高等计算机体系结构
第九讲: 流水线的精确异常和动态调度

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
2021-04-23

1

提醒：作业

- 作业 3
  - 今天截止
  - 流水线1
- 作业 4
  - 今晚发布，5月7日截止
  - 流水线2

2

2

实验2-5

- 预计5月7日发布，7月11日截止

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

4

4

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

5

5

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

6

6

流水和精确异常: 保持连续的语义

7

7

回顾：多周期执行

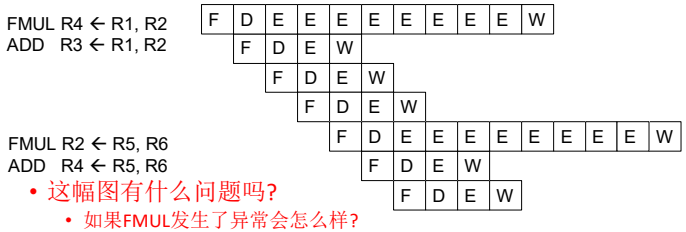
- 不是所有指令的“执行”时间都是一样长的
- 思路: 有多个不同的功能单元，会花费不同数量的时钟周期
  - 可以流水可以不流水
  - 能够使独立的指令在不同的功能单元上，在前序的长延时指令执行完毕之前就开始执行
- 程序序的保证和精确异常

8

8

回顾：流水线中的多周期执行

- 指令在执行阶段可能花费不同数量的时钟周期
  - 整型的 ADD vs. 浮点数的 MULT



- 这幅图有什么问题吗?
  - 如果FMUL发生了异常会怎么样?
  - ISA的连续语义是不保存的!

9

9

回顾：异常 vs. 中断

- 起因
  - 异常: 内部产生，作用于运行的线程
  - 中断: 外部产生，作用于运行的线程
- 处理的时机
  - 异常: 一旦检测出 (并且已经知道非投机)
  - 中断: 方便的时候
    - 除了非常高优先级的
      - 电源失效
      - 机器自检
- 优先级: 必须处理当前进程 (异常), 不一定 (中断)
- 处理的上下文: 进程 (异常), 系统 (中断)

10

10

回顾：精确异常/中断

- 当准备处理异常/中断时，体系结构状态应该是一致的

1. 所有之前的指令必须完全回收

2. 之后的指令一律不得回收

回收(Retire) = 提交(commit) = 执行完毕并且更新体系结构状态

11

11

回顾：为什么希望精确异常?

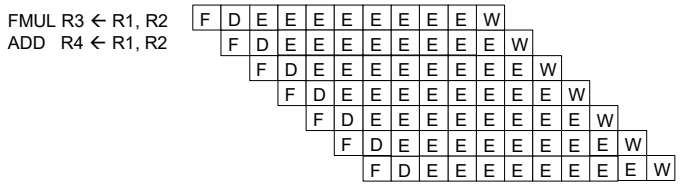
- 冯诺依曼模型ISA的语义
  - 冯诺依曼模型vs. 数据流
- 帮助软件调试
- 使异常恢复更容易, 比如页失效的恢复
- 使进程重启更容易
- 可以在软件中加入trap

12

12

回顾：流水线中确保精确异常

- 思路: 使每个操作花同样的时间



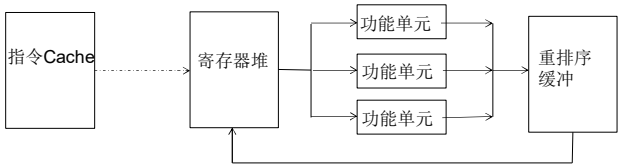
- 缺点
  - 访存操作会怎么样?
  - 每个功能单元100个时钟周期?

解决方案

- 重排序缓冲
- 历史缓冲
- 未来寄存器堆
- 检查点
- 推荐阅读
  - Smith and Plezskun, "Implementing Precise Interrupts in Pipelined Processors," IEEE Trans on Computers 1988 and ISCA 1985.
  - Hwu and Patt, "Checkpoint Repair for Out-of-order Execution Machines," ISCA 1987.

解决方案 I: 重排序缓冲 (ROB)

- 思路: 乱序执行指令, 产生体系结构状态可见的结果之前重排序
- 当指令译码时在ROB中预留一个条目
- 当指令执行完时，将结果写入ROB 中相应条目
- 当指令成为ROB中最旧的一条，并且已经执行完（没有异常）时，将结果移动到寄存器堆或者存储器



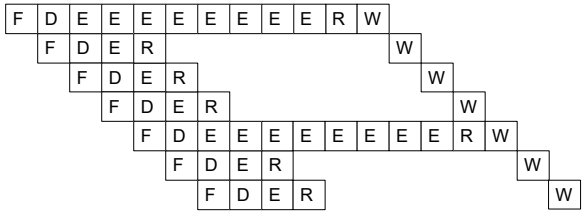
ROB 条目中有什么?

V	DestRegID	DestRegVal	StoreAddr	StoreData	PC	Valid bits for reg/data + control bits	Exc?

- 用有效位表征结果是否准备好

ROB: 独立操作

- 结果首先写入ROB, 在指令提交时写入寄存器堆



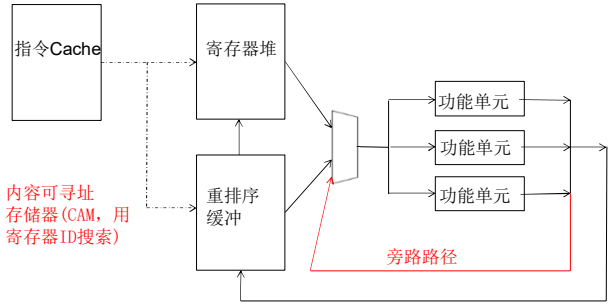
- 如果一个后续的操作需要的值在ROB中会怎么样?
  - 读ROB的同时读寄存器堆。 如何做到?

17

17

ROB: 如何访问?

- 一个寄存器的值可以在寄存器堆里，也可以在ROB中(还可以在旁路/转发路径上)



18

18

简化ROB访问

- 思路: 使用间接寻址的思想
- 先访问寄存器堆
  - 如果寄存器无效，寄存器堆存储ROB中包含（或将要包含）寄存器值的条目ID
  - 将寄存器映射到ROB条目
- 再访问ROB

- ROB条目中有什么?

V	DestRegID	DestRegVal	StoreAddr	StoreData	PC/IP	Control/valid bits	Exc?

  - 还能再简化一点吗?

19

19

用ROB为寄存器重命名

- 输出相关和反相关不是真正的相关
  - 为什么? 相同编号的寄存器互相之间并没有什么操作
  - **存在这样的相关是由于ISA允许的寄存器ID数（名字）缺乏**
- 将要保存结果的寄存器根据ROB条目重命名
  - 寄存器ID → ROB条目ID
  - 体系结构寄存器ID → 物理寄存器ID
  - 重命名后, ROB 条目ID被用来指向寄存器

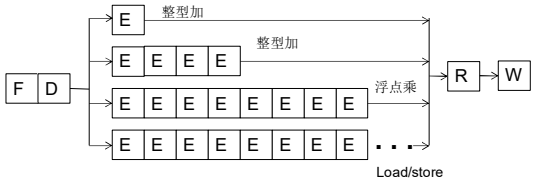
- 这将消除输出相关和反相关
  - 造成一个有大量寄存器的假象

20

20

使用ROB的按序执行流水线

- 译码 (D): 访问寄存器堆/ROB, 分配ROB条目, 检查指令是否可以执行, 是则**分发**指令
- 执行 (E):指令可以完全的乱序执行
- 完成 (R): 将结果写入ROB
- 回收/提交 (W): 检查异常; 如果没有, 将结果写入体系结构寄存器堆或存储器; 如果有, 清空流水线并启动异常处理程序
- 按序分发/执行, 乱序完成, 按序回收



21

21

ROB的Tradeoff

- 好处
  - 用很简单的概念来支持精确异常
  - 可以消除“虚假的”相关
- 坏处
  - 有可能需要访问ROB以获得尚未写入寄存器堆的结果
    - CAM 或间接寻址→ 增加延时和复杂性
- 其它希望消除上述缺陷的解决方案
  - 历史缓冲
  - 未来寄存器堆
  - 检查点

22

22

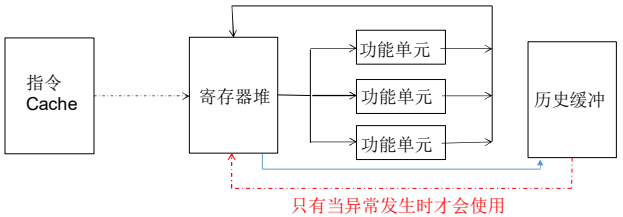
解决方案 II: 历史缓冲 (HB)

- 思路: 指令执行完成后更新寄存器堆, 但是当有异常发生时撤销那些更新（UNDO）
- 当指令译码时，预留一个HB条目
- 当指令执行完毕时，将目标地址中的旧值存在HB里
- 当指令是HB中最旧的一条并且没有发生异常/中断，丢弃该HB条目
- 当指令是HB中最旧的一条并且有异常需要处理，将HB中保存的旧值依次写回到体系结构状态

23

23

历史缓冲



- 好处:
  - 寄存器堆中保有最新的值，HB的访问不在关键路径上
- 坏处:
  - 需要读目的寄存器的旧值
  - 在异常时需要回滚HB→ 增加异常/中断的处理时延

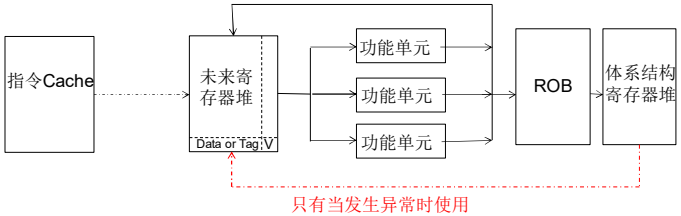
24

24

解决方案 III: 未来寄存器堆(FF) + ROB

- 思路: 维护两个寄存器堆 (投机的和体系结构的)
  - 体系结构的寄存器堆: 按程序序更新以获得精确异常
    - 使用ROB来保证按序的更新
  - 未来的寄存器堆: 一条指令执行完毕后立即更新(如果这条指令是最新的一条写寄存器堆的指令)
- FF 用于快速访问最近的寄存器值(投机的状态)
  - 前端寄存器堆
- 体系结构寄存器堆用于当发生异常时的状态恢复 (体系结构状态)
  - 后端寄存器堆

未来寄存器堆



■好处

❑不需要从ROB中读取值 (不需要CAM或者间接寻址)

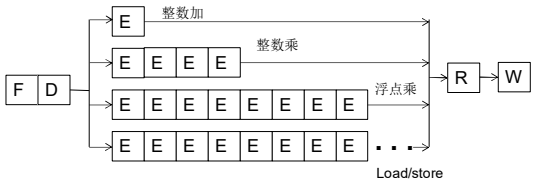
■坏处

❑多个寄存器堆

❑发生异常时需要从一个堆向另一个堆复制数据

有FF和ROB的按序执行流水线

- 译码 (D): 访问FF, 在ROB中分配条目, 检查指令是否可以执行，是则**分发**指令
- 执行 (E): 指令可以完全的乱序执行
- 完成 (R): 将结果写入ROB和FF
- 回收/提交 (W): 检查异常; 如果没有, 将结果写入体系结构寄存器堆或者存储器; 如果有, 清空流水线并启动异常处理程序
- 按序分发/执行, 乱序完成, 按序回收



流水线中检查和处理异常

- 当最旧的准备回收的指令被检查出导致了异常, 控制逻辑
  - 恢复体系结构状态(寄存器堆, IP/PC, 存储器)
  - 清空流水线中所有比该指令新的指令
  - 保存IP/PC和寄存器堆 (ISA指定)
  - 重定向取指引擎指向异常处理子程序
    - 异常向量

流水线的问题: 分支预测错误

- 分支预测错误相当于一个“异常”
  - 除非它对软件不可见
- 分支预测错误如何恢复?
  - 与异常处理类似，不同之处在于不用等到分支指令成为最旧的指令
  - 三种状态恢复方案都可以用
- 异常和分支预测错误的区别?
  - 分支预测错误比异常要普遍的多→需要快速的状态恢复以尽可能减小预测错误对性能的影响

状态恢复的延迟影响

- 异常服务延迟
- 中断服务延迟
- 为预测错误的分支之后的指令提供正确数据的延迟
- 就恢复延迟而言，三种状态保持的方法是否/如何做到合理?
  - ROB
  - HB
  - FF

分支状态恢复的动作和延迟

- ROB
  - 等待直到分支是最旧的一条指令
  - 清空后序流水线
- HB
  - 从历史缓冲最尾回滚撤销分支之后的所有指令，并且向寄存器堆恢复旧值
- FF
  - 等待直到分支是最旧的一条指令
  - 从体系结构寄存器堆向未来寄存器堆复制数据，并且清空流水线

能更好吗?

- 目标: 恢复前端状态 (未来寄存器堆)，这样可以使分支后正确的下一条指令能够在分支预测错误被解决后立即执行
- 思路: 当分支取指时对前端寄存器状态设立检查点，同时对比分支旧的指令结果保持状态更新

- Hwu and Patt, “Checkpoint Repair for Out-of-order Execution Machines,” ISCA 1987.

检查点

- 当分支译码时
  - 复制未来寄存器堆并关联到分支
- 当指令产生寄存器值
  - 所有晚于该指令的未来寄存器堆检查点更新到该状态
- 当分支预测错误被检测出来
  - 当分支预测错误解决时，恢复被错误预测的分支的未来寄存器堆检查点
  - 清空所有晚于分支的指令
  - 释放所有晚于分支的检查点

33

33

检查点

■好处?

■坏处?

34

34

寄存器 vs. 存储器

- 到目前为止, 主要考虑的是寄存器的情况
- 存储器会怎么样?
- 寄存器和存储器之间有什么根本的不同?
  - 寄存器的相关是静态可知的 – 存储器的相关是动态决定的
  - 寄存器的状态空间小 – 存储器的状态空间大
  - 寄存器状态对其它线程/处理器不可见 – 存储器状态在线程/处理器之间是共享的 (共享存储多处理器)

35

35

流水线的问题: store

- 处理存储器操作的乱序执行
  - 撤销存储器写比撤销寄存器写要难得多。 为什么?
  - 一种思路: 在ROB中保存store的地址/数据
    - Load指令如何发现这些数据?
  - 存储/写入缓冲: 类似于ROB, 但是只用于store指令
    - 按程序序尚未提交的store指令列表
    - 当store指令译码时: 分配一个存储缓冲的条目
    - 当store的地址和数据都可用时: 将地址和数据记录在存储缓冲的条目中
    - 当store是流水线中最旧的指令时: 更新存储器地址(cache)和存储的数据

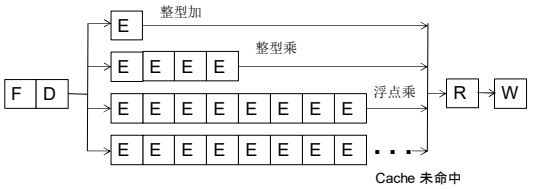
36

36

乱序执行
(动态指令调度)

37

按序执行的流水线



- 问题: 一个真正的数据相关会使流水线停止分发新的指令到功能(执行)单元
- 分发: 将指令送入功能单元的动作

38

38

能不能更好?

- 下面两段代码有何共同之处?

```
IMUL  R3 ← R1, R2          LD    R3 ← R1 (0)
ADD   R3 ← R3, R1          ADD   R3 ← R3, R1
ADD   R1 ← R6, R7          ADD   R1 ← R6, R7
IMUL  R5 ← R6, R8          IMUL  R5 ← R6, R8
ADD   R7 ← R3, R5          ADD   R7 ← R3, R5
```

- 答案: 第一个ADD指令使整个流水线停顿!
  - 由于源寄存器不可用使得ADD不能分发
  - 后续的**独立指令**也无法执行

- 上面两段代码的区别是什么?
  - 答案: Load 延迟是不确定的(在运行时才知道)
  - 这会产生什么影响?（考虑编译器和微体系结构）

39

39

防止分发停顿

- 有很多方法
- 我们已经讲过三种:
  - 1.
  - 2.
  - 3.
- 上述三种方法的缺点是什么?

- 还有其他方法吗?
  - 实际上, 我们介绍过基本的思想
    - 数据流: 当输入准备好时取指和发射指令
  - 问题: 按序分发 (调度, 或执行)
  - 解决方案: 乱序分发 (调度, 或执行)

40

40

乱序执行(动态调度)

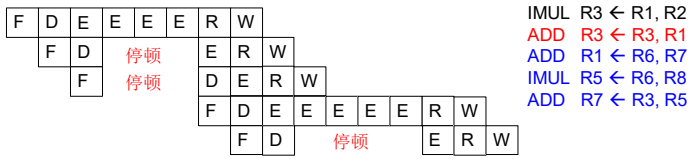
- 思路: 让相关的指令离独立的指令远一点（分离）
  - 相关指令的“休息区”: 保留站
- 监控休息区中每条指令的源“值”
- 当一条指令的所有源“值”可用，则分发该指令执行
  - 指令按数据流序 (非控制流) 分发
- 好处:
  - 延迟容忍: 允许独立指令在有长延迟操作时执行和完成

41

41

按序 vs. 乱序分发

- 按序分发+精确异常:



- 乱序分发+精确异常:

- 16 vs. 12 时钟周期

42

42

乱序执行的支撑条件

1. 需要在“值”的生产者和消费者之间建立连接
   - 寄存器重命名: 给每一个“值”关联一个“标签(tag)”
2. 需要缓冲指令直到它们准备好执行
   - 寄存器重命名之后向保留站插入指令
3. 指令需要持续跟踪源“值”是否可读
   - 当“值”被生产出来，广播“标签(tag)”
   - 保留站中的指令对比自己的“源标签”和广播标签 → 如果匹配, 则源值准备好
4. 当一条指令的所有源值都准备好，需要分发指令到它的功能执行单元 (FU)
   - 当所有源值准备好，指令唤醒
   - 如果有多条指令被唤醒, 需要为每个FU选择一条指令

43

43

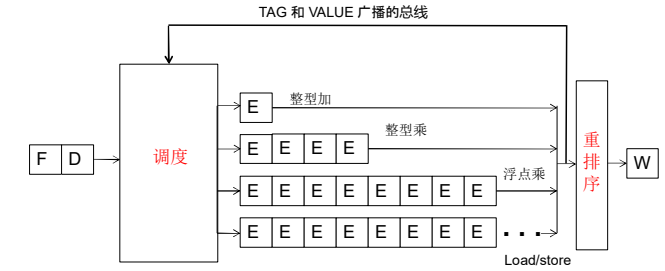
Tomasulo算法

- Robert Tomasulo发明了带寄存器重命名的乱序执行
  - 用于IBM 360/91的浮点运算单元
  - **阅读:** Tomasulo, “An Efficient Algorithm for Exploiting Multiple Arithmetic Units,” IBM Journal of R&D, Jan. 1967.
- 跟今天的方法有什么主要区别?
  - 精确异常: IBM 360/91不具备
  - Patt, Hwu, Shebanow, “HPS, a new microarchitecture: rationale and introduction,” MICRO 1985.
  - Patt et al., “Critical issues regarding HPS, a high performance microarchitecture,” MICRO 1985.
- 大多数高性能处理器使用的都是它的变种
  - 最初是在 Intel Pentium Pro, AMD K5
  - Alpha 21264, MIPS R10000, IBM POWER5, IBM z196, Oracle UltraSPARC T4, ARM Cortex A15

44

44

现代流水线的两个“驼峰”

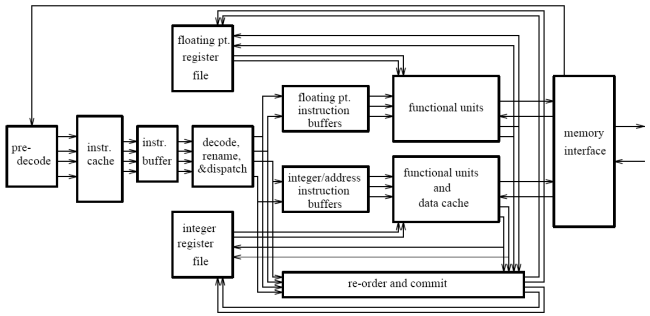


- 驼峰 1: 保留站(调度窗口)
- 驼峰 2: 重排序(ROB, 又叫指令窗口或者动态窗口)

45

45

乱序执行处理器一般的组织形式

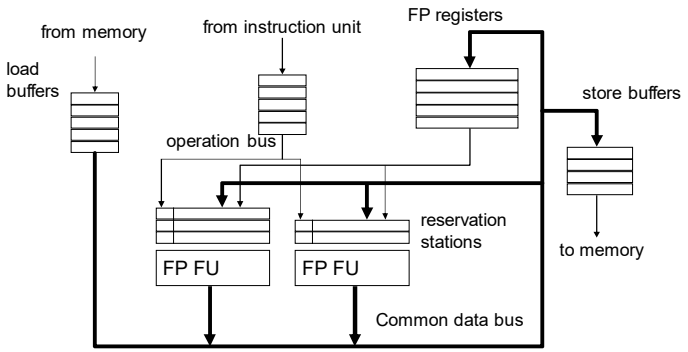


- Smith and Sohi, “The Microarchitecture of Superscalar Processors,” Proc. IEEE, Dec. 1995.

46

46

Tomasulo的机器: IBM 360/91



47

47

回顾：寄存器重命名

- 输出相关和反相关不是真正的相关
  - 为什么? 相同编号的寄存器互相之间并没有什么操作
  - **存在这样的相关是由于ISA允许的寄存器ID数（名字）缺乏**
- 寄存器ID被重命名为拥有该寄存器值的保留站条目ID
  - 寄存器ID → 保留站条目 ID
  - 体系结构寄存器 ID → 物理寄存器 ID
  - 重命名后, 保留站条目ID被用来指向寄存器
- 消除了输出相关和反相关
  - 即使ISA只有少量寄存器，也会产生与拥有大量寄存器差不多的性能影响

48

48

Tomasulo算法: 重命名

- 寄存器重命名表 (寄存器别名表)

	tag	value	valid?
R0			1
R1			1
R2			1
R3			1
R4			1
R5			1
R6			1
R7			1
R8			1
R9			1

Tomasulo算法

- 如果保留站在重命名之前可用
  - 指令 + 重命名的操作数 (源值/标识) 插入保留站
- 如果不可用则停顿
- 在保留站里, 每条指令:
  - 关注公共数据总线 (CDB)上数据源的标签
  - 当看到“自己的”标签，获取源值并保存在保留站
  - 当所有操作数可用, 指令准备分发
- 当指令准备好，向功能单元分发指令
- 指令在功能单元中完成操作之后
  - 进行 CDB 仲裁
  - 将打了标签的值送上CDB (标签广播)
  - 寄存器堆连接到 CDB
    - 寄存器包含一个标签，标记了最近写寄存器的写入者
    - 如果寄存器堆中的标签和广播的标签匹配，将广播的值写入寄存器 (并且置有效位)
  - 回收重命名标签

例子

MUL R3 ← R1, R2
ADD R5 ← R3, R4
ADD R7 ← R2, R6
ADD R10 ← R8, R9
MUL R11 ← R7, R10
ADD R5 ← R5, R11

F	D	E	W

- 假设 ADD (执行阶段4个时钟周期流水), MUL (执行阶段6个时钟周期流水)
- 假设只有一个加法器和乘法器
- 需要多少时钟周期
  - 非流水线机器
  - 非精确异常的按序分发流水线 (无转发和全转发)
  - 非精确异常的乱序分发流水线 (全转发)

例子(续1)

MUL R3 ← R1, R2
ADD R5 ← R3, R4
ADD R7 ← R2, R6
ADD R10 ← R8, R9
MUL R11 ← R7, R10
ADD R5 ← R5, R11

可以有多个时钟周期

F	D	E	W

MUL 6个时钟周期流水执行
ADD 4个时钟周期流水执行

没有数据转发时一共需要多少时钟周期? 采用scoreboarding

F	D	1	2	3	4	5	6	W																						
	F	D	-	-	-	-	-	-	D	1	2	3	4	W																
		F	-	-	-	-	-	-	-	D	1	2	3	4	W															
										F	D	1	2	3	4	W														
											F	D	-	-	-	-	D	1	2	3	4	5	6	W						
												F	-	-	-	-	-	D	-	-	-	-	-	-	D	1	2	3	4	W

31个时钟周期

例子(续2)

MUL R3 ← R1, R2
ADD R5 ← R3, R4
ADD R7 ← R2, R6
ADD R10 ← R8, R9
MUL R11 ← R7, R10
ADD R5 ← R5, R11

可以有多个时钟周期

F	D	E	W

MUL 6个时钟周期流水执行
ADD 4个时钟周期流水执行

有数据转发时一共需要多少时钟周期?

F	D	1	2	3	4	5	6	W														
	F	D	-	-	-	-	-	1	2	3	4	W										
		F	-	-	-	-	-	D	1	2	3	4	W									
								F	D	1	2	3	4	W								
									F	D	-	-	-	1	2	3	4	5	6	W		
										F	-	-	-	D	-	-	-	-	-	1	2	3	4	W

25个时钟周期

53

53

例子(续3)

MUL R3 ← R1, R2
ADD R5 ← R3, R4
ADD R7 ← R2, R6
ADD R10 ← R8, R9
MUL R11 ← R7, R10
ADD R5 ← R5, R11

可以有多个时钟周期

F	D	E	W

MUL 6个时钟周期流水执行
ADD 4个时钟周期流水执行

F	D	1	2	3	4	5	6	W														
	F	D	-	-	-	-	-	1	2	3	4	W										
		F	D	1	2	3	4	W														
			F	D	1	2	3	4	W													
				F	D	-	-	-	1	2	3	4	5	6	W							
					F	D	-	-	-	-	-	-	-	-	1	2	3	4	W			

20个时钟周期

Tomasulo算法+全转发

54

54

如何做到的?

寄存器别名表

	V	Tag	Value

“写入者”的标签

	源1			源2			源1			源2			
	V	Tag	Value	V	Tag	Value	V	Tag	Value	V	Tag	Value	
加法器的保留站 a							X						乘法器的保留站
b							Y						
c													
d													

+ Value Tag

* Value Tag

假设加法器和乘法器各自拥有独立的总线

55

55

周期 0

	V	Tag	Value
R1	1	...	1
	1		2
	.		.
	.		.
	.		.
	.		.
R11	1	...	11

寄存器别名表初始内容

所有保留站内容无效

56

56

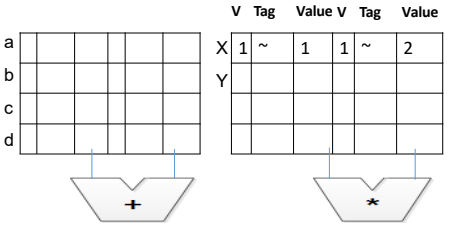
周期 2

MUL R3 ← R1, R2

— 从RAT中读取源
— 向RAT中写入目标（重命名其目标）
→ 分配一个保留站条目
→ 为它的目标寄存器分配一个标签
— 向分配的保留站条目放入它的源

周期2结束时：

	V	Tag	Value
R1	1	~	1
R2	1	~	2
R3	0	X	~
R4	1	~	4
.	.	.	.
R11	1	~	11



— MUL在X条目准备好执行（如果有多条指令同时准备好该怎么办？）
在保留站X条目里的两个源都有效

57

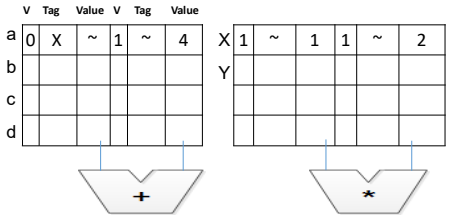
57

周期3

→ MUL X条目开始执行
→ ADD R5 ← R3，R4 完成重命名并且存入加法器保留站

周期3结束时：

	V	Tag	Value
R1	1	~	1
R2	1	~	2
R3	0	X	~
R4	1	~	4
R5	0	a	~
R6	1	~	6
.	.	.	.
R11	1	~	11



— ADD 在a条目无法ready，因为它的一个源没有准备好
→ 它等待带有X标签的广播(MUL X条目)

标签需要关联到生产者的RS条目吗？

不需要，标签只是参与传递的数值的标记，它确保数据流形式的数值传递；RS是在等待准备完成期间，指令的存放地点

58

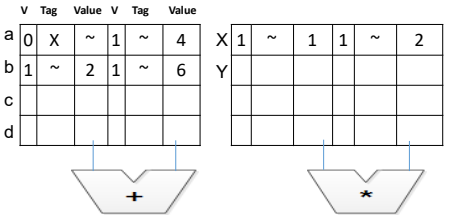
58

周期 4

— ADD R7← R2，R6 重命名并加入保留站

周期4结束时：

	V	Tag	Value
R1	1	~	1
R2	1	~	2
R3	0	X	~
R4	1	~	4
R5	0	a	~
R6	1	~	6
R7	0	b	~
.	.	.	.
R11	1	~	11



— ADD 在b条目准备好执行(两个源都准备好)
— 在周期5，它将会进入加法器执行，与程序序不同
→它先于a条目中的加法运算执行

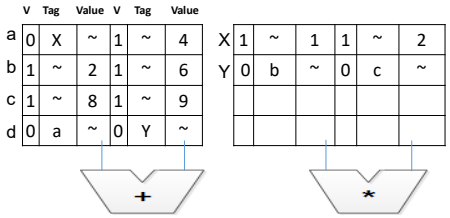
59

59

周期 7

周期7结束时：

	V	Tag	Value
R1	1	~	1
R2	1	~	2
R3	0	X	~
R4	1	~	4
R5	0	d	~
R6	1	~	6
R7	0	b	~
R8	1	~	8
R9	1	~	9
R10	0	c	~
R11	0	Y	~



所有6条指令被重命名
注意R5

60

60

周期 8

— MUL 的 X 和 ADD 的 b，广播它们的标签和值

— 保留站的条目等待这些标签、获取值并相应地设置有效位
→ 硬件需要什么结构来实现这些？(CAM 存储广播的标签)

— RAT的条目同样等待这些标签、获取值并相应地设置有效位

61

61

练习：带精确异常的例子

```
MUL  R3 ← R1, R2
ADD  R5 ← R3, R4
ADD  R7 ← R2, R6
ADD  R10 ← R8, R9
MUL  R11 ← R7, R10
ADD  R5 ← R5, R11
```

- 假设 ADD (4个时钟周期执行), MUL (6个时钟周期执行)
- 假设只有一个加法器和乘法器
- 需要多少时钟周期
  - 非流水线机器
  - 带ROB的按序分发流水线 (无转发和全转发)
  - 带ROB的乱序分发流水线 (全转发)

62

62

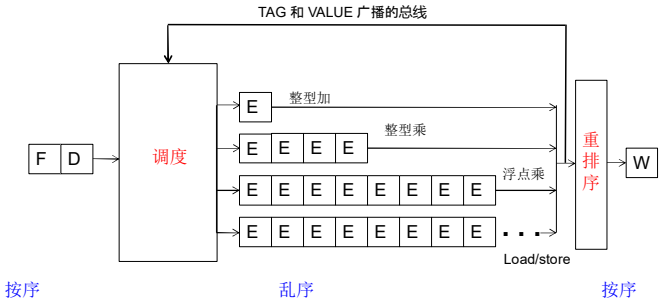
带精确异常的乱序执行

- 思路: 在指令提交体系结构状态之前利用重排序缓冲对指令进行重排序
- 当指令执行完成时更新寄存器别名表 (本质上是个未来寄存器堆)
- 当指令执行完成并且是最旧的一条指令时更新体系结构寄存器堆

63

63

带精确异常的乱序执行



- 驼峰 1: 保留站(调度窗口)
- 驼峰 2: 重排序(ROB, 又叫指令窗口或者动态窗口)

64

64

乱序执行的支撑条件

1. 在“值”的生产者和消费者之间建立连接
   - 寄存器重命名: 给每一个“值”关联一个“标签(tag)”
2. 缓冲指令直到它们准备好执行
   - 重命名之后向保留站插入指令
3. 指令持续跟踪源“值”是否可读
   - 当“值”被生产出来，广播“标签(tag)”
   - 指令对比自己的“源标签”和广播标签→ 如果匹配, 则源值准备好
4. 当一条指令的所有源值都准备好，分发给指令到它的执行单元 (FU)
   - 唤醒并选择/调度指令

65

65

乱序执行概念小结

- 寄存器重命名消除错误的相关, 建立了生产者和消费者的联系
- 缓冲使得流水线可以执行独立的操作以保持流水
- 标签广播使得指令之间能够交互生产出的值
- 唤醒和选择保证了乱序的分发

66

66

乱序执行: 受限的数据流

- 乱序引擎动态的构建一个程序块的数据流图
  - 哪个程序块?
- 数据流受限于指令窗口
  - 指令窗口: 所有已经译码但是尚未提交（回收）的指令
- 能对整个程序乱序执行吗?
- 为什么最好能?
- 换句话说，如何能有一个大的指令窗口?
- 用Tomasulo算法就能高效地处理乱序执行吗?

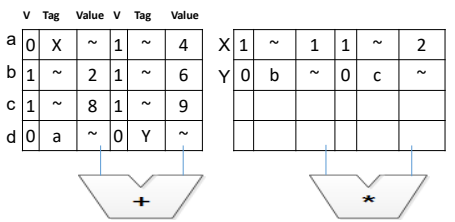
67

67

前面的例子：周期7结束时RAT和RS的状态

周期7结束时：

	V	Tag	Value
R1	1	~	1
R2	1	~	2
R3	0	X	~
R4	1	~	4
R5	0	d	~
R6	1	~	6
R7	0	b	~
R8	1	~	8
R9	1	~	9
R10	0	c	~
R11	0	Y	~



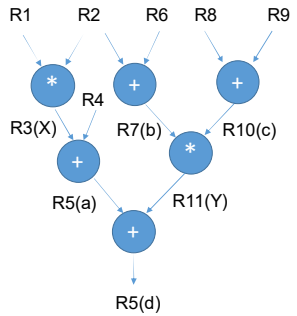
68

68

数据流图

```
MUL  R3 ← R1, R2   (X)
ADD  R5 ← R3, R4   (a)
ADD  R7 ← R2, R6   (b)
ADD  R10 ← R8, R9  (c)
MUL  R11 ← R7, R10 (Y)
ADD  R5 ← R5, R11  (d)
```

节点：指令执行的操作
弧： Tomasulo算法中的标签



69

69

受限的数据流

- 乱序执行的机器是“受限的数据流” 机
  - 基于数据流的执行被局限在微体系结构层
  - ISA 仍然是基于冯诺依曼模型的(顺序执行)
- 回顾数据流模型(ISA 层):
  - 数据流模型: 指令的取指和执行按照数据流的序
  - 操作数准备好
  - 没有指令指针(程序计数器)
  - 指令的序由数据流的相关性决定
    - 每条指令指明“谁”是结果的接收者
    - 当所有操作数准备好，指令就可以“发射”

70

70

思考

- 为什么乱序执行是有益的?
  - 如果所有的操作都占用1个时钟周期会怎么样?
  - 延迟容忍: 乱序执行能够通过并发执行独立的操作容忍多周期操作的延迟
- 如果一条指令要花费500个周期会怎么样?
  - 需要多大的指令窗口才能保证持续的译码?
  - 乱序执行能够容忍多少个周期的延迟?
  - 是什么限制了Tomasulo算法的延迟容忍的可扩展性?
    - 动态/指令窗口大小: 由寄存器堆、调度窗口、重排序缓冲等决定

71

71

回顾：寄存器 vs. 存储器

- 到目前为止, 我们考虑的都是指令之间基于寄存器的值的通信
- 存储器会怎么样?
- 寄存器和存储器之间有什么根本的不同?
  - 寄存器的相关是静态可知的 – 存储器的相关是动态决定的
  - 寄存器的状态空间小 – 存储器的状态空间大
  - 寄存器状态对其它线程/处理器不可见 – 存储器状态在线程/处理器之间是共享的(共享存储多处理器)

72

72

处理存储相关性(I)

- 乱序执行的机器中需要遵从存储的相关性
  - 需要在提供高性能的同时做到这一点
- 观察和问题:存储的地址直到load/store的执行阶段才能够获得
- 结果 1: 重命名存储地址很困难
- 结果 2: 决定load/store的相关或者独立需要在它们执行之后处理
- 结果 3: 当一个load/store的地址已经准备好, 可能同时会有新的/旧的load/store尚未确定地址

73

73

处理存储相关性(II)

- 什么时候可以在乱序执行引擎中调度一条load指令?
  - 问题: 一条新的load指令的地址比一条旧的store指令的地址先准备好
  - 被称为存储违例消解问题或未知地址问题
- 方法
  - 保守: 停顿 load 直到所有之前的 store 计算出它们的地址(或者甚至提交)
  - 积极: 假设 load 独立于地址未知的 store， 立即调度 load
  - 智能: 预测 (使用更复杂的预测器) load 是否与未知地址的 store 相关

74

74