高等计算机体系结构
第七讲: 流水线和控制相关（II）

栾钟治
北京航空航天大学 计算机学院 中德联合软件研究所
2020-04-17

1

提醒：作业

- 作业 3
  - 4月10日发布，4月24日截止
  - 流水线1
- 作业 4
  - 4月24日发布，5月8日截止
  - 流水线2

2

2

提醒：实验

- 实验1，今天截止
  - 用Logisim设计1个7指令单周期MIPS CPU
- 实验2-5，待定

3

3

阅读材料

- Patterson & Hennessy's *Computer Organization and Design: The Hardware/Software Interface* （计算机组成与设计：软硬件接口）
  - 第四章 (4.9-4.11)
- 选读
  - Smith and Sohi, "The Microarchitecture of Superscalar Processors," Proceedings of the IEEE, 1995
    - 更高级的流水线
    - 中断和异常处理
    - 乱序和超标量执行的概念
- 推荐阅读
  - McFarling, "Combining Branch Predictors," DEC WRL Technical Report, 1993.

4

4

回顾：流水线设计中的问题

- 流水段的平衡
  - 需要多少段以及每一段完成什么任务
- 有影响流水的事件时，保持流水线正确、顺畅、满负荷
  - 处理相关性（冒险）
    - 数据
    - 控制
  - 处理资源争用
  - 处理长时延（多个周期）操作
- 处理异常、中断
- 更高的要求：提高流水线的吞吐
  - 使停顿最少

5

5

回顾：如何处理数据相关

- 反相关和输出相关更容易处理
  - 在一个阶段中完成写操作并且保证程序序
- 五种处理流相关的基本方法
  - 检测并等待直到值在寄存器堆中可以访问
  - 检测并转发/旁路数据给相关的指令
  - 检测并消除相关性（在软件层面）
    - 不需要硬件检测相关性
  - 预测需要的值，“投机”执行，并且验证
  - 其它(细粒度多线程)
    - 不需要检测

6

6

回顾：硬件vs.软件互锁的问题

- 硬件和软件在数据相关处理中各扮演什么角色?
  - 基于软件的互锁
  - 基于硬件的互锁
  - 谁插入/管理流水线气泡?
  - 谁找到独立的指令填充“空闲”的流水线时隙(槽)?
  - 两种方法的优缺点各是什么?

7

7

回顾：硬件vs.软件互锁

- 硬件和软件在指令在流水线中执行的过程中发挥了什么作用?
  - 基于软件的互锁→ 静态调度
  - 基于硬件的互锁→ 动态调度
- 基于软件的指令调度→静态调度
  - 编译器对指令排序，硬件按这个序执行
  - 与之形成对比的是动态调度(硬件不按编译器给定的序执行指令)
  - 编译器怎么知道每条指令的延迟?
- 编译器不知道哪些信息使得静态调度很困难?
  - 答案: 所有在运行时（run time）决定的东西
    - 可变的操作延迟, 内存的地址, 分支的方向
- 编译器如何缓解这些困难(如何估计这些未知量)?
  - 答案: Profiling

8

8

回顾：控制相关

■问题: 下一个周期从PC里取出来的是什么?

■答案: 下一条指令的地址

❑所有的指令都和他们之前的指令存在控制相关。为什么？

■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?

■实际上，我们怎么知道取的指令是不是一个控制指令?

9

9

回顾：分支的类型

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

10

10

回顾：如何处理控制相关

• 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）

11

11

回顾：如何比停止取指更好...

• 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

这是好的猜测吗?

如果猜错了会损失什么？

• ~20% 的指令组合是控制流

- ~50 % 的“向前”控制流 (比如 if-then-else) 被执行
- ~90% 的“向后”控制流 (比如 loop) 被执行

总的来说，一般 ~70% 被执行， ~30% 不会执行

[Lee and Smith, 1984]

• “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?

12

12

回顾：猜测下一个PC = PC + 4

- 如何能让这种方式更有效率?
- 思路：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：制定控制流图，使得“可能的下一条指令”出现在不发生分支的路径上
  - 硬件：???（如何能在硬件中做到这一点…）
- 还能怎样使这种方式更高效?
- 思路：去掉控制流指令（或者尽量减少它的发生）
  1. 去掉不必要的控制流指令 → 组合推断(把条件推断组合起来)
  2. 将控制相关转化为数据相关 → 推断执行

13

13

如何处理控制相关

- 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）

14

14

推断执行

- 思路：将控制相关转化为数据相关
- 假设有一个条件转移指令…
  - CMOV condition, R1 ← R2
  - R1 = (condition == true) ? R2 : R1
  - 在大多数现代 ISA (x86, Alpha)中都有
- 分支 vs. CMOV 指令的代码示例

```
if (a == 5) {b = 4;} else {b = 3;}

CMPEQ condition, a, 5;
CMOV condition, b ← 4;
CMOV !condition, b ← 3;
```

15

15

推断执行

- 消除分支→ 使代码成“直线”推进（形成更大的基本代码块）
- 好处
  - 使预测“不发生分支”的效果更好（没有分支）
  - 编译器有更大的自由度来优化代码（没有分支）
    - 使控制流不妨碍指令的重排序优化
    - 只有数据相关会阻碍代码的优化
- 坏处
  - 无用功：一些指令被取指/执行，但是最终被丢弃（尤其是在容易预测的分支中更糟糕）
  - 需要额外的 ISA 支持
- 可以像这样消除所有的分支吗?

16

16

推断执行

- 选读
  - Allen et al., “Conversion of control dependence to data dependence,” POPL 1983.
  - Kim et al., “Wish Branches: Combining Conditional Branching and Predication for Adaptive Predicated Execution,” MICRO 2005.

17

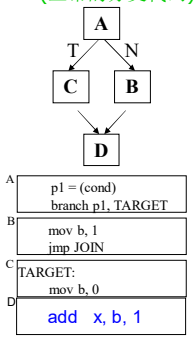
17

推断 (推断执行)

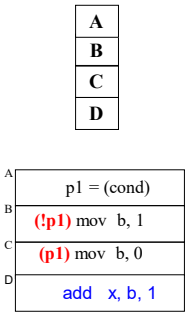
- 思路: 编译器将控制相关转化为数据相关→ 分支被消除了
  - 每条指令根据推断计算的结果置推断位
  - 只有推断结果是“TRUE”的指令被交付 (其他的被转换为 NOP)

```
if (cond) {
    b = 0;
}
else {
    b = 1;
}
```

(正常的分支代码)



(推断代码)

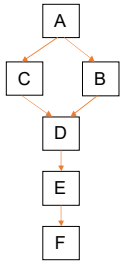


18

18

推断执行(II)

- 推断执行可以具有较高的性能和能效

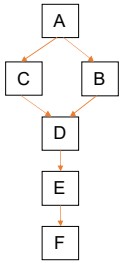


19

19

推断执行(II)

- 推断执行可以具有较高的性能和能效



20

20

推断执行(II)

- 推断执行可以具有较高的性能和能效

推断执行

取指	译码	重命名	调度	读寄存器堆	执行
B	A				

21

21

推断执行(II)

- 推断执行可以具有较高的性能和能效

推断执行

取指	译码	重命名	调度	读寄存器堆	执行
C	B	A			

22

22

推断执行(II)

- 推断执行可以具有较高的性能和能效

推断执行

取指	译码	重命名	调度	读寄存器堆	执行
D	C	B	A		

23

23

推断执行(II)

- 推断执行可以具有较高的性能和能效

推断执行

取指	译码	重命名	调度	读寄存器堆	执行
E	D	C	B	A	

24

24

推断执行(II)

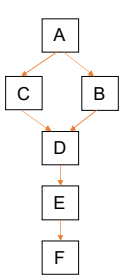
- 推断执行可以具有较高的性能和能效

25

25

推断执行(II)

- 推断执行可以具有较高的性能和能效

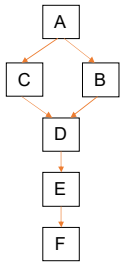


26

26

推断执行(II)

- 推断执行可以具有较高的性能和能效

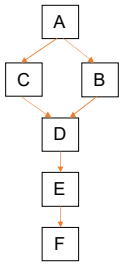


27

27

推断执行(II)

- 推断执行可以具有较高的性能和能效



28

28

推断执行(II)

- 推断执行可以具有较高的性能和能效

29

29

推断执行(II)

- 推断执行可以具有较高的性能和能效

30

30

推断执行(II)

- 推断执行可以具有较高的性能和能效

31

31

推断执行(III)

- 好处:
  - + 避免对难以预测的分支的错误预测
    - + 对某些分支不需要进行分支预测
    - + 如果预测错误的开销>由预测导致的无用功开销，适宜采用推断执行
  - + 可以对受到控制相关制约的代码进行优化
    - + 可以更加自由的移动推断执行的代码

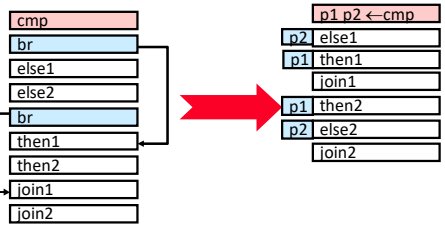
- 坏处:
  - -- 一些容易预测的分支也会导致无用功
    - -- 如果预测错误的开销<无用功开销将使性能下降
    - -- 适应性: 静态的推断不适应运行时分支行为，分支行为的变化与输入、控制流的路径相关
  - -- 需要额外的硬件和ISA的支持
  - -- 无法消除所有的难预测分支
    - -- 循环分支?

32

32

Intel Itanium的推断执行

- 每条指令都可以独立地进行推断
- 64个1-bit 推断寄存器
  每条指令有1个6-bit的推断域
- 一条指令当推断结果是false的时候等效于一条NOP



33

33

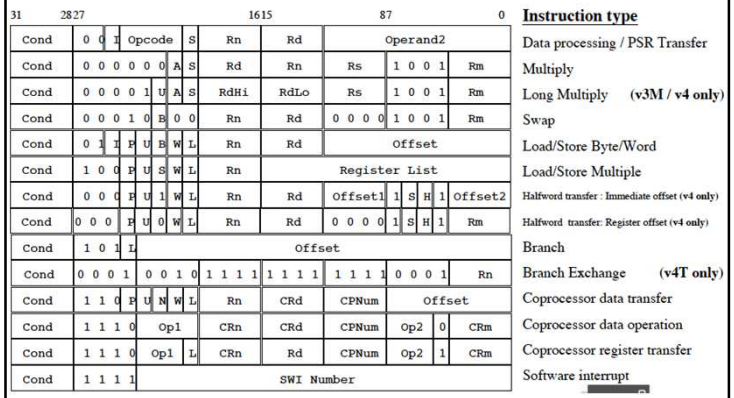
ARM ISA中的条件执行

- 几乎所有的ARM指令都可以包含一个可选的条件码
- 带条件码的指令只有在CPSR（当前程序状态寄存器）中的条件码标志对应到特定的条件时才会执行

34

34

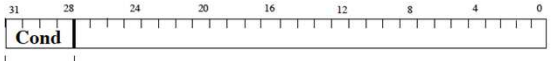
ARM ISA中的条件执行



35

35

ARM ISA中的条件执行



0000 = EQ - Z set (equal)
0001 = NE - Z clear (not equal)
0010 = HS / CS - C set (unsigned higher or same)
0011 = LO / CC - C clear (unsigned lower)
0100 = MI -N set (negative)
0101 = PL - N clear (positive or zero)
0110 = VS - V set (overflow)
0111 = VC - V clear (no overflow)
1000 = HI - C set and Z clear (unsigned higher)
1001 = LS - C clear or Z (set unsigned lower or same)
1010 = GE - N set and V set, or N clear and V clear (>or =)
1011 = LT - N set and V clear, or N clear and V set (>)
1100 = GT - Z clear, and either N set and V set, or N clear and V set (>)
1101 = LE - Z set, or N set and V clear,or N clear and V set (<, or =)
1110 = AL - always
1111 = NV - reserved.

36

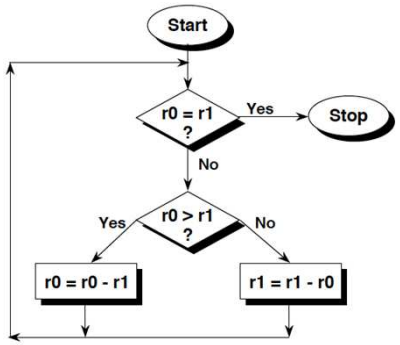
36

ARM ISA中的条件执行

* **To execute an instruction conditionally, simply postfix it with the appropriate condition:**
  - For example an add instruction takes the form:
    - `ADD r0,r1,r2 ; r0 = r1 + r2 (ADDAL)`
  - To execute this only if the zero flag is set:
    - `ADDEQ r0,r1,r2 ; If zero flag set then...`
      `; ... r0 = r1 + r2`
* **By default, data processing operations do not affect the condition flags (apart from the comparisons where this is the only effect). To cause the condition flags to be updated, the S bit of the instruction needs to be set by postfixing the instruction (and any condition code) with an "S".**
  - For example to add two numbers and set the condition flags:
    - `ADDS r0,r1,r2 ; r0 = r1 + r2`
      `; ... and set flags`

37

ARM ISA中的条件执行



* **Convert the GCD algorithm given in this flowchart into**
  1) "Normal" assembler, where only branches can be conditional.
  2) ARM assembler, where all instructions are conditional, thus improving code density.
* **The only instructions you need are CMP, B and SUB.**

38

ARM ISA中的条件执行

"Normal" Assembler

```
gcd     cmp r0, r1      ;reached the end?
        beq stop
        blt less        ;if r0 > r1
        sub r0, r0, r1  ;subtract r1 from r0
        bal gcd
less    sub r1, r1, r0  ;subtract r0 from r1
        bal gcd
stop
```

ARM Conditional Assembler

```
gcd     cmp   r0, r1      ;if r0 > r1
        subgt r0, r0, r1  ;subtract r1 from r0
        sublt r1, r1, r0  ;else subtract r0 from r1
        bne   gcd         ;reached the end?
```


39

理想化

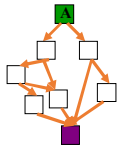
- 这样不好吗
  - 如果当一个分支真的会被预测错误的时候消除它（推断执行）
  - 如果一个分支会被预测正确的时候预测它
- 这样该多好
  - 如果推断不需要ISA支持

40

40

改进推断执行

- 推断的三个主要局限
  1. 适应性: 不能动态适应分支行为
  2. 复杂的控制流图: 不适用于循环/复杂的控制流图
  3. ISA: 需要ISA做较大的改动



- 愿望分支（Wish Branch） [Kim+, MICRO 2005]
  - 解决局限 1，部分解决局限 2 (循环)
- 动态推断执行
  - Diverge-Merge Processor [Kim+, MICRO 2006]
    - 解决局限 1, 2 (部分), 3

41

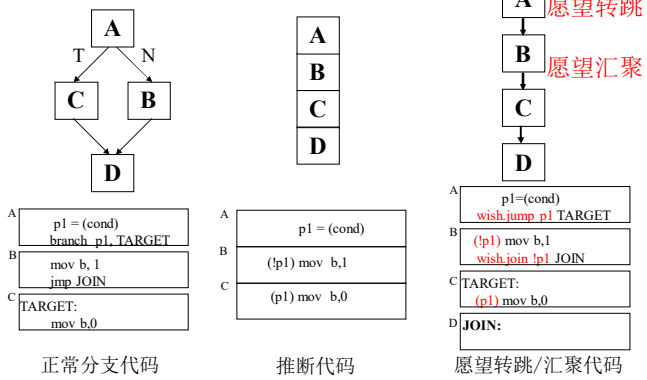
愿望分支

- **编译器**生成愿望分支代码，既可以当作推断代码也可以当作非推断代码（正常分支代码）执行
- **硬件**在运行时根据对分支预测的把握性来**决定**是执行推断代码还是正常分支代码
- **容易预测: 正常分支代码**
- **难预测: 推断代码**

- Kim et al., "Wish Branches: Enabling Adaptive and Aggressive Predicated Execution," MICRO 2006, IEEE Micro Top Picks, Jan/Feb 2006.

42

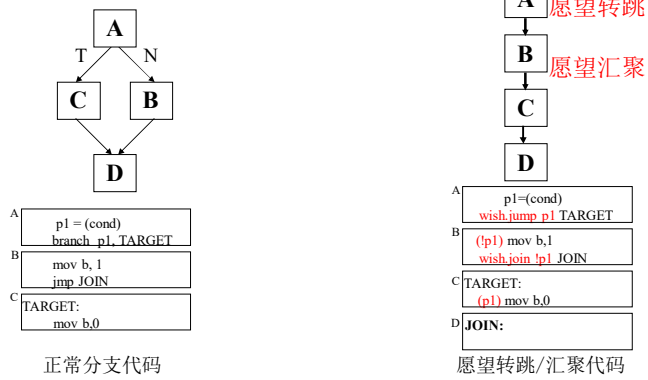
愿望转跳/汇聚



43

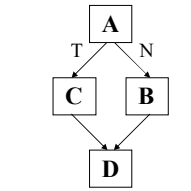
愿望转跳/汇聚

把握大



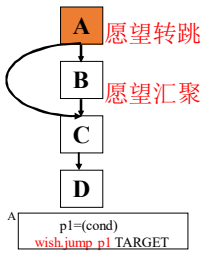
44

愿望转跳/汇聚



发生

把握大




```
A  p1 = (cond)
   branch_p1, TARGET
B  mov b, 1
   jmp JOIN
C  TARGET:
   mov b,0
```

正常分支代码

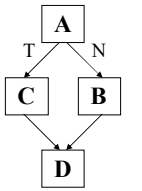
```
A  p1=(cond)
   wish.jump_p1 TARGET
```

愿望转跳/汇聚代码

45

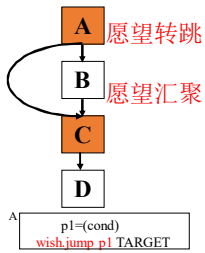
45

愿望转跳/汇聚



发生

把握大




```
A  p1 = (cond)
   branch_p1, TARGET
B  mov b, 1
   jmp JOIN
C  TARGET:
   mov b,0
```

正常分支代码

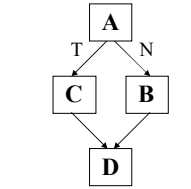
```
A  p1=(cond)
   wish.jump_p1 TARGET

C  TARGET:
   (1)  mov b,0
```

愿望转跳/汇聚代码

46

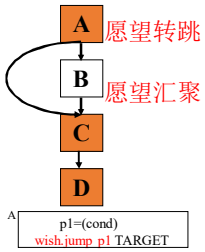
46

愿望转跳/汇聚



发生

把握大




```
A  p1 = (cond)
   branch_p1, TARGET
B  mov b, 1
   jmp JOIN
C  TARGET:
   mov b,0
```

正常分支代码

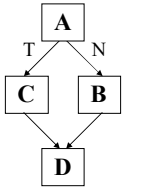
```
A  p1=(cond)
   wish.jump_p1 TARGET

C  TARGET:
   (1)  mov b,0
D  JOIN:
```

愿望转跳/汇聚代码

47

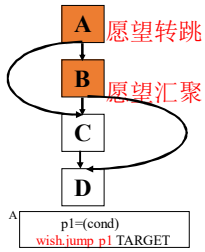
47

愿望转跳/汇聚



未发生

把握大




```
A  p1 = (cond)
   branch_p1, TARGET
B  mov b, 1
   jmp JOIN
C  TARGET:
   mov b,0
```

正常分支代码

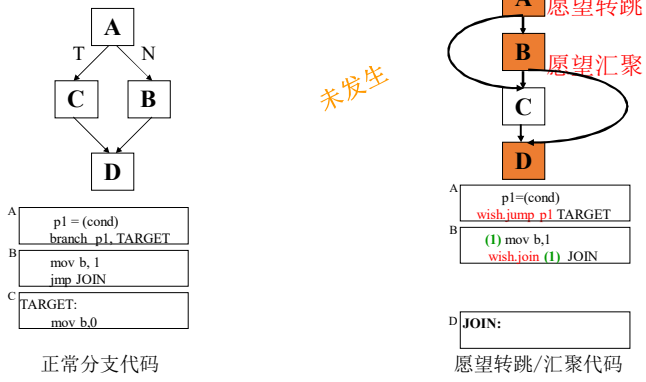
```
A  p1=(cond)
   wish.jump_p1 TARGET
B  (1) mov b,1
   wish.join (1)  JOIN
```

愿望转跳/汇聚代码

48

48

愿望转跳/汇聚

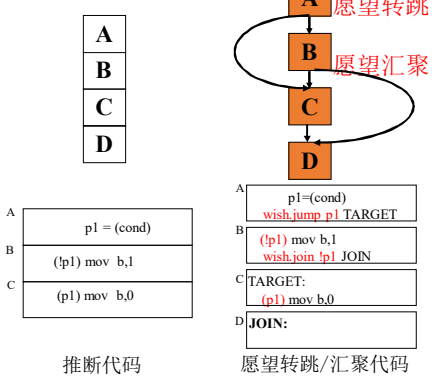


正常分支代码　　愿望转跳/汇聚代码

49

49

愿望转跳/汇聚

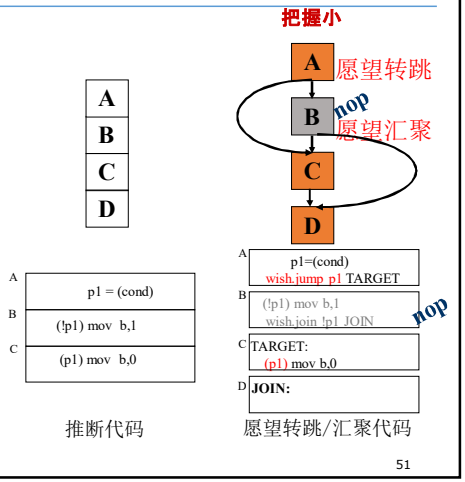


推断代码　　愿望转跳/汇聚代码

50

50

愿望转跳/汇聚



推断代码　　愿望转跳/汇聚代码

51

51

愿望分支vs. 推断执行

- 相较于推断执行的优点
  - **减小了推断的开销**
  - 通过使编译器生成更积极的推断代码以增加推断代码的收益
  - 使推断代码较少依赖于机器的配置 (比如分支预测器)
- 相较于推断执行的缺点
  - 额外的分支指令占用机器资源
  - 额外的分支指令竞争分支预测表项
  - 限制了编译器**代码优化**的空间

52

52

如何处理控制相关

- 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）

53

53

53

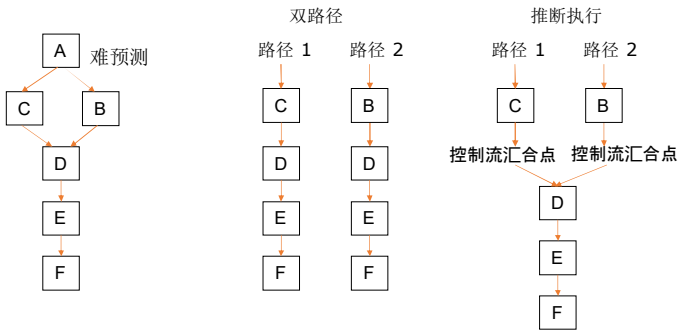
多路径执行

- 思路: 在条件分支之后两条路径都执行
  - 对于所有的分支: Riseman and Foster, "The inhibition of potential parallelism by conditional jumps," IEEE Transactions on Computers, 1972.
  - 对于难预测的分支: 动态评估把握性
- 好处:
  - + 当预测错的开销>无用功的开销时，可以提高性能
  - + 不需要ISA做什么改变
- 坏处:
  - -- 当遇到下一个难预测的分支该怎么办？再次执行两条路径？
  - -- 路径数成指数增加
  - -- 每条接下来的路径需要自己的寄存器, PC, GHR
  - -- 如果路径最终合并会导致无用功 (降低性能)

54

54

双路径执行vs. 推断



55

55

如何处理控制相关

- 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）

56

56

分支延迟(I)

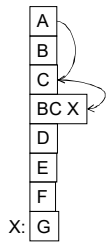
- 改变分支指令的语义
  - N条指令后分支
  - N个周期后分支
- 思路：延迟分支的执行，无论分支的方向如何，总是执行分支指令后紧跟的N条指令（延迟槽）
- 问题：如何找到指令填充延迟槽?
  - 分支必须与延迟槽指令相互独立
- 无条件分支：更容易找到填充延迟槽的指令
- 条件分支：条件计算不应依赖于延迟槽中的指令 → 填充延迟槽有难度

57

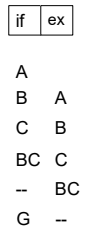
57

分支延迟(II)

正常的代码:

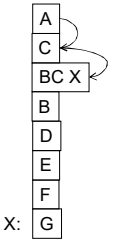


时间线:

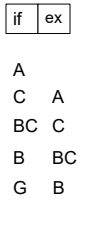


6 个周期

分支延迟的代码:



时间线:



5 个周期

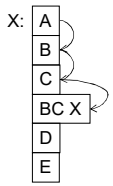
58

58

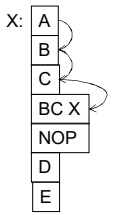
更有想象力的分支延迟(III)

- 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?

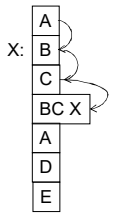
正常代码:



分支延迟代码:



带“挤压”的分支延迟:



59

59

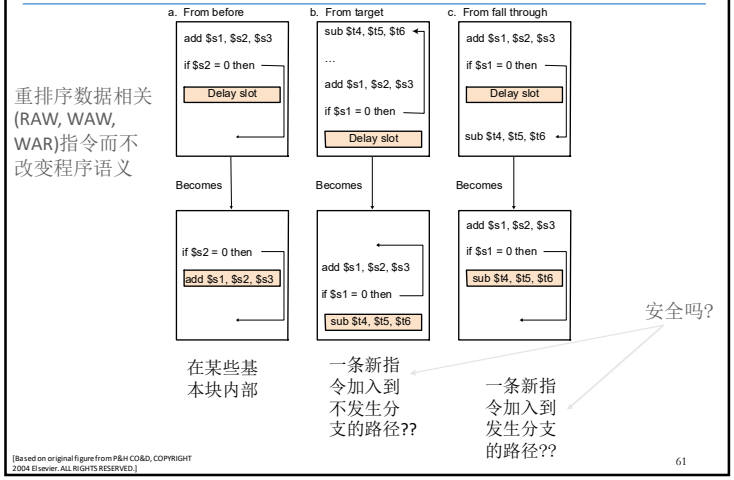
分支延迟(IV)

- 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?

60

60

填充延迟槽



重排序数据相关(RAW, WAW, WAR)指令而不改变程序语义

在某些基本块内部

一条新指令加入到不发生分支的路径??

一条新指令加入到发生分支的路径??

安全吗?

61

61

如何处理控制相关

- 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）

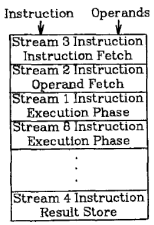
62

62

细粒度多线程

- 思路：硬件具有多线程的上下文。每个周期，指令获取机制从不同的线程获取指令
  - 当获取的分支/指令解析时，不需要在相同的线程中继续获取另一条指令
  - 解决分支/指令的延迟与其它线程的执行重叠

+ 处理一个线程内的控制和数据相关
  不需要额外的逻辑
-- 单线程的性能会降低
-- 用额外的逻辑保持线程上下文
-- 如果没有足够的线程覆盖整个流水线
  就不会有延迟重叠



63

63

细粒度多线程

- 思路：每个周期都切换到另一个线程，这样可以保证不会有同一个线程的两条指令并发的出现在流水线里
- 通过与来自其它线程有用的操作重叠延迟来容忍控制和数据相关带来的延迟
- 利用多线程的好处来改善流水线的利用率

- Thornton, "Parallel Operation in the Control Data 6600," AFIPS 1964.
- Smith, "A pipelined, shared resource MIMD computer," ICPP 1978.

64

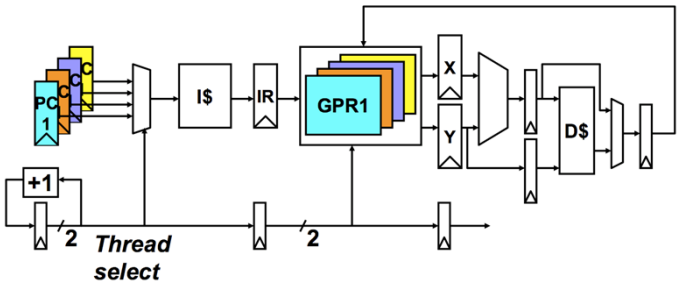
64

细粒度多线程：历史

- CDC 6600’的外围处理单元采用了细粒度多线程
  - Thornton, “Parallel Operation in the Control Data 6600,” AFIPS 1964.
  - 处理器每个周期执行不同的 I/O 线程
  - 每10个周期执行来自同一个线程的下一个操作
- Denelcor HEP (异构元处理器)
  - Smith, “A pipelined, shared resource MIMD computer,” ICPP 1978.
  - 120 个线程/处理器
  - 线程的可用队列和不可用队列 （等待）
  - 每个线程只能有1条指令在处理器的流水线中； 每个线程相互独立
  - 对每个线程而言，处理器看上去像一个非流水线的机器
  - 系统的吞吐量和单线程性能之间的折衷

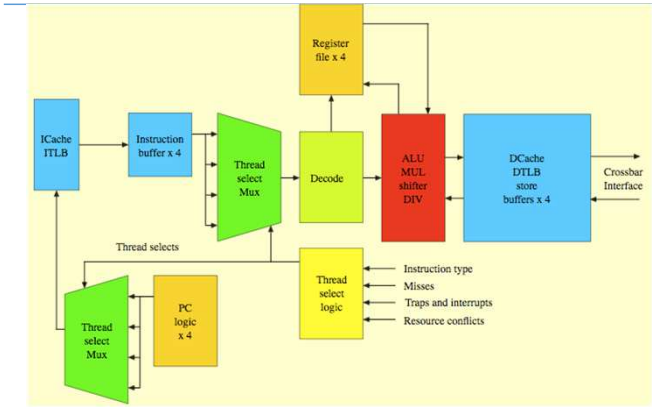
65

多线程的流水线示例



66

Sun Niagara 多线程流水线



Kongetira et al., “Niagara: A 32-Way Multithreaded Sparc Processor,” IEEE Micro 2005.

67

细粒度多线程

- 好处
  - + 不需要做指令间的相关性检查 （一个线程只有一条指令在流水线中）
  - + 不需要分支预测逻辑
  - + bubble周期被用来执行不同线程有用的指令
  - + 改善系统的吞吐量，延迟容忍和利用率
- 坏处
  - – 额外的硬件复杂性：多硬件上下文，线程选择逻辑
  - – 单线程性能下降（每隔N个周期取一条指令）
  - – 线程之间对cache和memory的资源争用
  - – 仍然需要一些线程之间的相关性检查逻辑（load/store）

68

如何处理控制相关

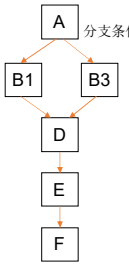
- 关键在于使流水线保持充满正确的动态指令序列
- 当指令是控制指令时可能的解决方案有：
  - 停顿流水线直到得到下一条指令的取指地址
  - 猜测下一条指令的取指地址（分支预测）
  - 采用延迟分支（分支延迟槽/时隙）
  - 其它(细粒度多线程)
  - 消除控制指令（推断执行）
  - 从所有可能的方向取指（如果知道的话）（多路径执行）

69

69

分支预测

- 处理器采用流水线为增加并发性
- 如何在存在分支时保持流水线满负荷?
  - 分支指令时猜测下一条指令
  - 需要猜测分支的方向和目标



流水线

取指	译码	重命名	调度	读寄存器堆		执行
A						

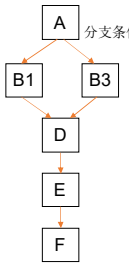
下一条取谁?

70

70

分支预测

- 处理器采用流水线为增加并发性
- 如何在存在分支时保持流水线满负荷?
  - 分支指令时猜测下一条指令
  - 需要猜测分支的方向和目标



流水线

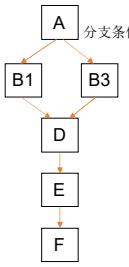
取指	译码	重命名	调度	读寄存器堆		执行
B1	A					

71

71

分支预测

- 处理器采用流水线为增加并发性
- 如何在存在分支时保持流水线满负荷?
  - 分支指令时猜测下一条指令
  - 需要猜测分支的方向和目标



流水线

取指	译码	重命名	调度	读寄存器堆		执行
D	B1	A				

72

72

分支预测

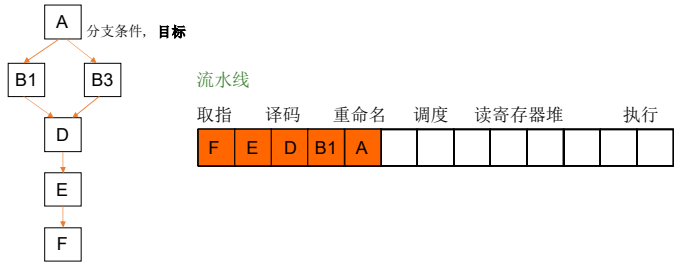
- 处理器采用流水线为增加并发性
- 如何在存在分支时保持流水线满负荷?
  - 分支指令时猜测下一条指令
  - 需要猜测分支的方向和目标

73

73

分支预测

- 处理器采用流水线为增加并发性
- 如何在存在分支时保持流水线满负荷?
  - 分支指令时猜测下一条指令
  - 需要猜测分支的方向和目标

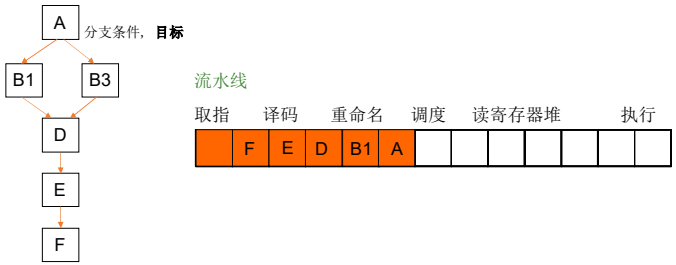


74

74

分支预测

- 处理器采用流水线为增加并发性
- 如何在存在分支时保持流水线满负荷?
  - 分支指令时猜测下一条指令
  - 需要猜测分支的方向和目标

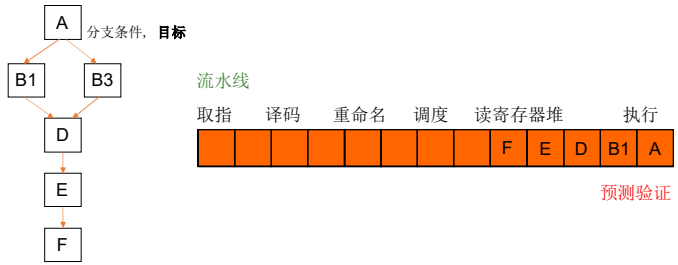


75

75

分支预测

- 处理器采用流水线为增加并发性
- 如何在存在分支时保持流水线满负荷?
  - 分支指令时猜测下一条指令
  - 需要猜测分支的方向和目标

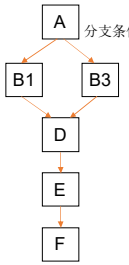


76

76

分支预测

- 处理器采用流水线为增加并发性
- 如何在存在分支时保持流水线满负荷?
  - 分支指令时猜测下一条指令
  - 需要猜测分支的方向和目标



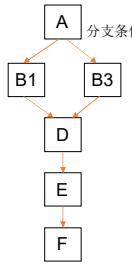
流水线

预测错误则清空流水线

77

分支预测

- 处理器采用流水线为增加并发性
- 如何在存在分支时保持流水线满负荷?
  - 分支指令时猜测下一条指令
  - 需要猜测分支的方向和目标

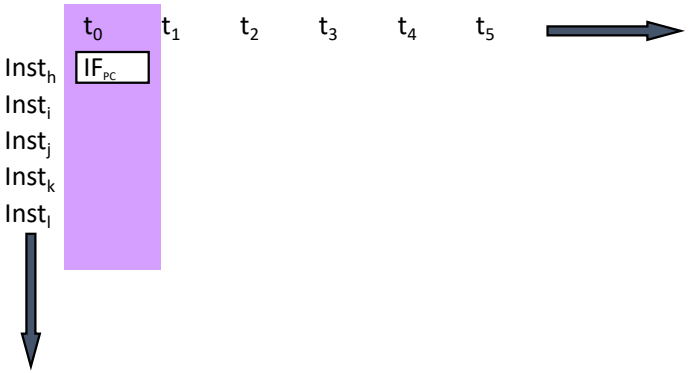


流水线

从正确的目标重新取指

78

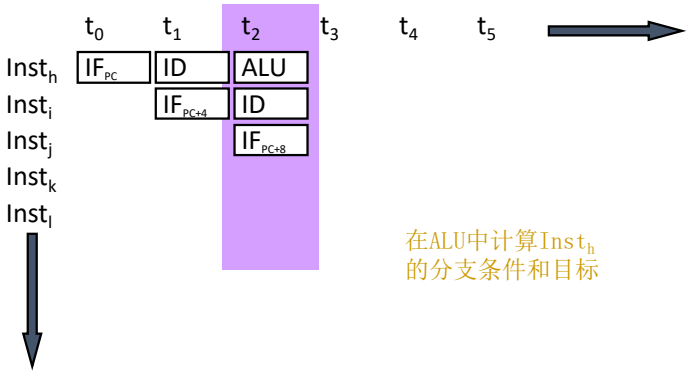
分支预测：总是 PC+4



$Inst_h$ 是一个分支指令

79

分支预测：总是 PC+4

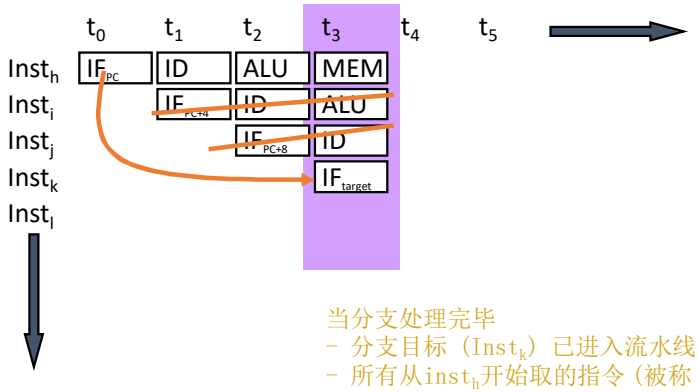


在ALU中计算$Inst_h$的分支条件和目标

$Inst_h$ 是一个分支指令

80

分支预测：总是 PC+4

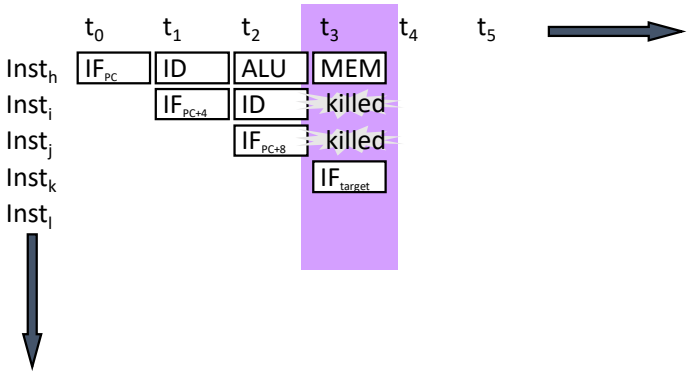


当分支处理完毕
- 分支目标（$Inst_k$）已进入流水线
- 所有从$inst_h$开始取的指令（被称为“错误路径”指令）必须被清空

$Inst_h$ 是一个分支指令

81

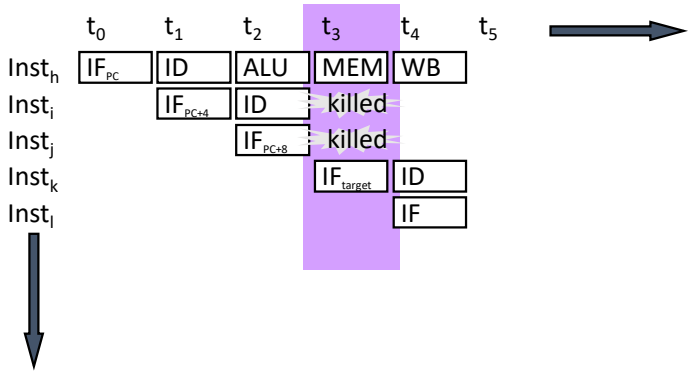
预测错误导致的流水线清空



$Inst_h$ 是一个分支指令

82

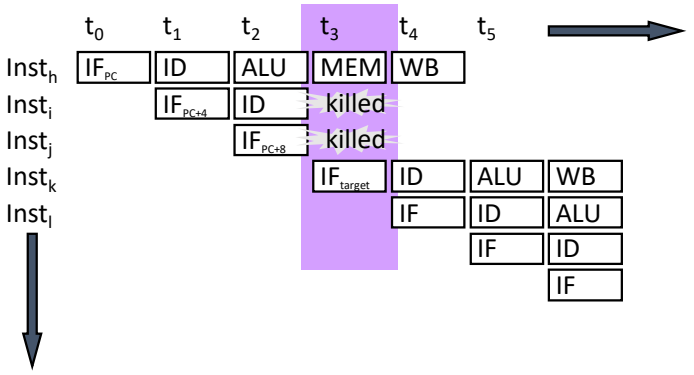
预测错误导致的流水线清空



$Inst_h$ 是一个分支指令

83

预测错误导致的流水线清空



$Inst_h$ 是一个分支指令

84

性能分析

- 猜测正确 ⇒ 没有惩罚　　~86% 的时间
- 猜测不正确 ⇒ 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

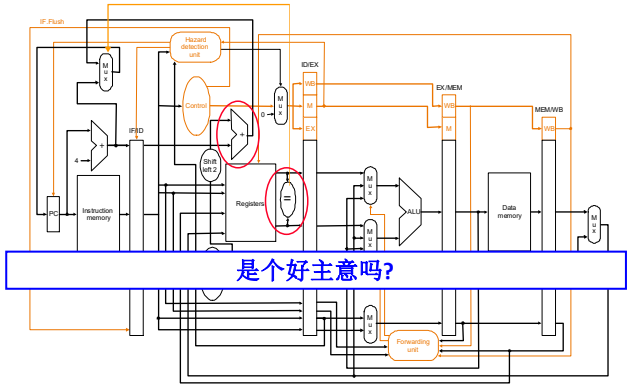
发生错误猜测的可能性　　错误猜测的惩罚

我们有可能减小这两者中的任何一个吗?

85

减小分支预测错误的代价

- 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

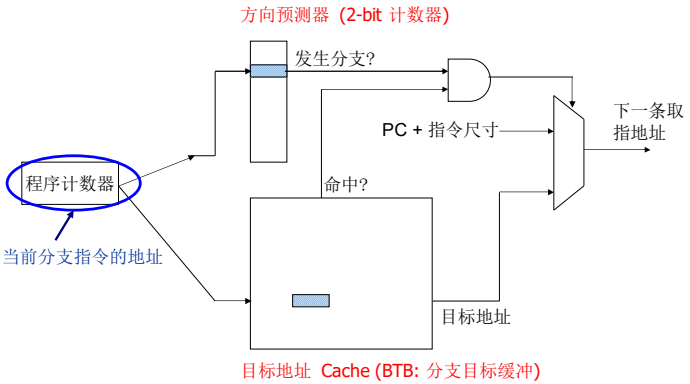
86

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

87

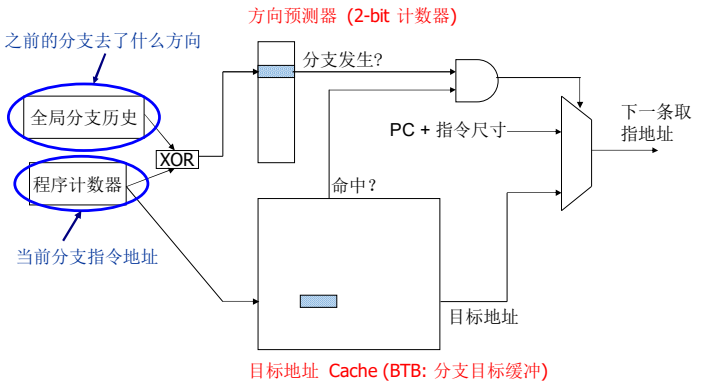
有BTB的取指和方向预测



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

88

更复杂的分支方向预测



89

89

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)

90

90

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

91

91

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

92

92

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

- Ball and Larus, "Branch prediction for free," PLDI 1993.
  - 20% 的预测错误率

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）
- 如何组合?
- 这三种技术有什么共同的缺陷?
  - 不能适应分支行为的动态变化
    - 可以利用动态编译器来解决这个问题，但是不是细粒度的 （同时动态编译器也有它自己的开销…）

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

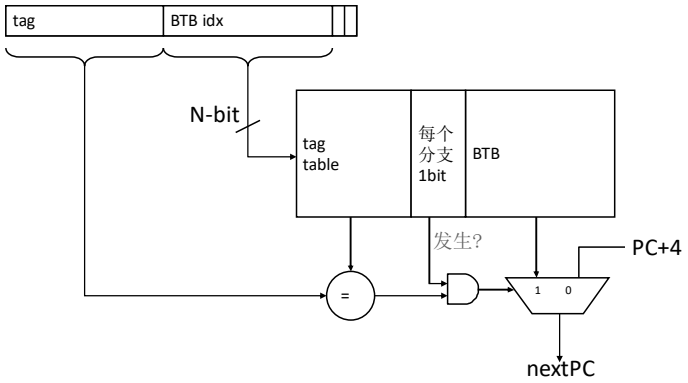
\+ 有大量迭代的循环分支

-- 只有少量迭代的循环分支

TNTNTNTNTNTNTNTNTNTN → 0% 准确率

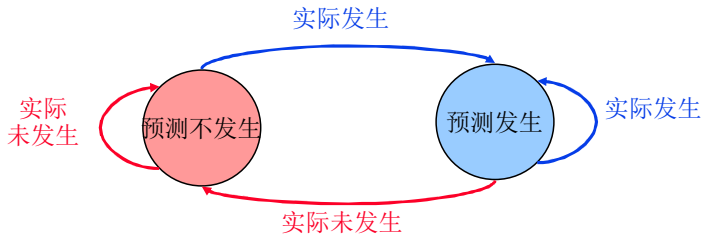
Last-time 预测器 CPI = [1 + (0.20*0.15) * 2] = 1.06 (假设 85% 准确率)

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



101

101

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
- Smith, "A Study of Branch Prediction Strategies," ISCA 1981.

102

102

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
TNTNTNTNTNTNTNTNTN → 50% 准确率
(假设初始时弱发生)

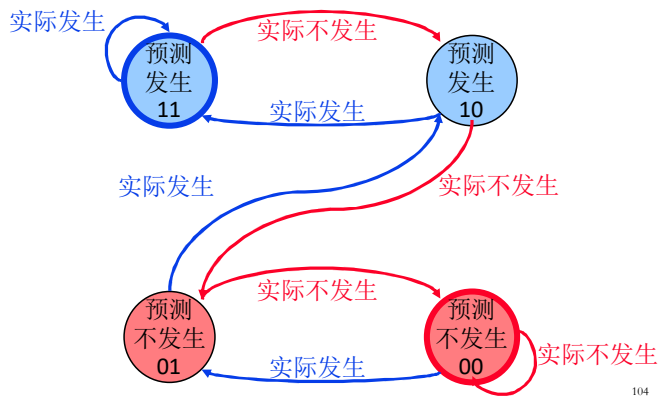
\+ 更好的预测准确率
2BC 预测器 CPI = [1 + (0.20*0.10) * 2] = 1.04 (90% 准确率)
-- 更多的硬件开销（但是计数器可以是BTB表项的一部分）

103

103

2-bit 饱和计数器的状态机

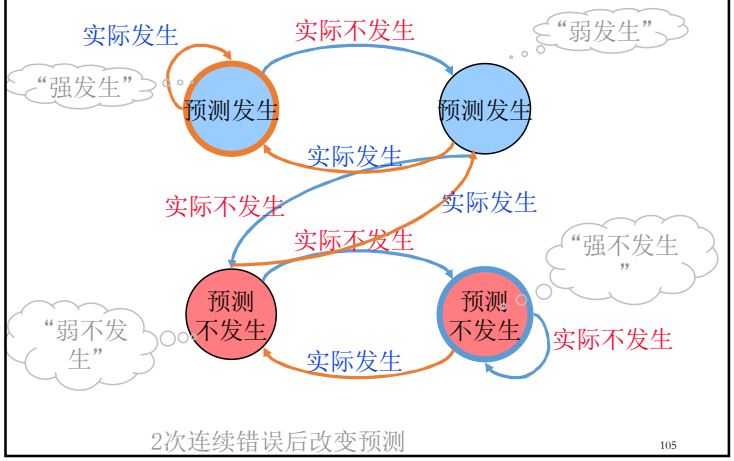
- 计数器使用饱和算术
  - 有一个代表最大值和最小值的符号



104

104

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

105

105

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?

106

106

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）

107

107

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

108

108

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

109

109

全局分支相关(I)

- 最近一个执行分支的结果与下一个分支结果相关

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

- 如果第一个分支不发生，第二个也不会发生

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

- 如果第一个发生了，第二个肯定不会发生

110

110

全局分支相关(II)

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

- 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

111

111

全局分支相关(III)

- 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

112

112

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

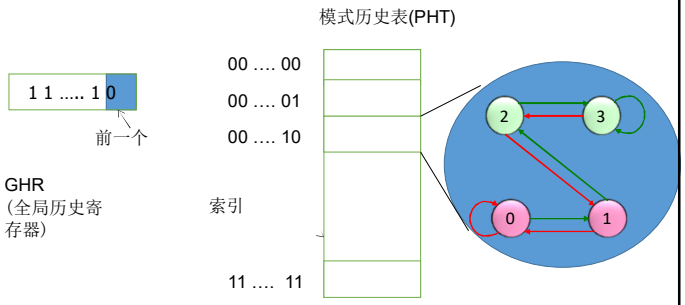
- 实现：
  - 用一个寄存器跟踪所有分支的“全局T/NT历史”→全局历史寄存器(GHR)
  - 使用GHR索引到一张表，表中记录了最近的过去与GHR中值相应的分支的结果 → 模式历史表(2-bit计数器表)

- 全局历史/分支预测器
- 使用两个层次的历史(GHR+GHR的历史)

113

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

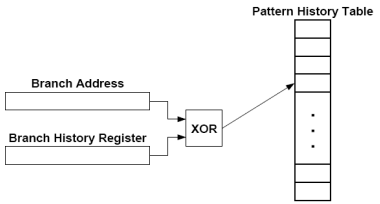


Yeh and Patt, “Two-Level Adaptive Training Branch Prediction,” MICRO 1991.

114

改进全局分支预测的准确性

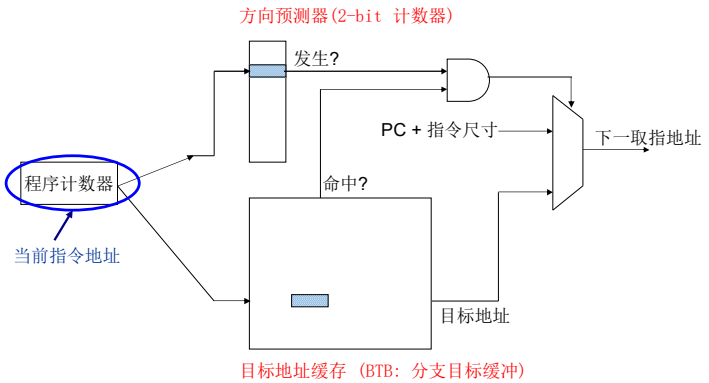
- 思路：为全局预测器增加更多的上下文信息来决定预测哪一个分支
  - Gshare预测器：GHR按分支地址哈希
  - \+ 更多的上下文信息
  - \+ 更好地利用模式历史表
  - -- 增加访问延迟



- McFarling, “Combining Branch Predictors,” DEC WRL Tech Report, 1993.

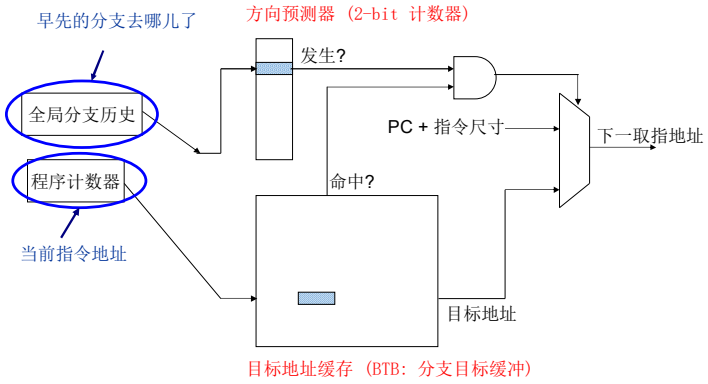
115

一层分支预测器



116

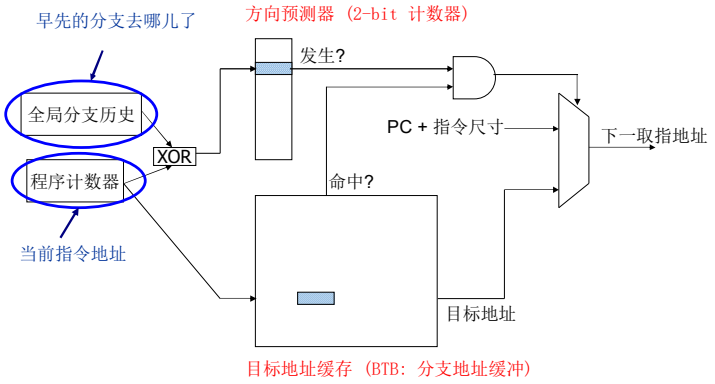
两层全局历史预测器



117

117

两层Gshare预测器



118

118

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

119

119

本地分支关联

```
for (i=1; i<=4; i++) { }
```

If the loop test is done at the end of the body, the corresponding branch will execute the pattern $(1110)^n$, where 1 and 0 represent taken and not taken respectively, and n is the number of times the loop is executed. Clearly, if we knew the direction this branch had gone on the previous three executions, then we could always be able to predict the next branch direction.

- McFarling, “Combining Branch Predictors,” DEC WRL TR 1993.

120

120

捕获本地分支的关联性

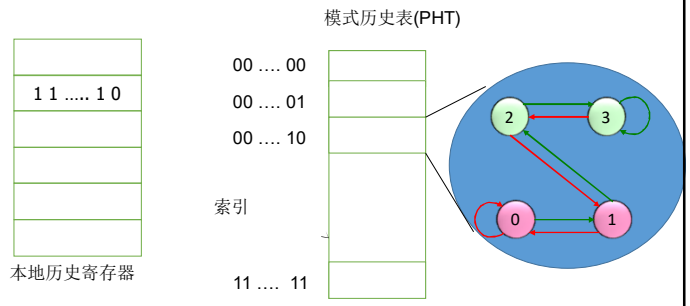
- 思路：每个分支都有一个历史寄存器
  - 将分支预测结果与该分支在“历史上发生/不发生”关联
- 根据上一次相同本地分支历史的分支结果作出预测
- 称为本地历史/分支预测器
- 使用两个层次的历史（每个分支历史寄存器 + 取那一个历史寄存器值的历史）

121

121

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

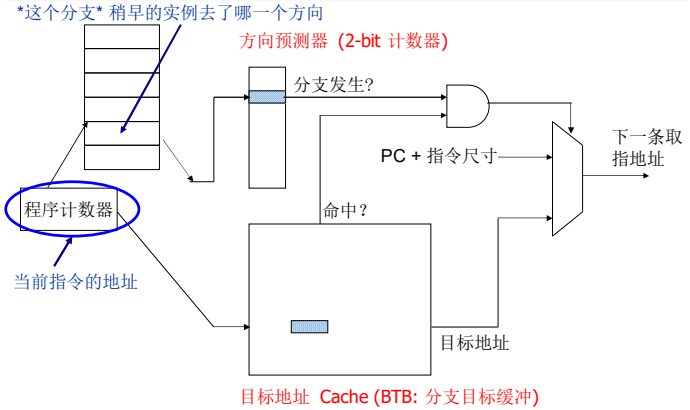


Yeh and Patt, “Two-Level Adaptive Training Branch Prediction,” MICRO 1991.

122

122

两层本地历史预测器



123

123

混合分支预测器

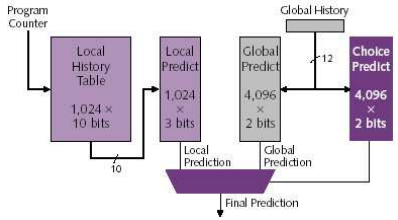
- 思路：使用不止一种类型的预测器（采用多种算法），选择“最佳”的预测
  - 比如，2-bit 计数器和全局预测器的混合
- 好处：
  - \+ 更好的准确率：不同的预测器更适用不同的分支
  - \+ 减少“热身”时间（先使用“进入状态”快的预测器，直到“慢热”的预测器热身完毕）
- 坏处：
  - -- 需要“元预测器”或“选择器”
  - -- 更长的访问时延

McFarling, “Combining Branch Predictors,” DEC WRL Tech Report, 1993.

124

124

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
- Kessler, "The Alpha 21264 Microprocessor," IEEE Micro 1999.

125

125

分支预测准确率(示例)

- 双模态: 由分支地址索引的2bc表

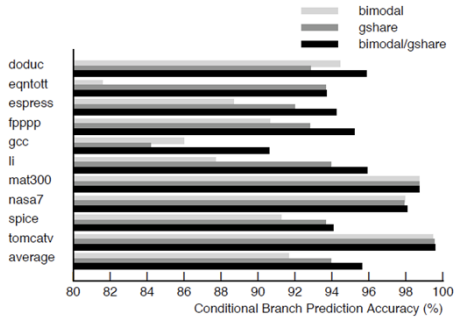


Figure 13: Combined Predictor Performance by Benchmark

126

126

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
- Chang et al., "Branch classification: a new mechanism for improving branch predictor performance," MICRO 1994.

127

127

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

128

128

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

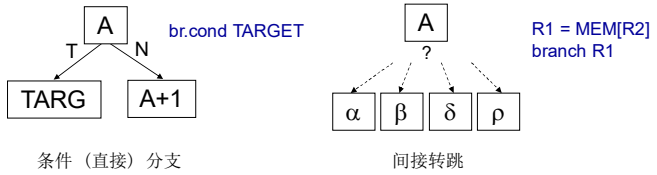
```
Call X
…
Call X
…
Call X
…
Return
Return
Return
```

- 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

129

间接分支预测(I)

- 寄存器间接分支有多个目标地址



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

130

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
  - Chang et al., "Target Prediction for Indirect Jumps," ISCA 1997.
  - \+ 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

131

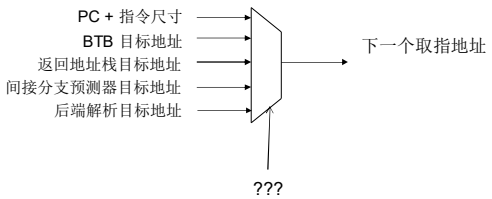
分支预测中的问题(I)

- 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

132

分支预测中的问题(II)

- 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



133

超标量处理器

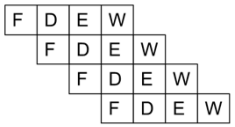
- “超标量”处理器
  - 尝试每个时钟周期执行超过1条指令
  - 每个周期必须取多条指令

- 考虑 2-way超标量取指的场景
  (case 1) 两条指令都是未发生的控制流指令
  - nPC = PC + 8

  (case 2) 其中一条指令是发生的控制流指令
  - nPC = 预测的目标地址
  - *注意* 两条指令都可以是控制流; 基于第一条指令预测发生与否
  - 如果第一条指令是预测发生的分支
    → 使第二条指令的取指失效

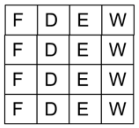
134

多指令取指: 概念



每周期取1条指令

缺点： Flynn 瓶颈
如果每周期取1条指令，那么每周期不可能完成>1条指令



每周期取>1条指令

两种可行的方法:
1、VLIW
编译器决定哪些指令可以并行执行，硬件简单
2、超标量
硬件检测同一个周期取到的指令之间的相关性

135

回顾：处理控制相关

- 处理流水线中的控制相关
  - 分支延迟
  - 细粒度多线程
  - 分支预测
    - 编译时(静态)
      - 总是不发生, 总是发生, 反向发生正向不发生, 基于分析
    - 运行时(动态)
      - Last time 预测器
      - 滞后: 2BC 预测器
      - 全局分支相关→ 两层全局预测器
      - 本地分支相关→ 两层本地预测器
  - 推断执行
  - 多路径执行

136