高等计算机体系结构
第六讲: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
2020-04-10

1

提醒：作业

- 作业 2
  - 今天截止
  - 单周期与多周期微体系结构
- 作业 3
  - 今晚发布，4月24日上课前截止提交
  - 流水线1

2

2

提醒：实验1

- 4月17日截止
  - 用Logisim设计1个7指令单周期MIPS CPU
- 学习MIPS ISA

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）
  - 附录 D
  - 第四章 (4.5-4.8, , 4.9-4.11)
- 选读
  - Maurice Wilkes, "The Best Way to Design an Automatic Calculating Machine," Manchester Univ. Computer Inaugural Conf., 1951.
  - Smith and Sohi, "The Microarchitecture of Superscalar Processors," Proceedings of the IEEE, 1995
  - 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

4

4

回顾：基本的多周期微体系结构

- 指令执行周期被划分为多个“状态”
  - 指令执行周期的每个阶段可以拥有多个状态
- 多周期微体系结构通过状态到状态的序列处理指令
  - 某个状态下机器的行为由该状态下的控制信号决定
- 整个处理器的行为可以被定义成一个有限状态机
- 在某个状态(时钟周期)中，控制信号控制
  - 数据通路如何处理数据
  - 如何为下一个时钟周期生成控制信号

5

5

回顾：微程序控制的优点

- 通过控制数据通路（用序列器），可以用非常简单的数据通路实现强有力的计算
  - 高级ISA翻译成微码（微指令序列）
  - 微码使得用最简单的数据通路***仿真***ISA成为可能
  - 微指令可以被看作是用户不可见的ISA
- 使ISA很容易扩展
  - 可以通过改变微码支持新的指令
  - 可以通过简单微指令的序列来支持复杂的指令
- 如果可以把任意指令序列化，那么也能够把任意“程序”序列化成微程序序列
  - 在微码中需要一些新的状态（如：循环计数器）来序列化更复杂的程序

6

6

回顾：是否可以更好?

- 在多周期设计中你看到哪些局限?
- 有限的并发
  - 在指令处理周期的不同阶段，一些硬件资源会闲置
  - 例如，当指令在“译码”或“执行”阶段，“取指”逻辑会闲置
  - 当发生访存时绝大多数数据通路闲置

7

7

回顾：流水线的基本思想

- 系统性更强
  - 多条指令流水线执行
  - 类比：指令的“装配线处理”
- 思路
  - 指令处理周期切分为不同的处理“阶段”
  - 保证有足够的硬件资源在每个阶段处理指令
  - 每个阶段处理不同的指令
    - 指令在连续的阶段中按照程序序连续地处理
- 好处：提升了指令处理的吞吐量（1/CPI）
- 坏处?

8

8

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)

9

9

回顾：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)

10

10

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

11

11

回顾：产生流水线*停顿*的原因

- 资源争用
- 相关性（指令之间）
  - 数据
  - 控制
- 长时延（多个周期）操作

12

12

回顾：相关和相关的类型

- 也叫“依赖”或者“冒险”
- 相关性表明了指令之间关于“序”的需求
- 两种类型
  - 数据相关
  - 控制相关
- 资源争用有时也叫资源相关
  - 但是,这种“相关”不是由程序语义表明的基本类型，所以我们把它和上面两种相关区别对待

13

13

回顾：处理资源争用

- 当处于两个流水段的指令需要同一个资源时会发生争用
- 解决方案 1：消除争用的起因
  - 复制资源或者提高资源的吞吐能力
    - 例如，将指令存储器（Cache）和数据存储器（Cache）分开
    - 例如，为存储结构设计多个端口
- 解决方案 2：检测资源争用，使其中一个争用流水段停顿
  - 让哪一个流水段停顿?
  - 例如：如果你的寄存器堆分别只有一个读和写端口会怎么样?

14

14

数据相关

- 数据相关的类型
  - 流相关(真正的数据相关-写后读)
  - 输出相关（写后写）
  - 反相关（读后写）
- 哪一种（些）会导致流水线停顿?
  - 对所有这些数据相关，都需要确保程序的语义正确
  - 流相关总是需要处理的，因为它构成了对一个***值***的真正相关
  - 反相关和输出相关的存在是由于（体系结构）寄存器数量有限
    - 它们是关于***名字***的相关，不是***值***

15

15

数据相关的类型

流相关

$r_3 \leftarrow r_1$ op r_2 写后读

$r_5 \leftarrow r_3$ op r_4 (RAW)

反相关

$r_3 \leftarrow r_1$ op r_2 读后写

$r_1 \leftarrow r_4$ op r_5 (WAR)

输出相关

$r_3 \leftarrow r_1$ op r_2 写后写

$r_5 \leftarrow r_3$ op r_4 (WAW)

$r_3 \leftarrow r_6$ op r_7

16

16

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
- 流相关更有意思
- 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

17

17

互锁

- 在流水线处理器中检测指令之间的相关性以确保执行正确
- 基于软件的互锁

  vs.
- 基于硬件的互锁
- MIPS 是什么的首字母缩写?
  - **M**icroprocessor without **I**nterlocked **P**iped **S**tages

18

18

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源寄存器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

19

19

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?

20

20

相关性的检测方法(II)

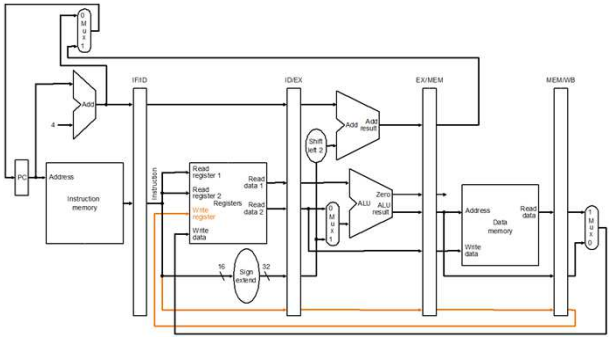
- 相关性检查逻辑（组合逻辑）
  - 用特殊的逻辑检查是否有任何前序指令会写任何当前译码指令的源操作数寄存器
  - 是：该指令/流水线停顿
  - 否：无需停顿… 没有流相关
- 优点：
  - 反相关和输出相关不会导致停顿
- 缺点：
  - 逻辑比计分板更复杂
  - 当我们设计的流水线结构更深更宽时逻辑会变得越发复杂

21

21

一旦检测出相关性

- 接下来该怎么做?



22

22

一旦检测出相关性

- 接下来该怎么做?
- 观察：两条指令之间的相关性会在相关数据产生之前被检测出来
- 选项 1：立即使相关指令停顿
- 选项 2：在必需时停顿相关指令 → 数据转发/旁路
- 选项 3：…

23

23

数据转发/旁路

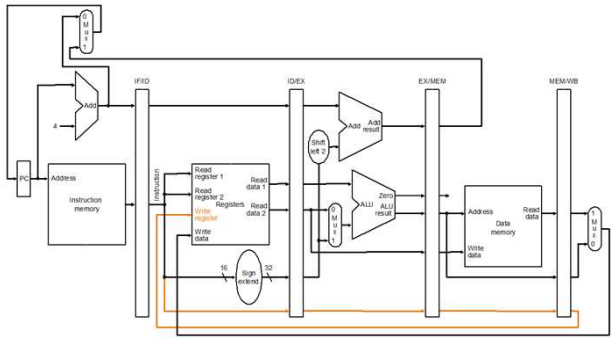
- 问题：消费者指令（产生相关者)不得不等待在译码阶段直到生产者指令将值写回寄存器堆
- 目的：不希望让流水线做不必要的停顿

24

24

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
-



25

25

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
- 目的：不希望让流水线做不必要的停顿
- 观察：消费者指令需要的数据可以直接由流水线上的前序阶段提供（不是只能够来自寄存器堆）
- 思路：增加额外的相关性检查逻辑和数据转发通路（总线）将生产者产生的值立即提供给消费者
- 好处：消费者可以在流水线上流动直到所需的值能够提供的点 → 减少停顿

26

26

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

27

27

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?

28

28

处理数据相关：
深入探索和实现方法

29

29

回顾：如何处理数据相关

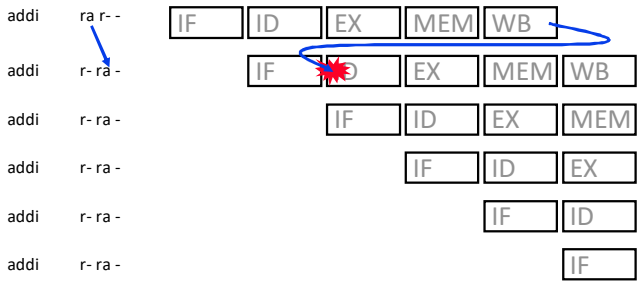
- 五种处理流相关的基本方法
  - 检测并等待直到值在寄存器堆中可以访问
  - 检测并转发/旁路数据给相关的指令
  - 检测并消除相关性（在软件层面）
    - 不需要硬件检测相关性
  - 预测 需要的值， “投机”执行，并且验证
  - 其它(细粒度多线程)
    - 不需要检测

30

30

处理写后读相关

- 下面的流相关导致5阶段流水线的冲突

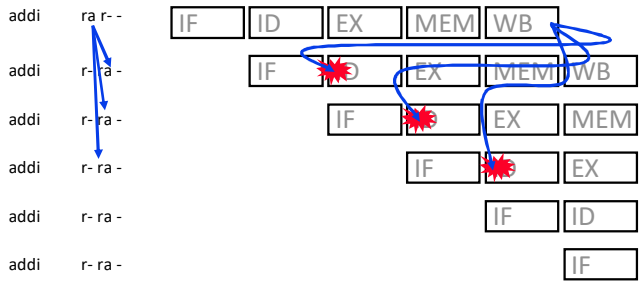


31

31

处理写后读相关

- 下面的流相关导致5阶段流水线的冲突

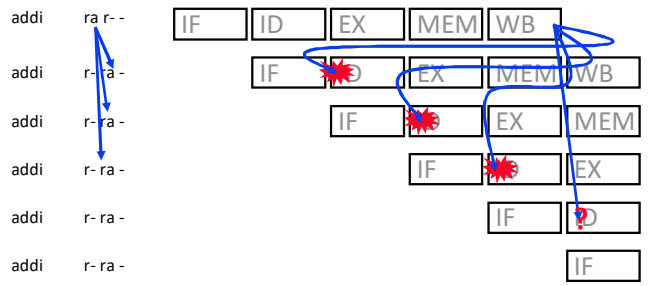


32

32

处理写后读相关

- 下面的流相关导致5阶段流水线的冲突



33

寄存器数据相关性分析

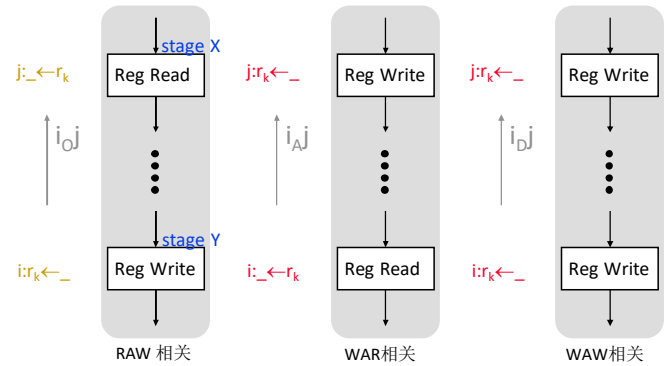
	R/I-Type	LW	SW	Br	J	Jr
IF						
ID	读 RF	读 RF	读 RF	读 RF		读 RF
EX						
MEM						
WB	写 RF	写 RF				

■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?

34

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

35

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				

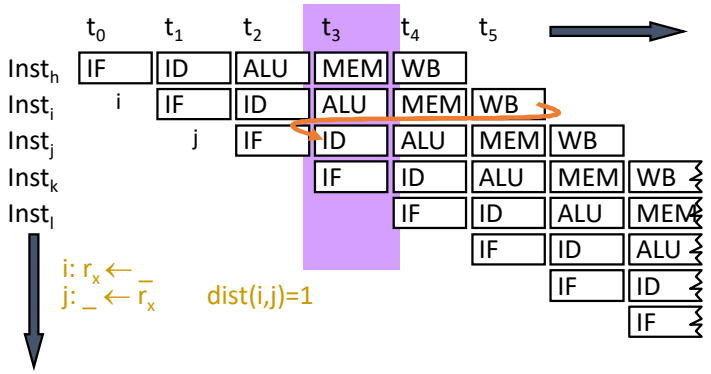
- 指令 I_A 和 I_B (I_A 先于 I_B执行) 有 RAW 相关的条件
  - I_B (R/I, LW, SW, Br or JR) 读 I_A (R/I or LW) 写的寄存器
  - $dist(I_A, I_B) \le dist(ID, WB) = 3$

WAW 和 WAR 相关?

内存数据相关?

36

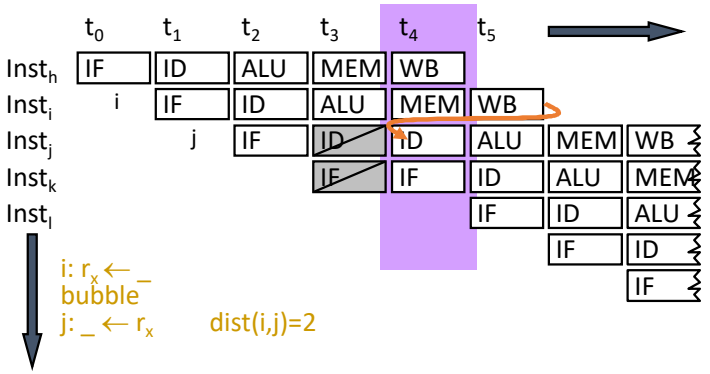
流水线停顿：解决数据相关



37

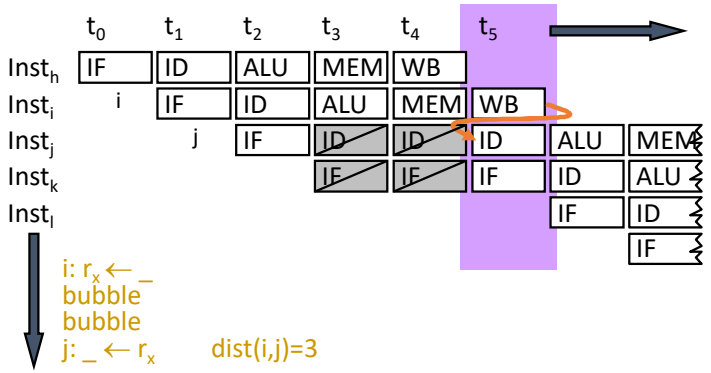
37

流水线停顿：解决数据相关



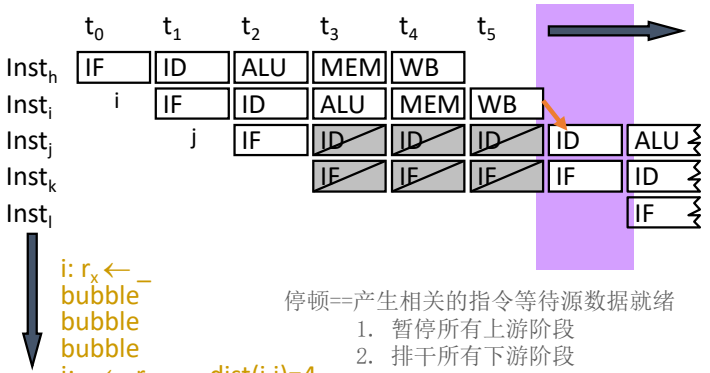
38

流水线停顿：解决数据相关



39

流水线停顿：解决数据相关

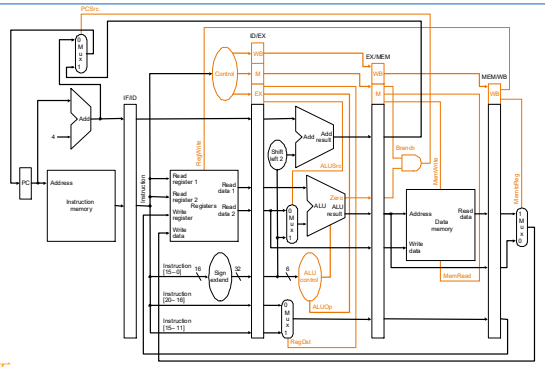


停顿==产生相关的指令等待源数据就绪

1. 暂停所有上游阶段
2. 排干所有下游阶段

40

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

41

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
  - $dist(I_A, I_B) \leq dist(ID, WB) = 3$
- 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，指令 I_B 需要停顿

42

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

43

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

44

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

45

45

数据转发(或数据旁路)

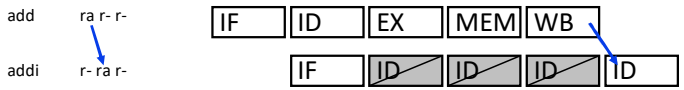
- 可以直观地将RF看作某种状态
  - “add rx ry rz”字面的意思就是分别从 RF[ry] 和 RF[rz] 取到值并把结果放进 RF[rx]
- 但是，RF只是计算抽象的一部分
  - “add rx ry rz”意味着 1. 得到前面指令的结果分别确定 RF[ry] 和 RF[rz] 的值，2. 直到另一条指令重新确定 RF[rx] 之前，指向RF[rx]的后序指令将使用这条指令的结果

46

46

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



47

47

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

48

48

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) ≤ 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

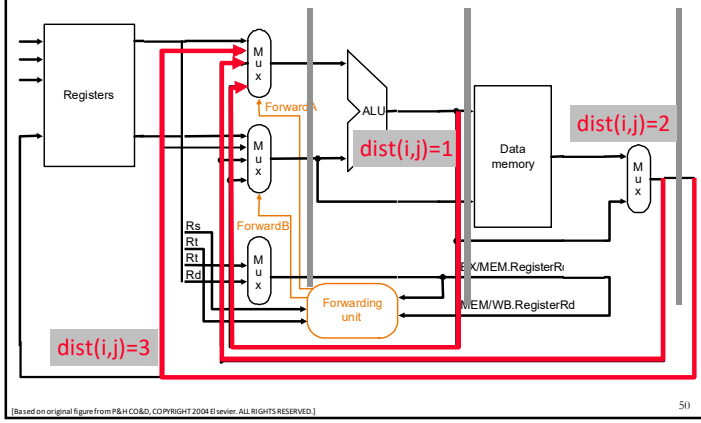
  ⇒ 从数据通路上而不是从RF中取回操作数

  ⇒ 当有多个未完成的操作数定义时取回最新产生的那一个

49

49

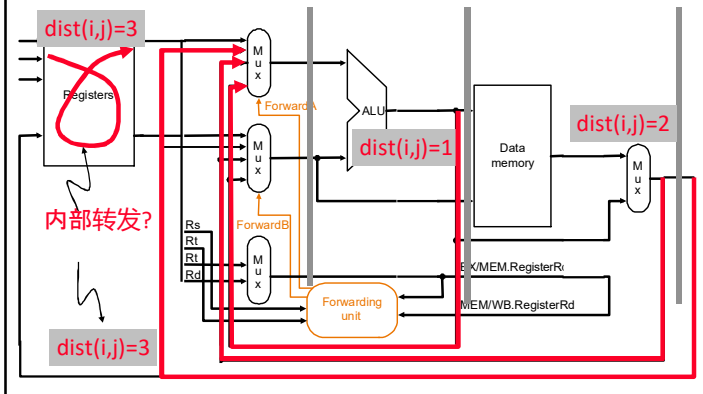
数据转发路径(v1)



50

50

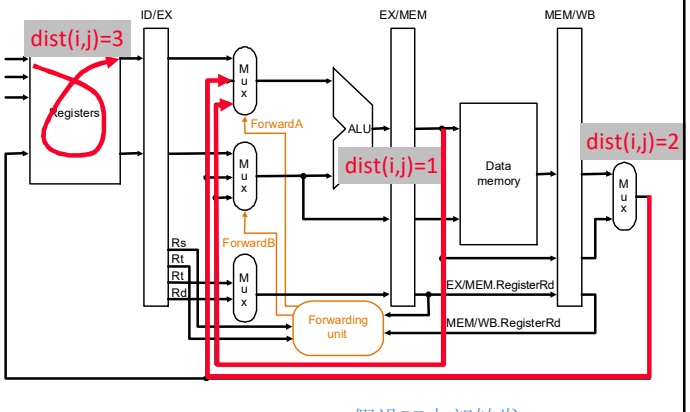
数据转发路径(v1)



51

51

数据转发路径(v2)



假设RF内部转发

52

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

为什么 use_rs( ) 没有出现在转发逻辑中?

53

53

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?

54

54

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

55

55

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:
```

56

56

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?

57

57

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

58

58

处理控制相关

59

59

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?

60

60

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

61

61

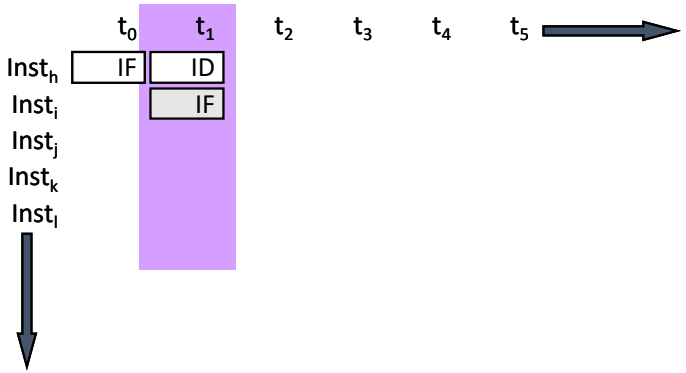
如何处理控制相关

- 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

停止取指直到下一个PC可用：这是好主意吗?

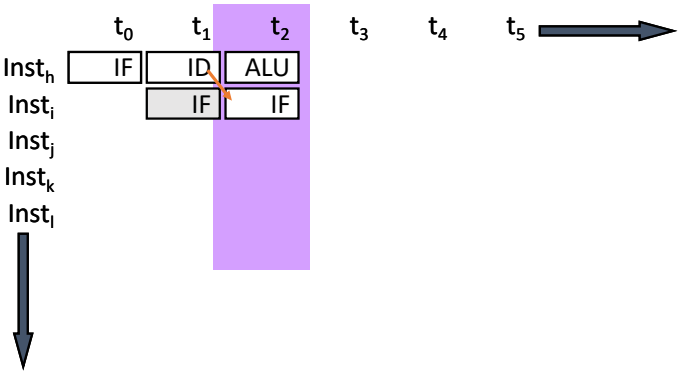


对于非控制分支指令及无条件分支指令可能是的!

63

63

停止取指直到下一个PC可用：这是好主意吗?

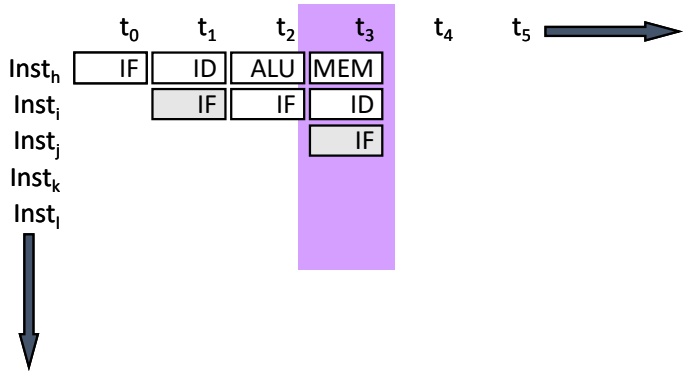


对于非控制分支指令及无条件分支指令可能是的!

64

64

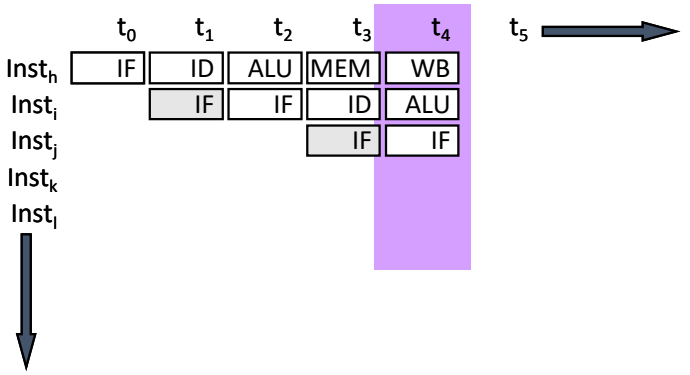
停止取指直到下一个PC可用：这是好主意吗?



对于非控制分支指令及无条件分支指令可能是的!

65

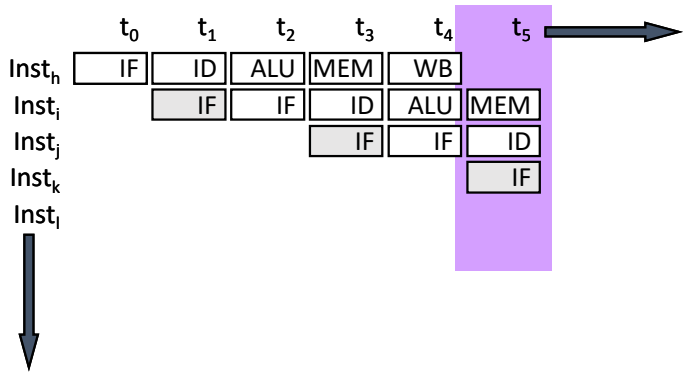
停止取指直到下一个PC可用：这是好主意吗?



对于非控制分支指令及无条件分支指令可能是的!

66

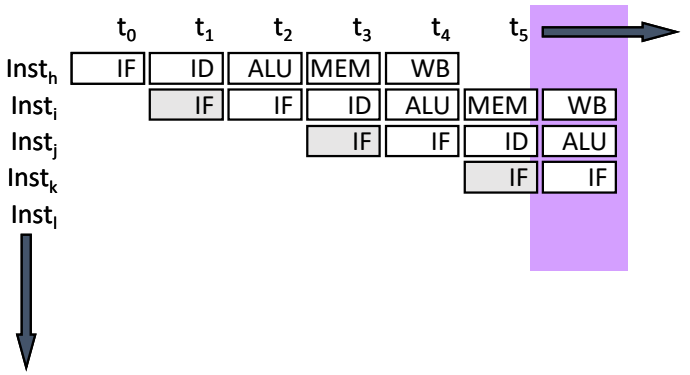
停止取指直到下一个PC可用：这是好主意吗?



对于非控制分支指令及无条件分支指令可能是的!

67

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!

68

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?

69

69

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…)

70

70

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

71

71

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

72

72