高等计算机体系结构
第五讲: 流水线的数据相关和控制相关

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

1

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）
  - 附录 D
  - 第四章 (4.5-4.8, 4.9-4.11)
- 选读
  - Patt & Patel's *Introduction to Computing Systems: From Bits and Gates to C and Beyond* （计算机系统概论）
    - 附录C ：LC-3b ISA及微体系结构

2

回顾：理想的流水线

- 目标：增加少量成本(指令处理的硬件开销)提升吞吐量
- 重复相同的操作
  - 对大量不同的输入执行同样的操作
- 重复独立的操作
  - 重复的操作之间没有相关性
- 统一划分子操作
  - 处理可以被平均地划分成相同延时的子操作(不共享资源)

3

回顾指令流水线：并非理想的流水线

■相同的操作... 不是!

⇒ 不同的指令不一定需要所有的阶段

- 迫使不同的指令流经相同的多段流水线

→ 外部碎片 (对于某些指令会有某些流水段闲置)

■统一的子操作 ... 不是!

⇒ 很难平衡不同的流水段

- 不是所有流水段都完成同样的工作量

→ 内部碎片 (有些流水段完成的太快但仍旧需要占用同样的时钟周期时间)

■独立的操作... 不是!

⇒ 指令之间互相不是独立的

- 需要检测和解决指令之间的相关性以确保流水线操作的正确性

→ 流水不是永远流动的 (它会停顿)

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

相关性的检测方法(I)

- 计分板 Scoreboarding
  - 寄存器堆中的每一个寄存器都有一个与之相关的有效位
  - 一条指令写一个寄存器时会重置该寄存器的有效位
  - 一条指令在译码阶段会检查所有相关资源和目的寄存器是否有效
    - 是：无需停顿… 没有相关
    - 否：该指令停顿
- 优点：
  - 简单… 每个寄存器1位
- 缺点：
  - 所有类型的相关都会导致停顿，不仅仅是流相关

7

7

反相关和输出相关时不停顿

- 如何修改计分板方法来实现这样的能力?

8

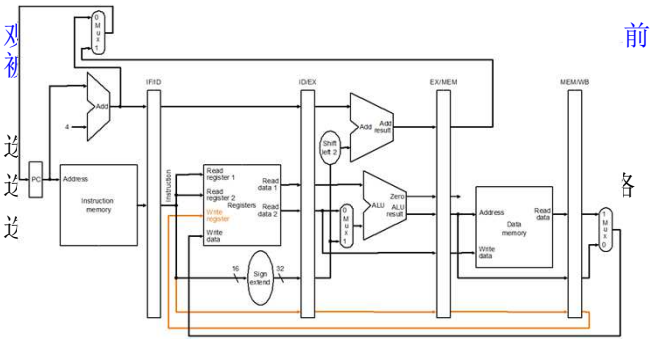
8

相关性的检测方法(II)

- 相关性检查逻辑（组合逻辑）
  - 用特殊的逻辑检查是否有任何前序指令会写任何当前译码指令的源操作数寄存器
  - 是：该指令/流水线停顿
  - 否：无需停顿… 没有流相关
- 优点：
  - 反相关和输出相关不会导致停顿
- 缺点：
  - 逻辑比计分板更复杂
  - 当我们设计的流水线结构更深更宽时逻辑会变得越发复杂

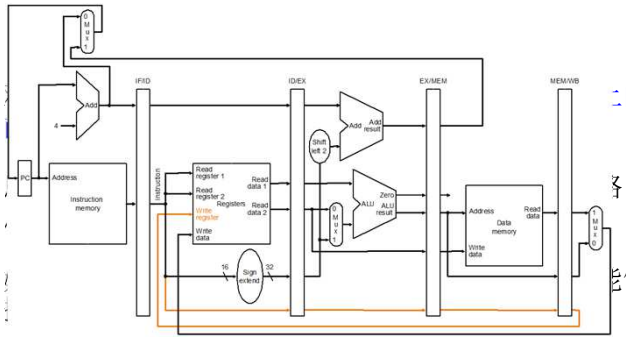
一旦检测出相关性

- 接下来该怎么做?
-
-
-
-



数据转发/旁路

- 问题：消费者指令（产生相关者）不得不等待在译码阶段直到生产者指令将值写回寄存器堆
-
-
-
-



数据相关的特例

- 控制相关
  - 有关指令指针/程序计数器的数据相关

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?

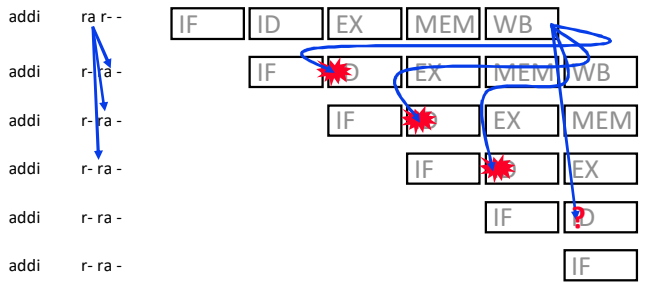
处理数据相关:
深入探索和实现方法

回顾：如何处理数据相关

- 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

处理写后读相关

- 下面的流相关导致5阶段流水线的冲突



寄存器数据相关性分析

	R/I-Type	LW	SW	Br	J	Jr
IF						
ID	读 RF	读 RF	读 RF	读 RF		读 RF
EX						
MEM						
WB	写 RF	写 RF				

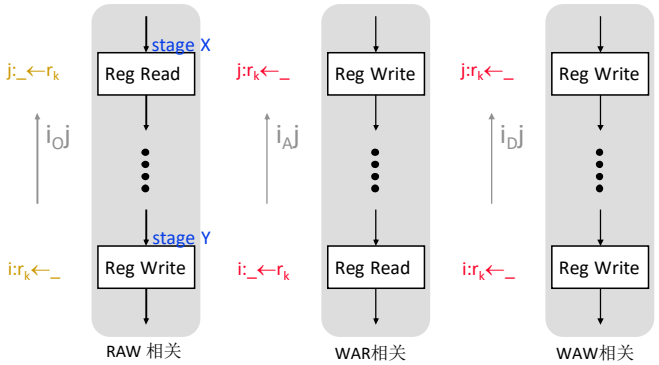
■对给定的流水线，什么情况下2条数据相关指令有潜在的冲突可能?

- ❑RAW, WAR, WAW?
- ❑与指令类型有关?
- ❑两条指令的距离?

17

17

流水线上的安全和不安全移动



dist(i,j) ≤ dist(X,Y) ⇒ 继续 j 不安全
dist(i,j) > dist(X,Y) ⇒ 安全

18

RAW 相关分析示例

	R/I-Type	LW	SW	Br	J	Jr
IF						
ID	读 RF	读 RF	读 RF	读 RF		读 RF
EX						
MEM						
WB	写 RF	写 RF				

- 指令 I_A 和 I_B (I_A 先于 I_B执行) 有 RAW 相关的条件
  - I_B (R/I, LW, SW, Br or JR) 读 I_A (R/I or LW) 写的寄存器
  - dist(I_A, I_B) ≤ dist(ID, WB) = 3

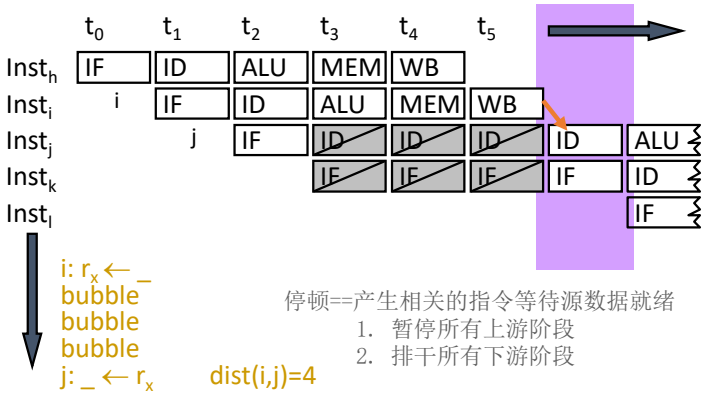
WAW 和 WAR 相关?

内存数据相关?

19

19

流水线停顿：解决数据相关

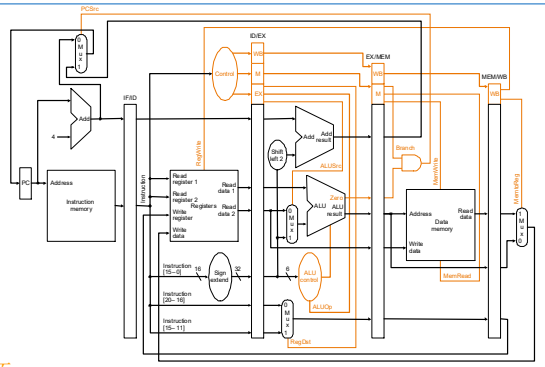


停顿==产生相关的指令等待源数据就绪

1. 暂停所有上游阶段
2. 排干所有下游阶段

20

如何实现停顿



- 停顿
  - 禁用 PC 和 IR 的触发；确保被停顿的指令停在它的阶段
  - 在停顿阶段之后的阶段插入“非法”指令/nops

21

停顿的条件

- 指令 I_A 和 I_B (I_A 先于 I_B执行) 有 RAW 相关的条件
  - I_B (R/I, LW, SW, Br or JR) 读 I_A (R/I or LW) 写的寄存器
  - dist(I_A, I_B) ≤ dist(ID, WB) = 3
- 换句话说，当处于ID阶段的指令 I_B 需要读的寄存器被指令 I_A 在 EX，MEM 或 WB 阶段写入时，指令 I_B 需要停顿

22

停顿的条件

- 辅助函数
  - rs(I) returns rs field of I
  - use_rs(I) returns true if I requires RF[rs] and rs!=r0
- 停顿 when
  - ($rs(IR_{ID})$==$dest_{EX}$) && $use\_rs(IR_{ID})$ && $RegWrite_{EX}$ or
  - ($rs(IR_{ID})$==$dest_{MEM}$) && $use\_rs(IR_{ID})$ && $RegWrite_{MEM}$ or
  - ($rs(IR_{ID})$==$dest_{WB}$) && $use\_rs(IR_{ID})$ && $RegWrite_{WB}$ or
  - ($rt(IR_{ID})$==$dest_{EX}$) && $use\_rt(IR_{ID})$ && $RegWrite_{EX}$ or
  - ($rt(IR_{ID})$==$dest_{MEM}$) && $use\_rt(IR_{ID})$ && $RegWrite_{MEM}$ or
  - ($rt(IR_{ID})$==$dest_{WB}$) && $use\_rt(IR_{ID})$ && $RegWrite_{WB}$
- 至关重要的是，EX，MEM 和 WB 阶段在后序指令停顿时会连续向前推进

23

停顿对性能的影响

- 每一个停顿周期实际上对应1个被浪费的ALU周期
- 对于一个有N条指令和S个停顿周期的程序而言，
  平均 CPI=(N+S)/N
- S 依赖于
  - 出现RAW相关的频率
  - 出现相关的指令的确切间隔
  - 相关之间的距离

24

示例

- for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) { ..... }

```
                addi    $s1, $s0, -1
for2tst:        slti    $t0, $s1, 0
                bne     $t0, $zero, exit2
                sll     $t1, $s1, 2
                add     $t2, $a0, $t1
                lw      $t3, 0($t2)
                lw      $t4, 4($t2)
                slt     $t0, $t4, $t3
                beq     $t0, $zero, exit2
                ........
                addi    $s1, $s1, -1
                j       for2tst
exit2:
```

3 stalls
3 stalls
3 stalls
3 stalls
3 stalls
3 stalls

数据转发(或数据旁路)

- 可以直观地将RF看作某种状态
  - “add rx ry rz”字面的意思就是分别从 RF[ry] 和 RF[rz] 取到值并把结果放进 RF[rx]
- 但是，RF只是计算抽象的一部分
  - “add rx ry rz”意味着 1. 得到前面指令的结果分别确定 RF[ry] 和 RF[rz] 的值，2. 直到另一条指令重新确定 RF[rx] 之前，指向RF[rx]的后序指令将使用这条指令的结果
- 重要的是保持操作之间正确的“数据流”，因此

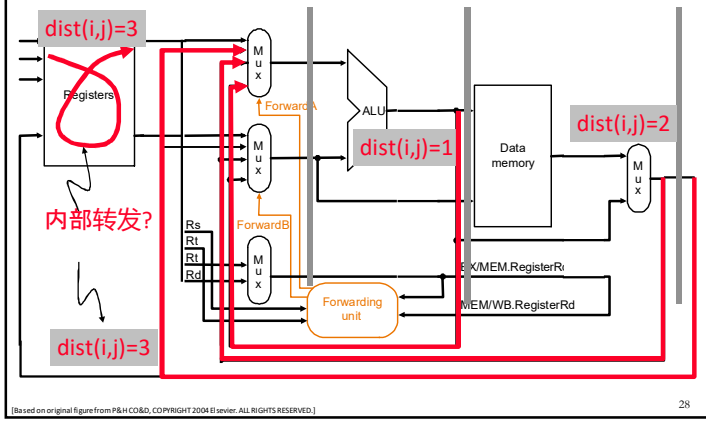
通过转发解决 RAW 相关

- 指令 I_A 和 I_B (I_A 先于 I_B执行) 有 RAW 相关的条件
  - I_B (R/I, LW, SW, Br or JR) 读 I_A (R/I or LW) 写的寄存器
  - $dist(I_A, I_B) \leq dist(ID, WB) = 3$
- 换句话说，当处于ID阶段的指令 I_B 需要读的寄存器被指令 I_A 在 EX，MEM 或 WB 阶段写入时，I_B需要的操作数不在RF中

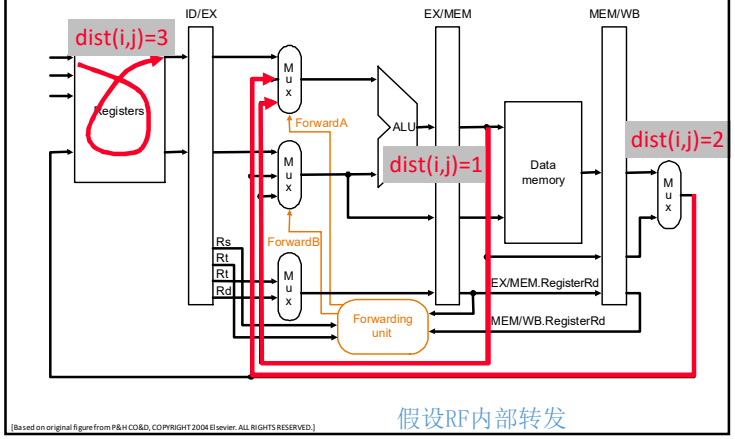
  ⇒ 从数据通路上而不是从RF中取回操作数

  ⇒ 当有多个未完成的操作数定义时取回最新产生的那一个

数据转发路径(v1)



数据转发路径(v2)



假设RF内部转发

29

数据转发逻辑(v2)

if (rs_{EX}!=0) && (rs_{EX}==$dest_{MEM}$) && $RegWrite_{MEM}$ then
  forward operand from MEM stage // dist=1
else if (rs_{EX}!=0) && (rs_{EX}==$dest_{WB}$) && $RegWrite_{WB}$ then
  forward operand from WB stage // dist=2
else
  use A_{EX} (operand from register file) // dist >= 3

排序很重要!! 必须先检查最新的匹配

上面的逻辑中没有考虑什么?

30

30

数据转发(相关性分析)

	R/I-Type	LW	SW	Br	J	Jr
IF						
ID						use
EX	use produce	use	use	use		
MEM		produce	(use)			
WB						

- 即使采用数据转发，如果与紧邻的前序LW指令存在RAW 相关也需要停顿

31

31

回顾前面的示例

- for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) { ..... }

```
            addi   $s1, $s0, -1
for2tst:    slti   $t0, $s1, 0
            bne    $t0, $zero, exit2
            sll    $t1, $s1, 2
            add    $t2, $a0, $t1
            lw     $t3, 0($t2)
            lw     $t4, 4($t2)
            nop
            slt    $t0, $t4, $t3
            beq    $t0, $zero, exit2
            ........
            addi   $s1, $s1, -1
            j      for2tst
exit2:
```

32

32

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?

33

33

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

34

34

处理控制相关

35

35

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?

36

36

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

37

37

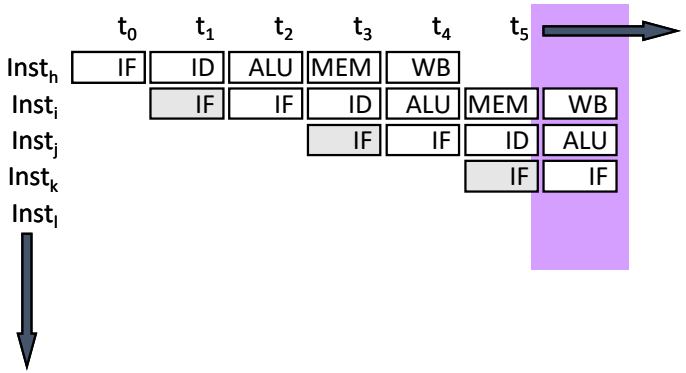
如何处理控制相关

- 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）

38

38

停止取指直到下一个PC可用：这是好主意吗?



对于非控制分支指令及无条件分支指令可能是的!

39

39

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?

40

40

猜测下一个PC = PC + 4

- 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：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…）

41

41

猜测下一个PC = PC + 4

- 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）
- 怎么做?
  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

42

42

组合推断

- 复杂的推断可被转换成多个分支
  - `if ((a == b) && (c < d) && (a > 5000)) { … }`
    - 3 个条件分支
- 问题：这会增加控制相关的数量
- 思路：将推断操作组合起来给一个分支指令，而不是每个推断一个分支
  - 推断的存储和操作利用条件寄存器
  - 用一个分支检查经过组合的推断的值

\+ 代码中的分支更少 → 更少的预测/停顿

\-- 可能增加不必要的工作

-- 如果第一个断言是错误的，就不需要计算后续的断言

- IBM RS6000 、POWER等体系结构中使用了条件寄存器

43

43

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）

44

44

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;
```

45

45

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?

46

46

推断执行

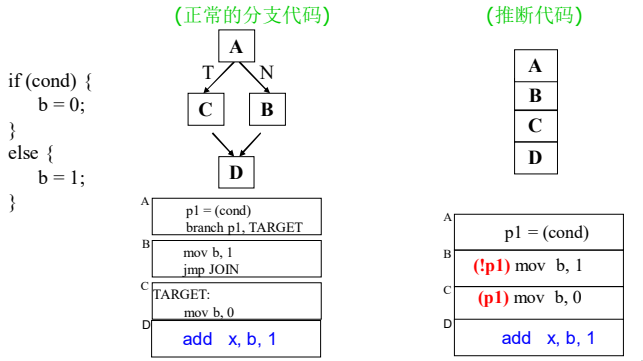
- 选读
  - Allen et al., “Conversion of control dependence to data dependence,” POPL 1983.
  - Kim et al., “Wish Branches: Combining Conditional Branching and Predication for Adaptive Predicated Execution,” MICRO 2005.

47

47

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)



48

48

推断执行(II)

- 推断执行可以具有较高的性能和能效

49

49

推断执行(III)

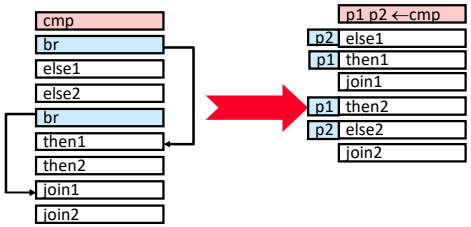
- 好处:
  - + 避免对难以预测的分支的错误预测
    - + 对某些分支不需要进行分支预测
    - + 如果预测错误的开销>由预测导致的无用功开销，适宜采用推断执行
  - + 可以对受到控制相关制约的代码进行优化
    - + 可以更加自由的移动推断执行的代码
- 坏处:
  - -- 一些容易预测的分支也会导致无用功
    - -- 如果预测错误的开销<无用功开销将使性能下降
    - -- 适应性: 静态的推断不适应运行时分支行为，分支行为的变化与输入、控制流的路径相关
  - -- 需要额外的硬件和ISA的支持
  - -- 无法消除所有的难预测分支
    - -- 循环分支?

50

50

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



51

51

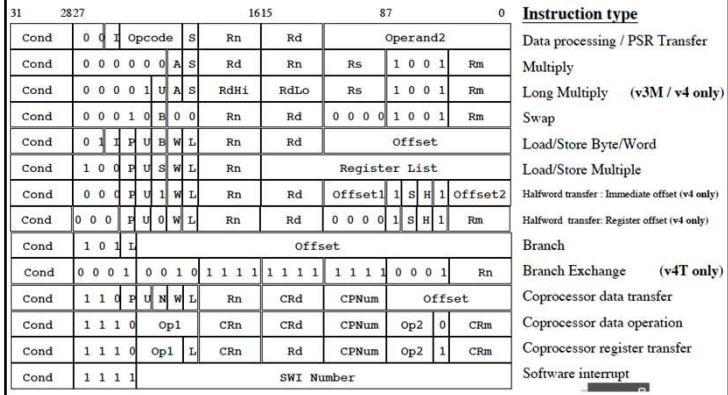
ARM ISA中的条件执行

- 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

52

52

ARM ISA中的条件执行

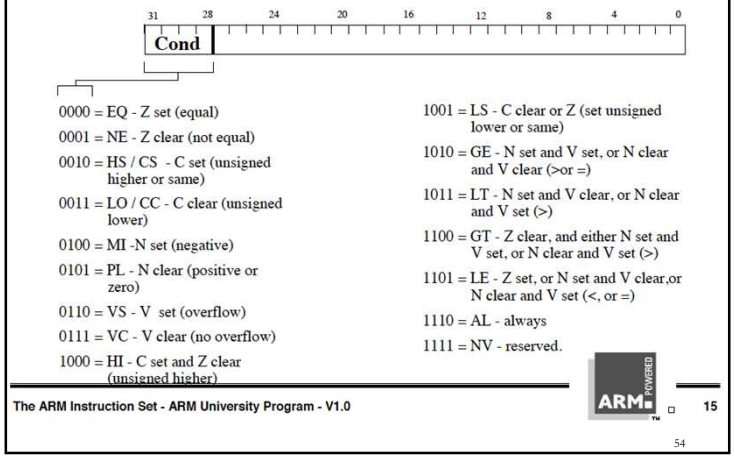


31–28	27–16	15–8	7–0	Instruction type
Cond	0 0 I Opcode S	Rn Rd	Operand2	Data processing / PSR Transfer
Cond	0 0 0 0 0 0 A S	Rd Rn	Rs 1 0 0 1 Rm	Multiply
Cond	0 0 0 0 1 U A S	RdHi RdLo	Rs 1 0 0 1 Rm	Long Multiply (v3M / v4 only)
Cond	0 0 0 1 0 B 0 0	Rn Rd	0 0 0 0 1 0 0 1 Rm	Swap
Cond	0 1 I P U B W L	Rn Rd	Offset	Load/Store Byte/Word
Cond	1 0 0 P U S W L	Rn	Register List	Load/Store Multiple
Cond	0 0 0 P U 1 W L	Rn Rd	Offset1 1 S H 1 Offset2	Halfword transfer : Immediate offset (v4 only)
Cond	0 0 0 P U 0 W L	Rn Rd	0 0 0 0 1 S H 1 Rm	Halfword transfer: Register offset (v4 only)
Cond	1 0 1 L	Offset		Branch
Cond	0 0 0 1	0 0 1 0 1 1 1 1	1 1 1 1 0 0 0 1 Rn	Branch Exchange (v4T only)
Cond	1 1 0 P U N W L	Rn CRd	CPNum Offset	Coprocessor data transfer
Cond	1 1 1 0	Op1 CRn	CRd CPNum Op2 0 CRm	Coprocessor data operation
Cond	1 1 1 0	Op1 L CRn	Rd CPNum Op2 1 CRm	Coprocessor register transfer
Cond	1 1 1 1	SWI Number		Software interrupt

53

53

ARM ISA中的条件执行



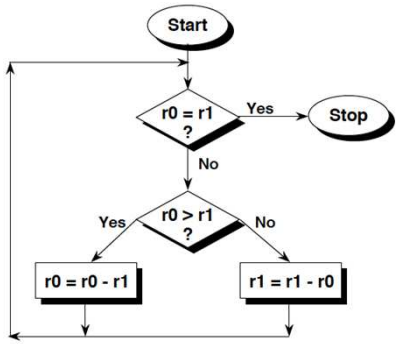
54

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`

55

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.**

56

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?
```


57

57

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

58

58

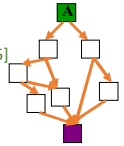
改进推断执行

■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

59

59

愿望分支

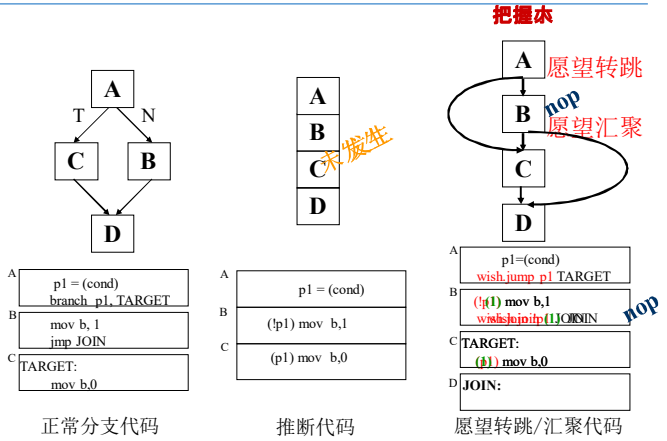
- **编译器**生成愿望分支代码，既可以当作推断代码也可以当作非推断代码（正常分支代码）执行
- **硬件**在运行时根据对分支预测的把握性来**决定**是执行推断代码还是正常分支代码
- **容易预测: 正常分支代码**
- **难预测: 推断代码**

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

60

60

愿望转跳/汇聚



61

61

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

62

62

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）

63

63

分支延迟(I)

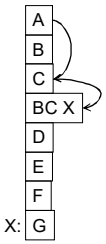
- 改变分支指令的语义
  - N条指令后分支
  - N个周期后分支
- 思路：延迟分支的执行，无论分支的方向如何，总是执行分支指令后紧跟的N条指令（延迟槽）
- 问题：如何找到指令填充延迟槽?
  - 分支必须与延迟槽指令相互独立
- 无条件分支：更容易找到填充延迟槽的指令
- 条件分支：条件计算不应依赖于延迟槽中的指令 → 填充延迟槽有难度

64

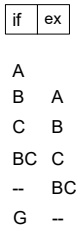
64

分支延迟(II)

正常的代码:

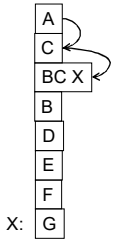


时间线:

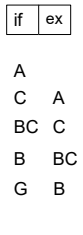


6 个周期

分支延迟的代码:



时间线:

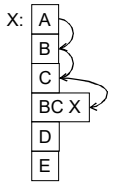


5 个周期

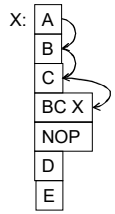
更有想象力的分支延迟(III)

- 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?

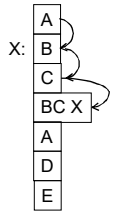
正常代码:



分支延迟代码:



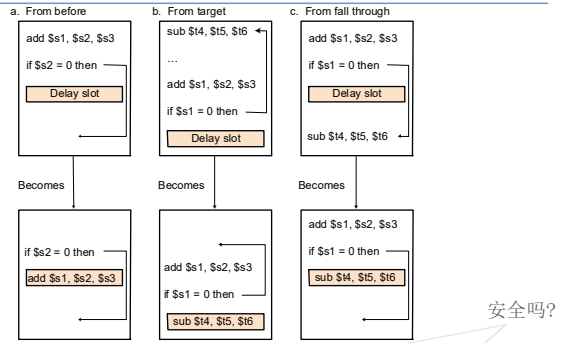
带“挤压”的分支延迟:



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?

填充延迟槽



重排序数据相关(RAW, WAW, WAR)指令不会改变程序语义

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?

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）

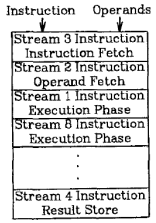
69

69

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



70

70

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.

71

71

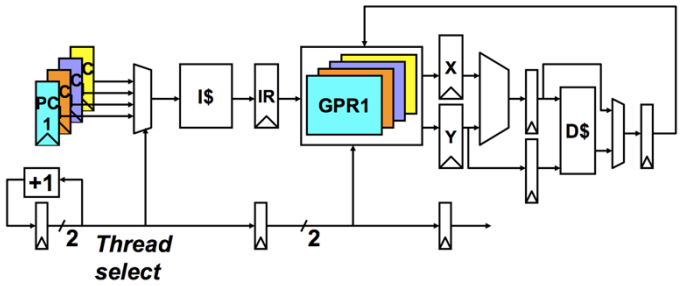
细粒度多线程：历史

- 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

72

72

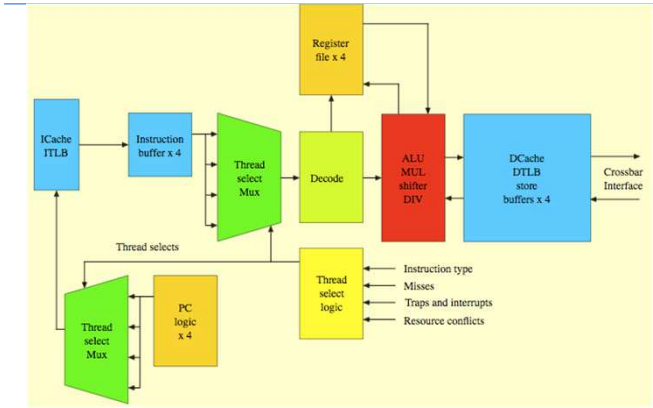
多线程的流水线示例



73

73

Sun Niagara 多线程流水线



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

74

74

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）

75

75

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）

76

76

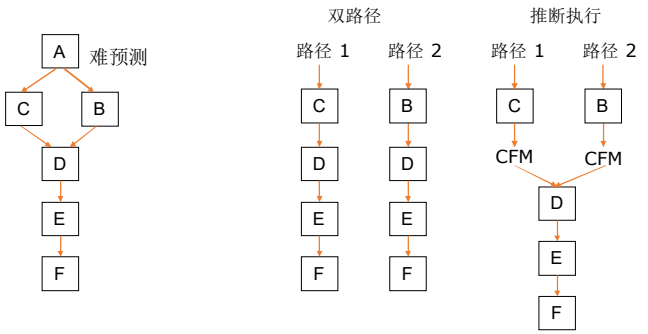
多路径执行

- 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)

77

77

双路径执行vs. 推断



78

78

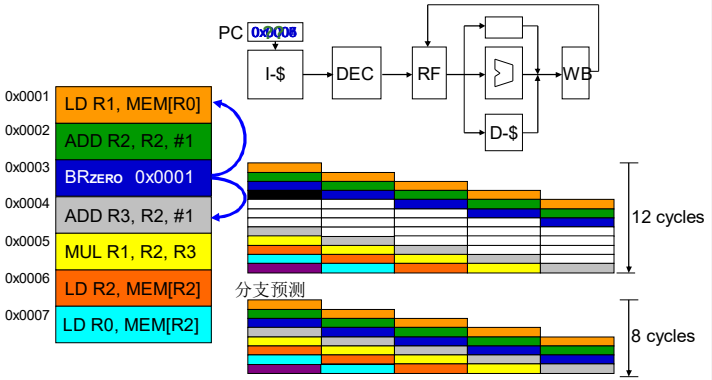
如何处理控制相关

- 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）

79

79

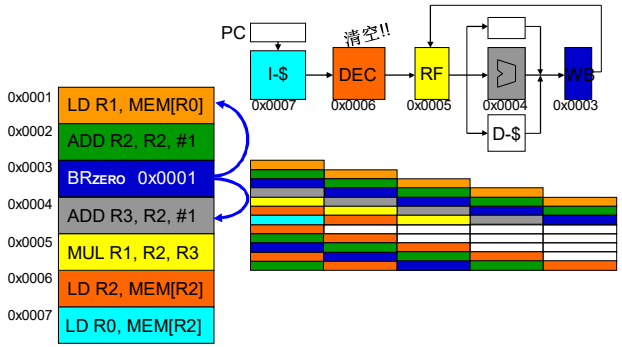
分支预测：猜测下一条获取的指令



80

80

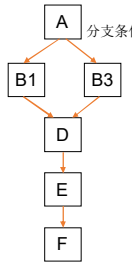
预测错误的代价



81

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

下一条取谁? 预测验证

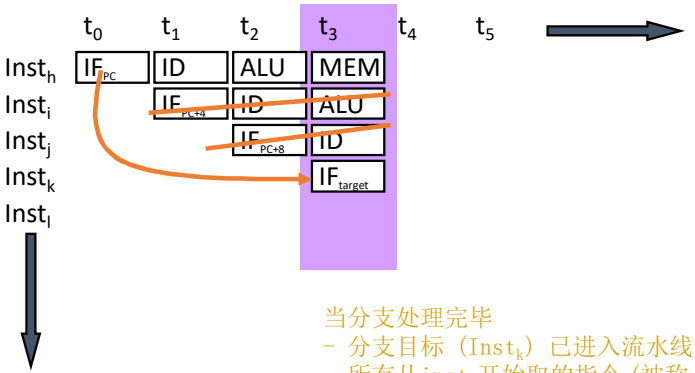
预测错误则清空流水线

从正确的目标重新取指

82

82

分支预测：总是 PC+4

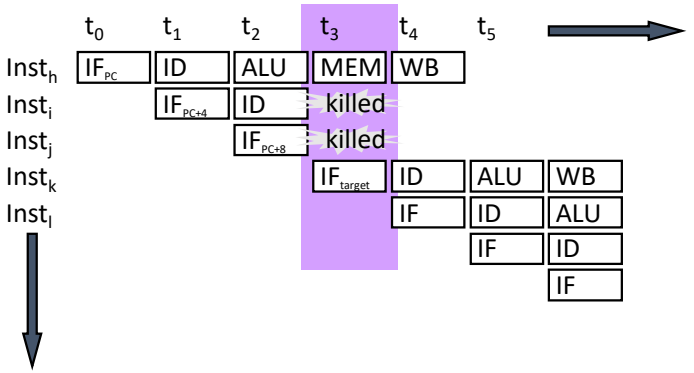


当分支处理完毕
- 分支目标（$Inst_k$）已进入流水线
- 所有从$inst_h$开始取的指令（被称为“错误路径”指令）必须被清空

$Inst_h$ 是一个分支指令

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

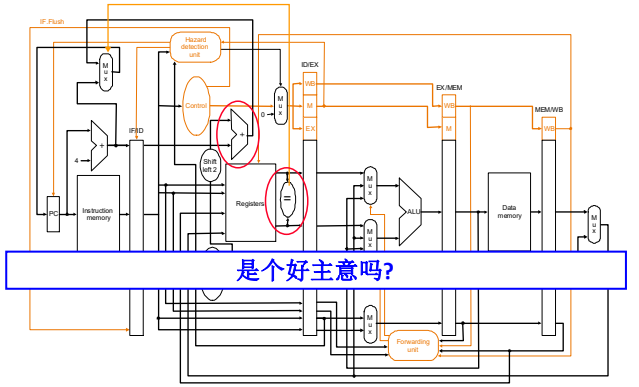
发生错误猜测的可能性　　错误猜测的惩罚

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

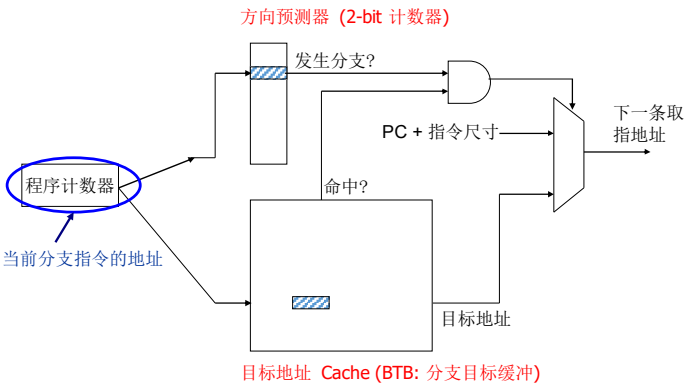
86

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