bit 饱和计数器的状态机

- 计数器使用饱和算术
  - 有一个代表最大值和最小值的符号



➢35

使用2-bit计数器产生的滞后效应



2次连续错误后改变预测

➢36

够好了吗?

- 很多基于2-bit预测的程序有~85-90% 的准确率（也叫做双模态预测）
- 这样足够好了吗?
- 分支的问题有多大?

➢37

重新思考分支问题

- 控制流指令(分支）很常见
  - 占所有指令的15-25%
- 问题：控制流指令之后的下一个取指地址在流水线处理器中会在N个周期后仍难以确定
  - N个周期：（最小）分支解决延迟
  - 分支时停顿浪费指令处理带宽（降低IPC）
    - N x IW 个指令槽被浪费（IW：发射宽度）
- 如何在分支之后保持流水线充满?
- 问题：需要在分支指令被取出时决定下一个取指地址（避免流水线气泡）

➢38

分支问题的重要性

- 假设一个5发射宽度的超标量流水线有20个周期的分支解决时延
- 取500条指令要花费多长时间?
  - 假设连续取指，并且5条指令中有1条是分支
  - 100% 准确率
    - 100 个周期（获取的所有指令都在正确的路径）
    - 没有做无用功
  - 99% 准确率
    - 100（正确路径）+ 20（错误路径）= 120 个周期
    - 20% 额外的取指
  - 98% 准确率
    - 100（正确路径）+ 20 * 2（错误路径）= 140 个周期
    - 40% 额外的取指
  - 95% 准确率
    - 100（正确路径）+ 20 * 5（错误路径）= 200个周期
    - 100%额外的取指

➢39

能不能做的更好?

- Last-time和2BC预测器利用“last-time”可预测性
- 认识1：一个分支的结果可能和其它分支的结果相关
  - 全局分支相关
- 认识2：一个分支的结果可能和同一个分支过去的结果相关（不仅仅是上一次分支执行的结果）
  - 本地分支相关

➢40

全局分支相关

❖最近一个执行分支的结果与下一个分支结果相关

❖如果第一个分支不发生，第二个也不会发生

```
if (cond1)
...
if (cond1 AND cond2)
```

❖如果第一个发生了，第二个肯定不会发生

```
branch Y: if (cond1) a = 2;
...
branch X: if (a == 0)
```

❖如果 Y 和 Z 都发生， X 也发生

❖如果 Y 或 Z 不发生， X 也不发生

```
branch Y: if (cond1)
...
branch Z: if (cond2)
...
branch X: if (cond1 AND cond2)
```

❖Eqntott, SPEC 1992

```
if (aa==2)    ;; B1
                aa=0;
    if (bb==2)    ;; B2
                bb=0;
    if (aa!=bb) {  ;; B3
                ....
                }
```

如果 B1 不发生（aa=0@B3）并且 B2 不发生（bb=0@B3），则 B3 肯定不发生

捕获全局分支相关

❖思路：将分支结果与所有分支的“全局T/NT历史”关联

❖根据上一次相同全局分支历史的分支结果作出预测

❖实现：

- ➢用一个寄存器跟踪所有分支的“全局T/NT历史”➔全局历史寄存器(GHR)
- ➢使用GHR索引到一张表，表中记录了最近的过去与GHR中值相应的分支的结果 ➔ 模式历史表(2-bit计数器表)

❖全局历史/分支预测器

❖使用两个层次的历史(GHR+GHR的历史)

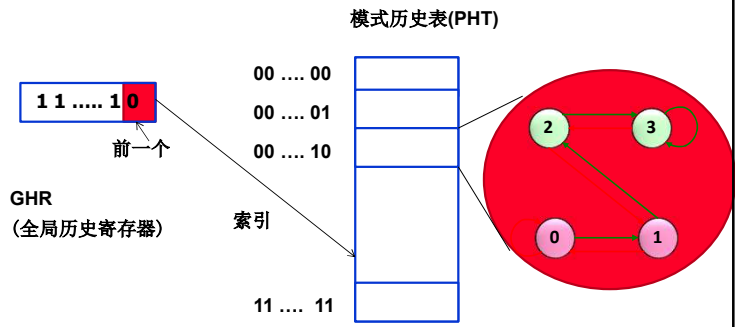
两层全局分支预测

❖ 第一层：全局分支历史寄存器 (N bits)

- ➢前N次分支的方向

❖ 第二层：每个历史表项的饱和计数器表

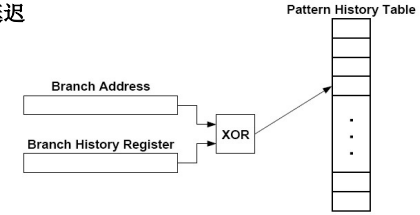
- ➢上一次相同的历史情况下的分支方向



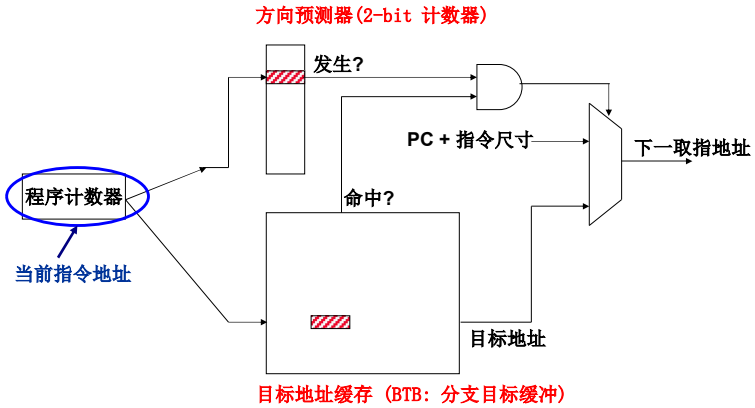
改进全局分支预测的准确性

❖ 思路：为全局预测器增加更多的上下文信息来决定预测哪一个分支

- ➢Gshare预测器：GHR按分支地址哈希
- \+ 更多的上下文信息
- \+ 更好地利用模式历史表
- \- 增加访问延迟

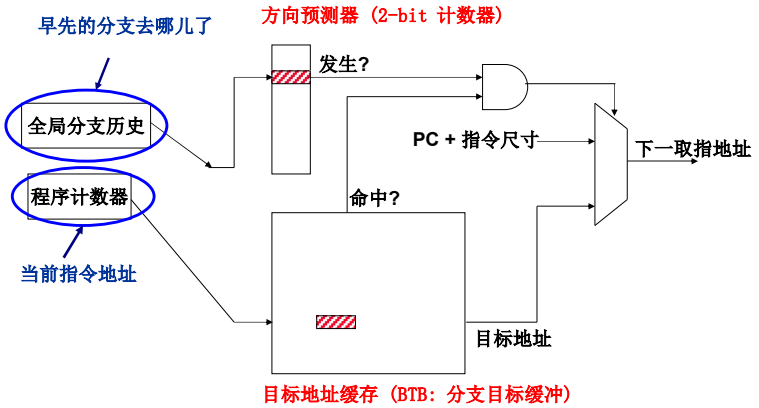


一层分支预测器



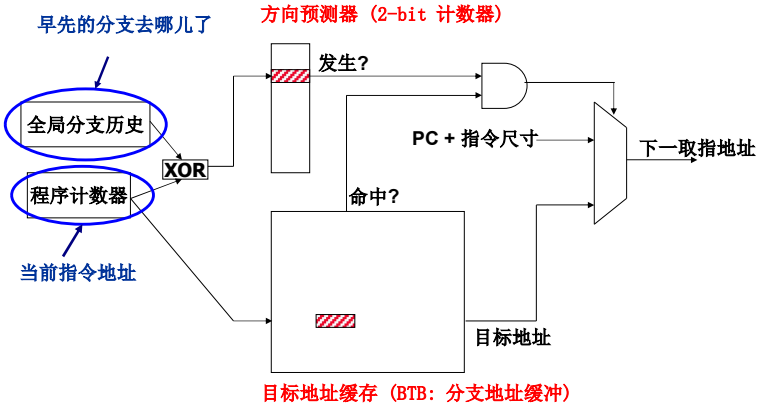
➢45

两层全局历史预测器



➢46

两层Gshare预测器



➢47

还能更好吗?

- Last-time和2BC预测器利用“last-time”可预测性
- 认识1: 一个分支的结果可能和其它分支的结果相关
  - 全局分支相关
- 认识2: 一个分支的结果可能和同一个分支过去的结果相关(不仅仅是跟“last-time”分支执行的结果)
  - 本地分支相关

```
for (i=0; i<=4; i++) {}
(1110)^n
```

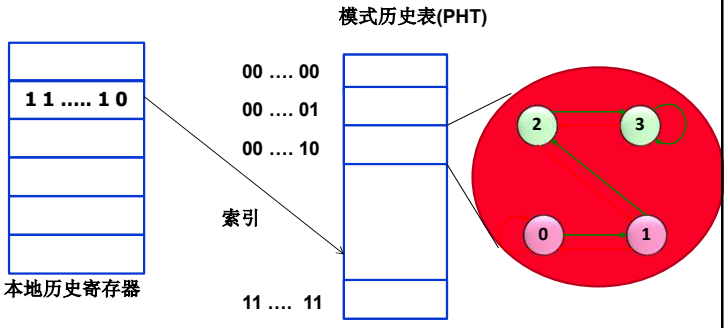

➢48

捕获本地分支的关联性

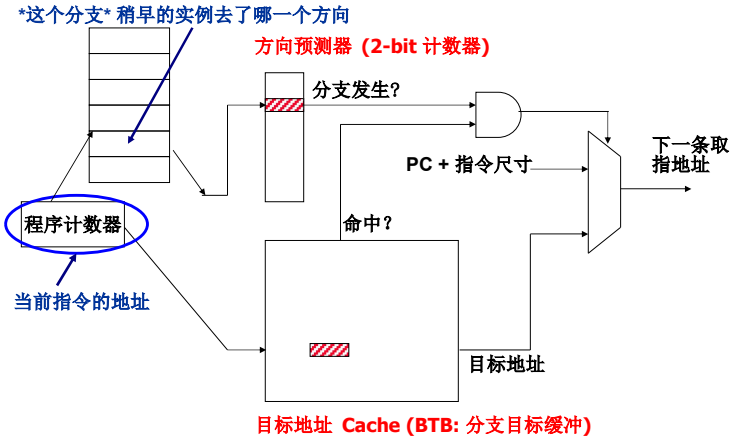
- 思路：每个分支都有一个历史寄存器
  - 将分支预测结果与该分支在“历史上发生/不发生”关联
- 根据上一次相同本地分支历史的分支结果作出预测
- 称为本地历史/分支预测器
- 使用两个层次的历史（每个分支历史寄存器 + 取那一个历史寄存器值的历史）

两层本地历史预测器

- 第一层：一组本地历史寄存器（每个N bits）
  - 选择基于分支指令地址的历史寄存器
- 第二层：每一个历史条目的饱和计数器表
  - 上一次相同的历史情况下的分支方向



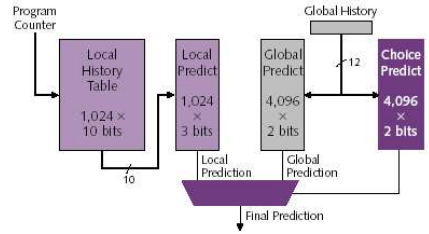
两层本地历史预测器



混合分支预测器

- 思路：使用不止一种类型的预测器（采用多种算法），选择“最佳”的预测
  - 比如，2-bit 计数器和全局预测器的混合
- 好处：
  - \+ 更好的准确率：不同的预测器更适用不同的分支
  - \+ 减少“热身”时间（先使用“进入状态”快的预测器，直到“慢热”的预测器热身完毕）
- 坏处：
  - – 需要“元预测器”或“选择器”
  - – 更长的访问时延

Alpha 21264 锦标赛预测器（混合预测）



- 最小的分支惩罚：7 cycles
- 典型的分支惩罚：11+ cycles
- 48K bits 的目标地址保存在 I-cache
- 预测器表在上下文切换时重置

➢53

分支预测准确率(示例)

- 双模态：由分支地址索引的2bc表

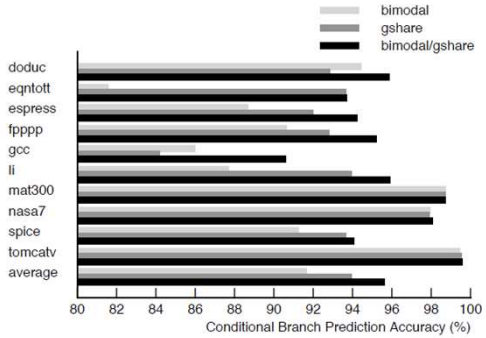


Figure 13: Combined Predictor Performance by Benchmark

➢54

有偏向性的分支

- 观察：很多分支会偏向某一个方向（比如，99% 发生）
- 问题：这些分支破坏了分支预测的结构 → 给分支预测表和历史寄存器造成“干扰”，使得对其它类型的分支预测变得困难
- 解决方案：检测这样的有偏向性的分支，用更简单的预测器预测它们

➢55

回顾: 分支类型

类型	取指阶段能判断的分支方向	下一个可能地址的数量?	何时能够解析出下一个取指的地址?
条件分支	不知道	2	执行 (寄存器相关)
无条件分支	总是发生转跳	1	译码 (PC + offset)
调用	总是发生转跳	1	译码 (PC + offset)
返回	总是发生转跳	多	执行 (寄存器相关)
间接分支	总是发生转跳	多	执行 (寄存器相关)

不同类型的分支处理方式不同

➢56

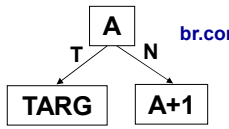
调用和返回预测

- 直接调用容易预测
  - 总是发生，单个目的地址
  - 调用在BTB中标记, 由BTB预测目的地址
- 返回是间接分支
  - 函数可以由代码中的多个点调用
  - 返回指令可能有多个目的地址
    - 相同函数的每一个调用点的下一条指令
  - 观察: 通常每个返回对应一个调用
  - 思路: 使用栈来预测返回地址 (返回地址栈)
    - 取到调用指令: 返回地址（下一条指令）压入堆栈
    - 取到返回指令: 弹出堆栈，使用该地址作为预测的目的地址
    - 大部分时间准确: 8-entry栈→ > 95% 准确率

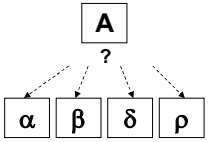
```
Call X
  …
Call X
  …
Call X
  …
Return
Return
Return
```


间接分支预测(I)

- 寄存器间接分支有多个目标地址



br.cond TARGET



R1 = MEM[R2]
branch R1

条件（直接）分支　　间接转跳

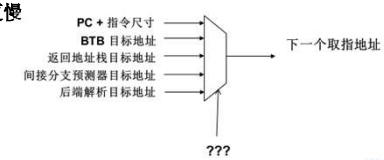
- 用来实现
  - Switch-case 语句
  - 虚函数调用
  - 转移表 (函数指针)
  - 接口调用

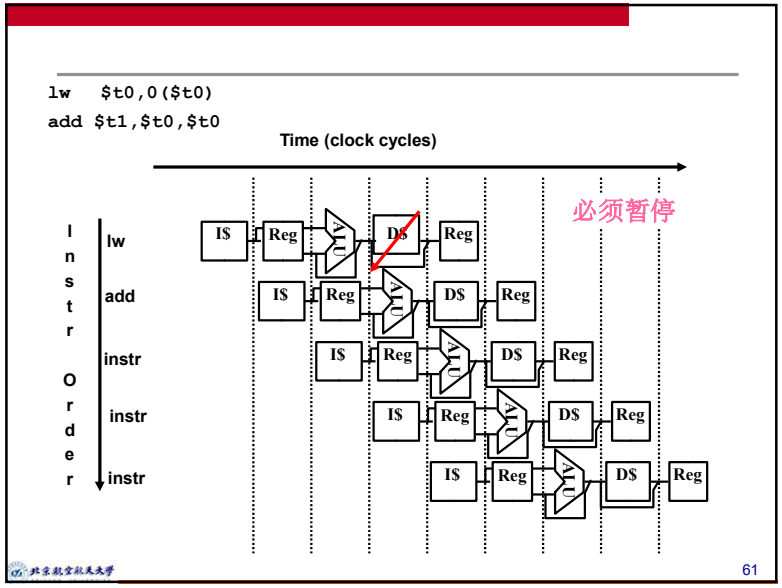
间接分支预测(II)

- 不需要预测方向
- 思路 1: 预测上一次解析的目标就是下一次要取的地址
  - \+ 简单: 使用BTB 存储目标地址
  - \- 不准: 50% 准确率 (经验数据). 很多间接分支会在不同的目标之间切换
- 思路 2: 基于历史的目标预测
  - 比如, 用GHR XOR 间接分支PC来索引BTB
  - \+ 更准确
  - \- 一个间接分支会映射到（可能很）多个BTB表项
    - \- 会与其他分支发生Conflict miss (直接或间接)
    - \- 在分支只有极少的目标地址的情况下，空间利用率低

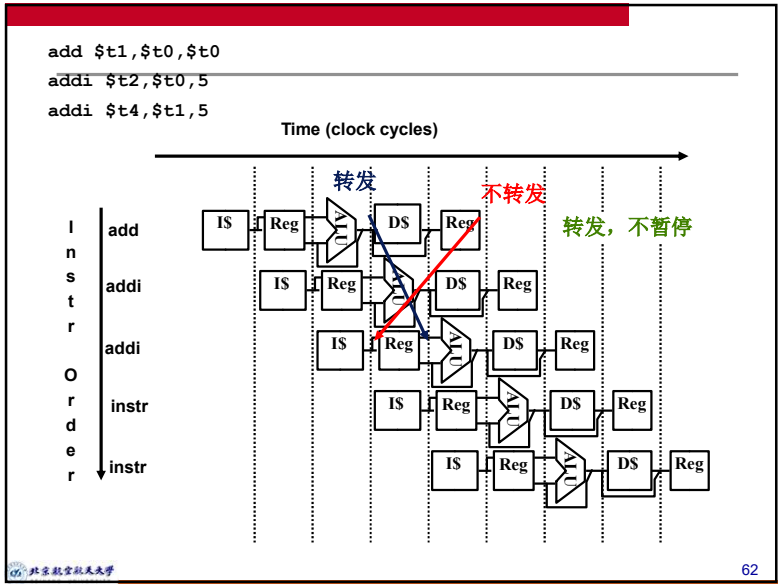
分支预测中的问题

- 需要在取指结束之前识别出分支
- 如何做到?
  - BTB 命中 → 说明取的指令是一个分支
  - BTB 表项包含一个分支的“类型”
- 如果没有BTB怎么办?
  - 在流水线中加气泡直到目标地址计算出来
  - 比如, IBM POWER4
- 时延: 时延对预测很关键
  - 需要为下一个周期产生取指地址
  - 更大更复杂的预测器更准确但是更慢

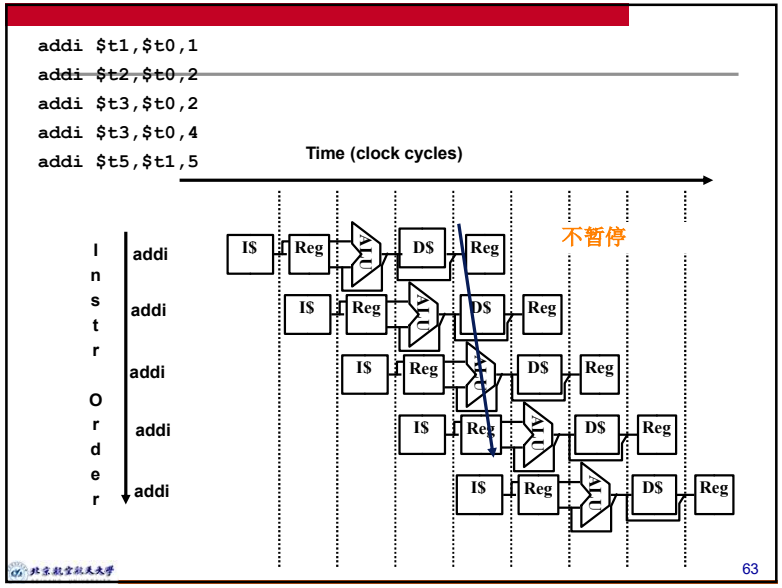




➢61



➢62



➢63

- 冒险降低了流水的效率
  - 导致暂停/气泡
- 结构冒险
  - 数据通路部件使用的冲突
- 数据冒险
  - 需要等待之前指令的结果
- 控制冒险
  - 下一条指令地址的不确定性
  - 分支和转跳延迟槽

➢64

处理器设计总结

- ISA和微体系结构
- 单周期处理器
- 多周期处理器
- 流水线处理器

➢65

计算机性能评价

- 响应时间与吞吐量
  - 响应时间：从提交作业到完成作业所花费的时间
    - 响应时间是完成一个任务所花的时间总和，包括内存访问时间、执行IO操作的时间、以及运行必要的操作系统代码所需的时间。
  - 吞吐量：一定时间间隔内完成的作业数
    - 多任务操作系统更侧重于优化系统的整体吞吐量，而不会特别最小化某个特定程序的响应时间
  - 个人用户更关心响应时间，企业级计算机的管理人员更关心吞吐量
  - 对于企业级计算机以外的应用，响应时间是评价计算机性能的主要依据

➢66

计算机性能评价

- 响应时间与CPU执行时间
  - 对于多任务系统，应该从响应时间中去除因为等待I/O操作而花去的时间和CPU执行其他程序所花费的时间，为此引入CPU执行时间的概念。
  - CPU执行时间是CPU真正花在运行一个程序上的时间。

程序的CPU执行时间

＝程序的CPU时钟周期数 × 时钟周期长度

$$= \frac{\text{程序的CPU时钟周期数}}{\text{时钟频率}}$$

程序的CPU时钟周期数

＝程序的指令数×每条指令的平均时钟周期数

➢67

计算机性能评价

- CPI：指令平均执行时钟周期数
  - CPI：Clock cycles Per Instruction。
  - 不同指令功能不同，所需时间也不同，CPI只是某一机器中一个程序或程序片段每条指令所用时钟周期的平均值。
  - 不同指令集的CPI比较没有实际意义。

CPU 时间=CPU 时钟周期数/频率；

CPU 时间=CPU 时钟周期数*时钟周期长；

平均时钟周期数 CPI=CPU 时钟周期数/IC（指令的条数）；

CPU 时间=（IC*CPI）/频率 f；

$$\text{CPU 的时钟周期数} = \sum_{i=1}^{n}(CPI_i \times I_i)$$

$$CPI = \frac{\sum_{i=1}^{n}(CPI_i \times I_i)}{IC} = \sum_{i=1}^{n}(CPI_i \times \frac{I_i}{IC})$$

➢68

计算机性能评价

❖MIPS：百万指令每秒

- MIPS：Million Instruction Per Second
- 不同指令集的MIPS比较没有实际意义
- 即使同一台机器，用不同的测试程序测出来的MIPS值也可能不一样。

$$\text{MIPS} = \frac{\text{指令条数}}{\text{执行时间} * 10^6} = \frac{f}{CPI * 10^6}$$

❖MFLOPS：百万浮点数操作每秒

- MFLOPS： Million Floating point Operations Per Second
- 可以比较不同机器的浮点运算能力，但有局限性
- MFLOPS不仅和机器有关，也和所用测试程序有关
- MFLOPS与整数、浮点操作的比例有关

$$\text{MFLOPS} = \frac{\text{程序中的浮点操作次数}}{\text{执行时间} * 10^6}$$

计算机性能评价

❖影响计算机性能的因素

- 指令数：取决于指令集体系结构(ISA)，与指令集的具体实现无关；编译器对指令数具有很大的影响；
- CPI：机器的实现细节（存储系统结构、处理器结构）；测试程序包含的各类指令的组成等；
- 时钟周期：与机器的实现细节密切相关

影响因素	影响
算法	指令数、CPI
程序设计语言	指令数、CPI
编译器	指令数、CPI
ISA	指令数、CPI、时钟周期
硬件实现	CPI、时钟周期

计算机性能评价

❖示例一

假设在一台 40MHZ 处理机上运行 200,000 条指令的目标代码，程序主要由四种指令组成。根据程序跟踪实验结果，已知指令混合比和每种指令所需的指令数如下。计算在单处理机上用跟踪数据运行程序的平均 CPI，并根据所得的 CPI，计算相应的 MIPS 速率。

指令类型	CPI	指令混合比
算术和逻辑	1	60%
高速缓存命中的加载/存储	2	18%
转移	4	12%
高速存储缺失的存储器访问	8	10%

[解]

CPI = 1*60% + 2*18% + 4*12% + 8*10% = 2.24

MIPS = f/(CPI*10^6) = (40*10^6)/(2.24*10^6) = 17.86

计算机性能评价

❖示例二

对于一台 400MHz 计算机执行标准测试程序，程序中指令类型，执行数量和平均时钟周期数如下：

指令类型	指令执行数量	平均时钟周期数
整数	45000	1
数据传送	75000	2
浮点	8000	4
分支	1500	2

求该计算机的有效 CPI、MIPS 和程序执行时间。

解：$CPI = \sum (IC_i \times CPI_i) / IC$

$$CPI = \frac{45000\times1 + 75000\times2 + 8000\times4 + 1500\times2}{45000 + 75000 + 8000 + 1500} = 1.776$$

$$MIPS\text{速率} = \frac{f}{CPI} = \frac{400\times10^6}{1.776} = 225.225MIPS$$

程序执行时间=

$$(45000\times1 + 75000\times2 + 8000\times4 + 1500\times2)/(400\times10^6) = 5.75\times10^{-4}\,\text{s}$$