高等计算机体系结构
第四讲: 多周期和流水线

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
  - 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

2

多周期微体系结构

- 目标：使每一条指令的执行只（大致）花费它该花费的时间
- 思路
  - 时钟周期的决定独立于指令处理时间
  - 每条指令需要花费多少时钟周期
    - 一条指令执行过程中会有多次状态转换
    - 每条指令的状态变换是不同的

3

回顾： "处理指令"的步骤

- ISA 抽象地说明给定一条指令和A，A'应该是什么
  - 定义一个抽象的有限态机
    - 状态 = 程序员可见的状态
    - 次态逻辑 = 指令执行的规范
  - 从 ISA 的视角，指令执行的过程中A和A'之间没有"中间状态"
    - 每条指令对应一个状态转换
- 微体系结构实现 A 向 A'的转换
  - 有很多种实现方式的选择
  - 我们可以加入程序员不可见的状态来优化指令执行的速度：每条指令有多个状态转换
    - 选择 1: A → A'（在一个时钟周期内完成 A 到 A' 的转换）
    - 选择 2: A → A+MS1 → A+MS2 → A+MS3 → A'(使用多个时钟周期完成 A 到 A' 的转换)

4

多周期微体系结构

AS = 指令执行之前程序员可见的体系结构状态

第1步：在一个时钟周期内处理一部分指令

第2步：在下一个时钟周期内处理一部分指令

…

AS' =指令执行之后程序员可见的体系结构状态

5

5

多周期设计的好处

- 关键路径设计
  - 可以独立地针对每条指令的最糟糕情况来优化关键路径
- 基本(典型)设计
  - 可以通过优化执行“重要”指令（占用大量执行时间）所需的状态数来达到需要的效果
- 平衡设计
  - 不需要提供比实际需求更多的资源或能力
    - 一条指令需要多次使用资源“X”并不意味着需要多个“X”
    - 使硬件更高效：一条指令可以多次***重用***硬件部件

6

6

性能分析

- 指令执行时间
  - {CPI} x {clock cycle time}
- 程序执行时间
  - 所有指令的[{CPI} x {clock cycle time}]之和
  - {指令数} x {平均 CPI} x {clock cycle time}
- 单周期微体系结构的性能
  - CPI = 1
  - Clock cycle time 长
- 多周期微体系结构的性能
  - CPI = 每条指令不同
    - 平均 CPI → 希望能很小
  - Clock cycle time 短

有两个独立的自由度可以优化

7

7

CPI vs. 主频

- CPI vs. 时钟周期长度
- 互相矛盾
  - 对一条指令来说，减少一个就会增加另一个
  - 为什么?
- 多条指令并发处理可以使平均CPI被平摊/减小
  - 同一个时钟周期被用来处理多条指令
  - 例如：流水线，超标量等

8

8

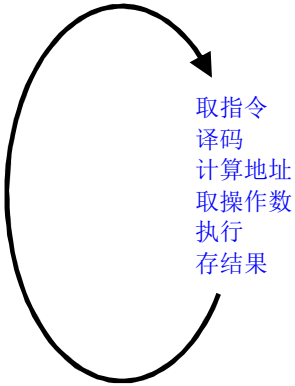
如何实现多周期?

- Maurice Wilkes, "The Best Way to Design an Automatic Calculating Machine," Manchester Univ. Computer Inaugural Conf., 1951.
- 微码/微程序控制的概念
- 实现
  - 可以按照描述状态之间序列的有限状态机来实现"指令处理"的步骤，最终状态机回到"取指令"状态
  - 状态由控制信号推定
  - 下一个状态的控制信号由当前状态决定

9

9

指令执行周期



10

10

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

11

11

微程序控制相关术语

- 与当前状态相关的控制信号
  - 微指令
- 从一个状态过渡到另一个状态的动作
  - 决定下一个状态以及下一个状态的微指令
  - 微序列（生成）
- 控制存储（器）为每一个可能的状态存储控制信号
  - 为整个有限状态机存储微指令
- 微序列（控制）器决定下一个时钟周期（下一个状态）将会用到的控制信号集合

12

12

在一个时钟周期里发生了什么?

- 对当前状态控制的控制信号(微指令)
  - 在数据通路中推进
  - 为下一个周期生成控制信号 (微指令)
- 数据通路和微序列器并发操作
- 问题: 为什么不在当前周期生成当前周期需要的控制信号?
  - 会使时钟周期延长
  - 为什么?

13

13

关于控制存储的几个问题

■什么控制信号能够被存入控制存储?

■什么控制信号只能由硬连线逻辑生成?

❑ 什么信号必须在数据通路中处理才能得到?

14

14

关于微序列器的高级问题

- 如果机器出现中断会发生什么?
- 如果指令产生异常会怎么样?
- 如何使用这种控制结构实现一条复杂指令?
  - 考虑 REP MOVS
- 访存对齐
  - 硬件如何保证读写的正确性
- 内存映射I/O
  - 地址控制逻辑决定访存指令的地址是内存还是I/O设备
  - 相应地驱动内存或I/O设备并且设置多路选择器
  - 有些控制信号不能保存在控制存储中

15

15

抽象的力量

- 控制存储的微指令概念使得硬件设计者具有一种新的抽象: 微程序设计
- 设计者可以将任何希望的操作翻译成微指令序列
- 设计者只需要提供
  - 实现目标操作所需的微指令序列
  - 具有正确驱动微指令序列能力的控制逻辑
  - 其它必须附加的数据通路控制信号(如果操作不能翻译成已有的控制信号)

16

16

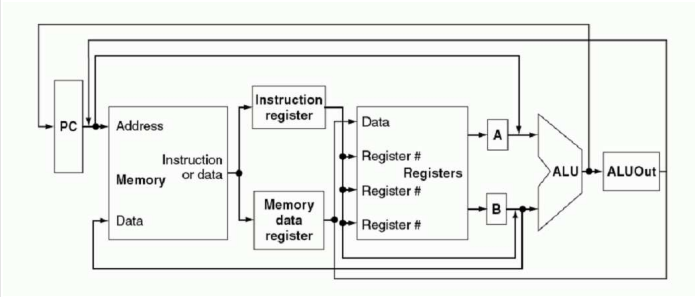
微程序设计的多周期MIPS

- Patterson & Hennessy, 附录 D
- 任何 ISA 都可以这样实现

17

17

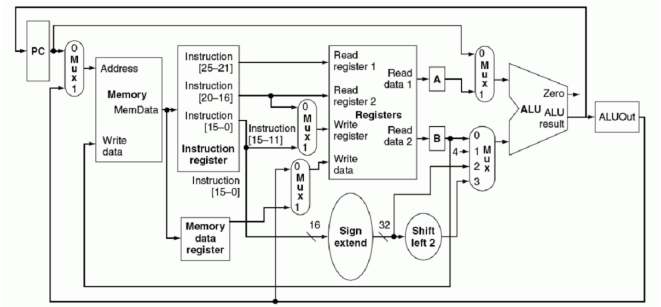
高层抽象的多周期数据通路



18

18

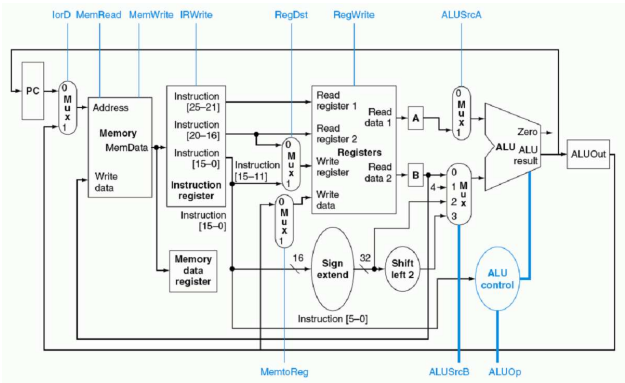
MIPS处理基本指令的多周期数据通路



19

19

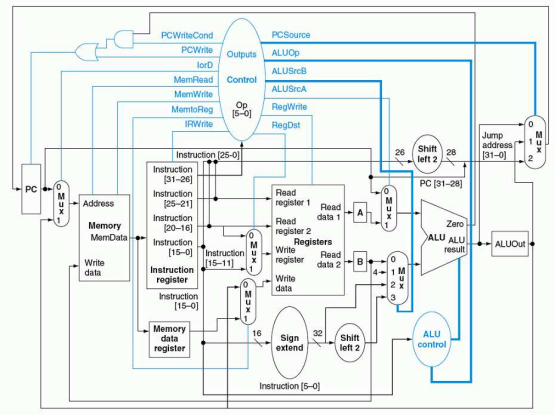
带控制信号的MIPS多周期数据通路



20

20

完整的数据通路（带控制信号）

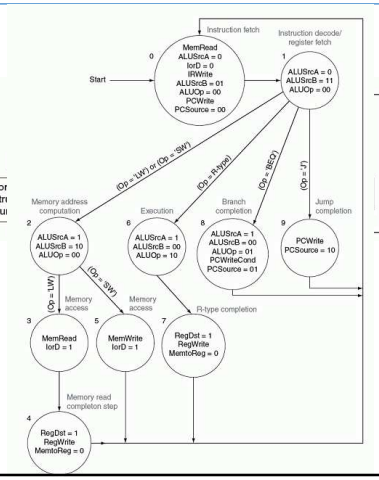


[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

21

21

微程序设计的多周期MIPS-状态机

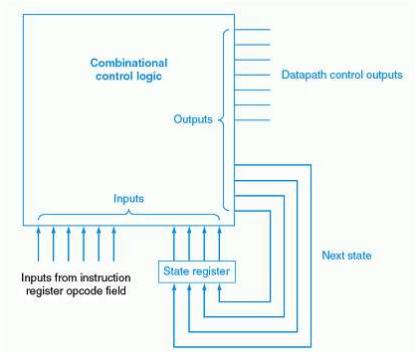


[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

22

22

MIPS FSM的控制逻辑

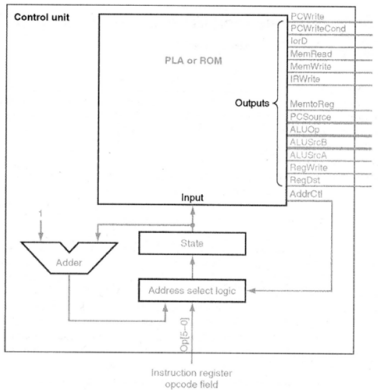


[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

23

23

MIPS FSM 的微程序设计控制

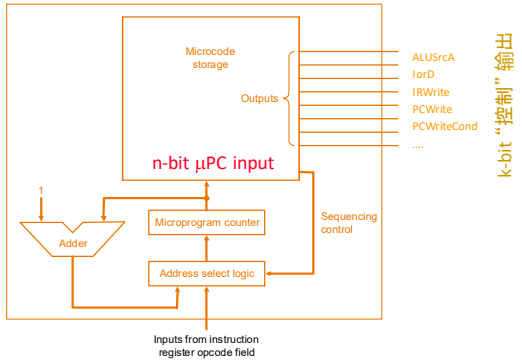


[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

24

24

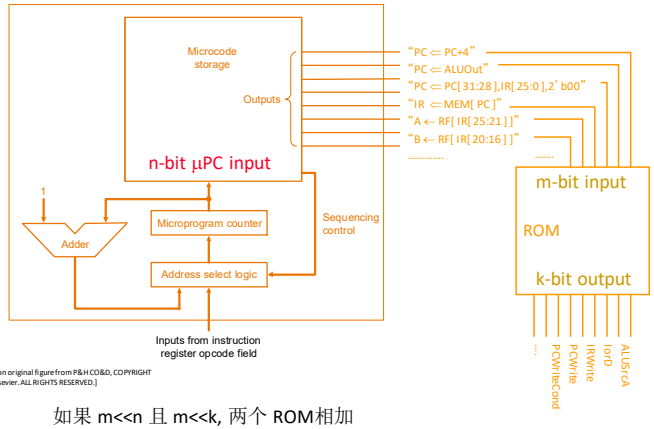
水平微码



控制存储：$2^n \times$ k bit （不包括序列生成逻辑）

25

垂直微码



如果 m<<n 且 m<<k, 两个 ROM相加
($2^n \times m + 2^m \times k$ bit) 应该小于水平微码的ROM ($2^n \times k$ bit)

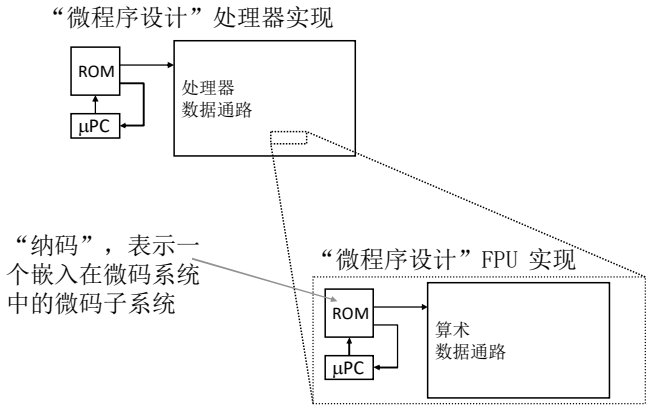
26

"纳码" 和 "毫码"

- *"纳码"*：比 "正牌微码" 低一级
  - 为微控制数据通路中的子系统（例如一个复杂的浮点运算模块）做的微程序设计控制
- *"毫码"*：比 "正牌微码" 高一级
  - 可以被微控制器调用的ISA级别的子程序，用以处理复杂的操作和系统功能
  - 例如，Heller and Farrell, "Millicode in an IBM zSeries processor," IBM JR&D, May/Jul 2004.
- 在这两种情况下，需要避免将主微控制器复杂化
- 可以理解为不同抽象层次下的 "微码"

27

纳码概念图解



28

回顾：单周期微架构的复杂性

- 人为因素
  - 所有指令都和最慢的指令一样慢
- 低效
  - 所有指令都和最慢的指令一样慢
  - 必须为所有指令提供最坏情况下的资源
  - 对于一条指令执行周期中在不同阶段会访问同一个资源的情况，必须为该资源提供“副本”
- 不一定是实现ISA的最简单方法
  - REP MOVS, INDEX, POLY等指令的单周期实现?
- 不容易优化/提升性能
  - 对通常情况(普通指令)做优化不起作用
  - 任何时候都要优化最坏的情况

29

29

回顾：微体系结构设计原则

- 关键路径设计
  - 找到时延最大的组合逻辑，尽可能的减小它的时延
- 基本（典型）设计
  - 在重要的地方花时间和资源
    - 提升机器设计目标要求的应有能力
  - 通常情况 vs. 特殊情况
- 平衡设计
  - 平衡流过硬件部件的指令/数据流
  - 平衡完成工作所需要的硬件
- *单周期微体系结构是如何遵循这些原则的？*

30

30

多周期设计的好处

- 关键路径设计
  - 可以独立地针对每条指令的最糟糕情况来优化关键路径
- 基本（典型）设计
  - 可以通过优化执行“重要”指令（占用大量执行时间）所需的状态数来达到需要的效果
- 平衡设计
  - 不需要提供比实际需求更多的资源或能力
    - 一条指令需要多次使用资源“X”并不意味着需要多个“X”
    - 使硬件更高效：一条指令可以多次***重用***硬件部件

31

31

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

32

32

硬件升级

- 对微码升级/打补丁的能力（处理器发货之后）
  - 不用更换处理器就可以增加新的指令!
  - “修复”硬件实现的缺陷
- 例如
  - IBM 370 Model 145: 微码存储在主存中，可以在重启之后升级
  - IBM System z: 与 370/145类似
    - Heller and Farrell, “Millicode in an IBM zSeries processor,” IBM JR&D, May/Jul 2004.
  - B1700 微码可以在处理器运行时更新
    - 用户可微编程的机器!

33

33

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

34

34

是否可以利用闲置的硬件改善并发?

- 目标: 并发 → 吞吐量（一个周期内完成更多的“工作”）
- 思路: 当一条指令在它的处理阶段使用某些资源的同时，使用该指令不需要的闲置资源处理其它指令
  - 例如，当译码一条指令时，取下一条指令
  - 例如，当执行一条指令时，译码另一条指令
  - 例如，当一条指令访问数据存储器时，执行另一条指令
  - 例如，当一条指令写回结果到寄存器堆的时候，另一条指令访问数据存储器

35

35

流水线：基本思想

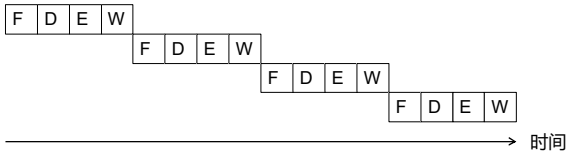
- 系统性更强
  - 多条指令流水线执行
  - 类比: 指令的“装配线处理”
- 思路
  - 指令处理周期切分为不同的处理“阶段”
  - 保证有足够的硬件资源在每个阶段处理指令
  - 每个阶段处理不同的指令
    - 指令在连续的阶段中按照程序序连续地处理
- 好处: 提升了指令处理的吞吐量（1/CPI）
- 坏处？ 请开始思考这个问题……

36

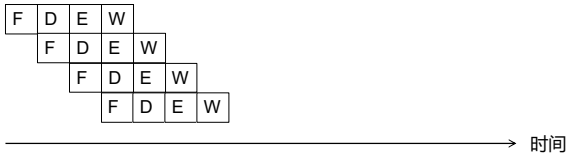
36

例子：执行4条独立的ADD指令

- 多周期：每条指令4个时钟周期



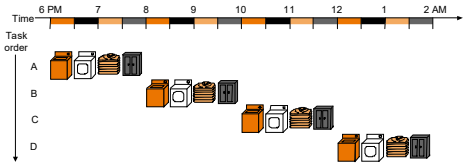
- 流水线：4条指令4个周期（稳定状态）



37

37

洗衣房类比



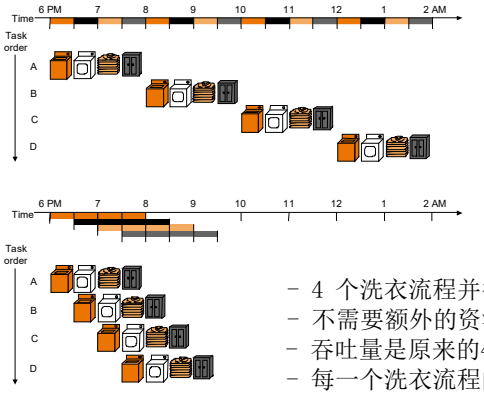
- “脏衣物装入洗衣机”
- “洗衣程序结束，湿衣物装入甩干机”
- “甩干程序结束，取出干衣物熨烫”
- “熨烫结束，将衣物取走”

- 同一个洗衣流程的各个步骤是顺序相关的
- 不同的洗衣流程是互相无关的
- 不同的步骤不共享资源

38

38

多个洗衣流程流水

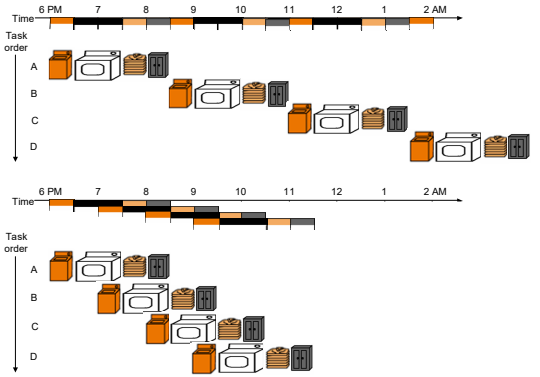


- 4 个洗衣流程并行
- 不需要额外的资源
- 吞吐量是原来的4倍
- 每一个洗衣流程的延迟不变

39

39

多个洗衣流程流水：可能的实际情况

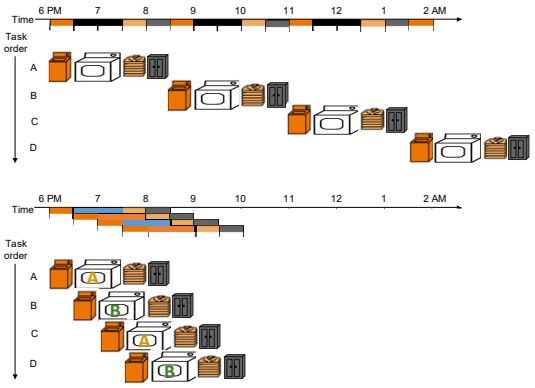


最慢的步骤决定吞吐量

40

40

多个洗衣流程流水：可能的实际情况



使用2个干洗机可获得“理想”的吞吐量

41

41

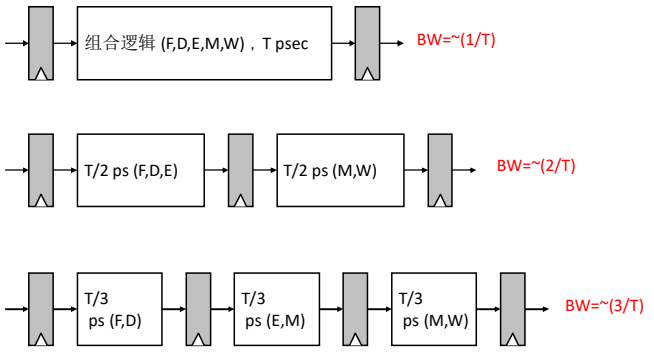
理想的流水线

- 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)
- 类似的例子：汽车装配线，洗衣
  - 指令处理“周期”?

42

42

理想的流水线



43

43

真实一点的流水线：吞吐量

- 延迟为 T 的非流水线

  BW = 1/(T+S) ， S = 锁存延迟

- K阶段流水线

  $BW_{k\text{-}stage}$ = 1 / (T/k +S)

  BW_{max} = 1 / (1个门延迟 + S)

44

44

真实一点的流水线：开销

- 综合开销为G的非流水线

  Cost = G+L ， L = 锁存开销

- K阶段流水线

  $Cost_{k-stage}$ = G + Lk

45

45

流水线指令处理

46

46

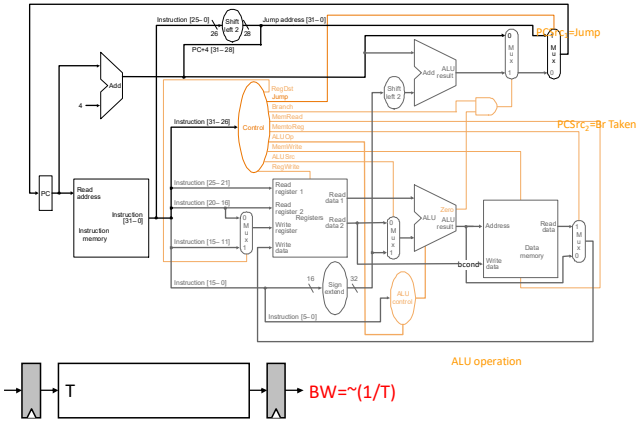
回顾：指令处理周期

1. 取指令（IF）
2. 指令译码和取寄存器操作数（ID/RF）
3. 执行/计算内存地址（EX/AG）
4. 取内存操作数（MEM）
5. 保存/写回结果（WB）

47

47

回顾单周期微体系结构

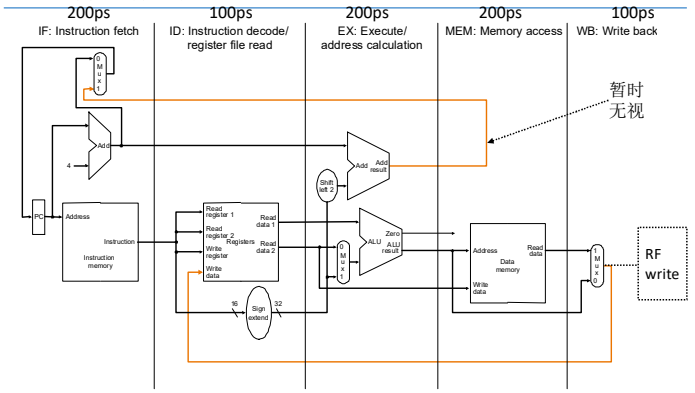


BW=~(1/T)

48

48

划分阶段

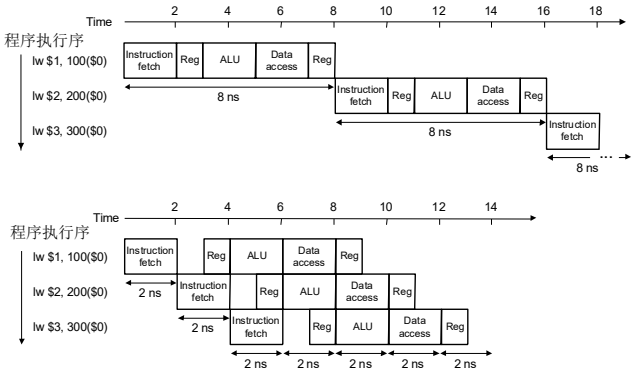


这样划分正确吗?

为什么不是4个或者6个阶段?

49

指令流水线吞吐

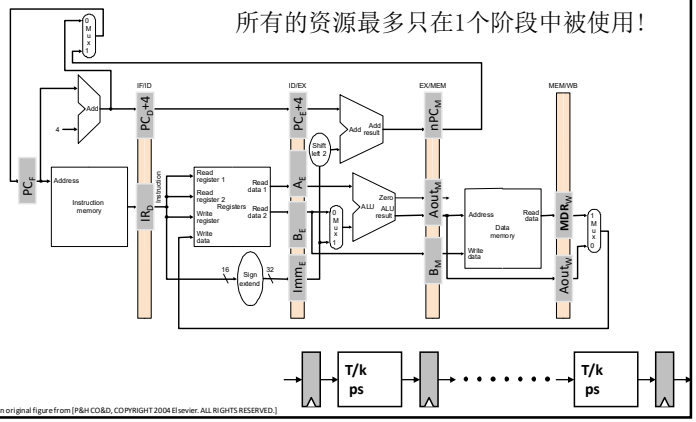


5阶段的加速比不是理想模型中预计的5… 为什么?

50

实现流水线处理：流水线寄存器

所有的资源最多只在1个阶段中被使用!

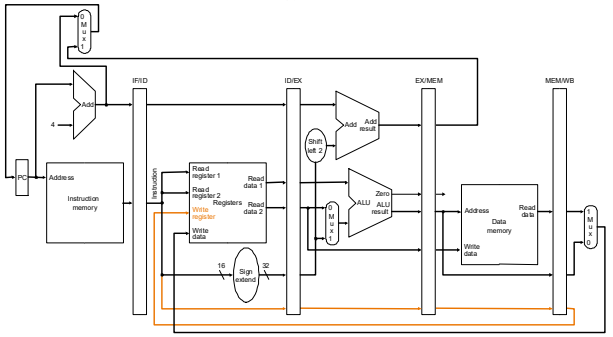


51

流水线操作示例

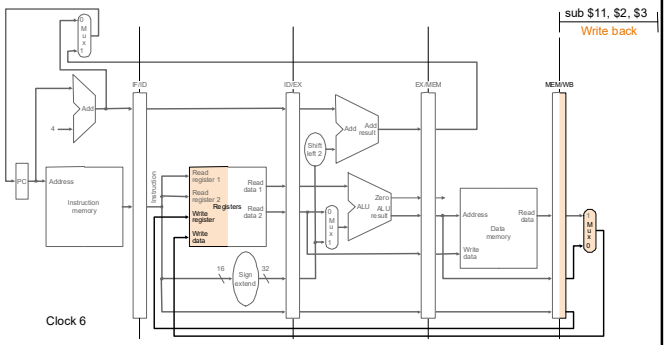
所有指令必须遵循同样的路径和时序流经流水线各流水段

对性能有什么影响?



52

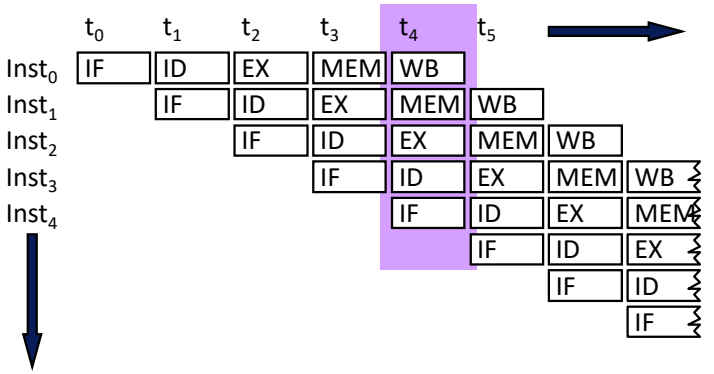
流水线操作示例



53

53

图解流水线操作：操作视图



54

54

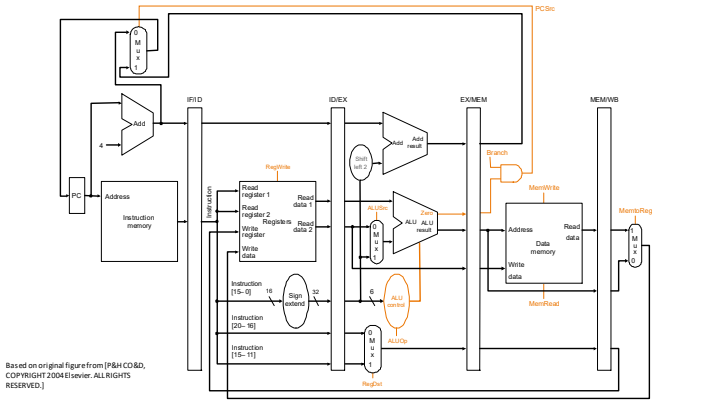
图解流水线操作：资源视图

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
IF	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	I_{10}
ID		I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9
EX			I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8
MEM				I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7
WB					I_0	I_1	I_2	I_3	I_4	I_5	I_6

55

55

流水线中的控制点



和单周期相同的控制点集合!!

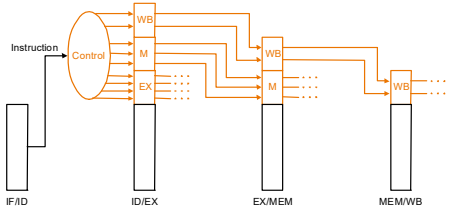
56

56

流水线中的控制信号

- 对于给定的指令
  - 与单周期同样的控制信号，但是
  - 控制信号需要根据阶段的划分在不同周期获得

⇒用与单周期相同的逻辑进行一次译码，然后缓存控制信号直到被使用



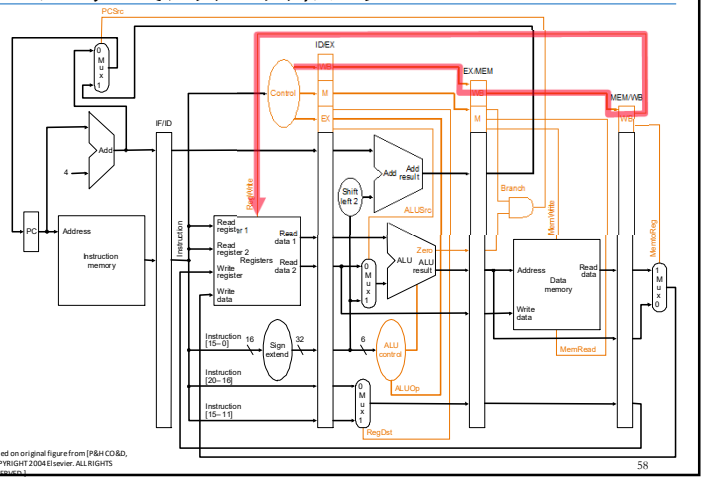
⇒或者携带相关的“指令字/字段”流经流水线，在每个流水段内部译码（仍然使用相同的逻辑）

哪一种更好?

57

57

流水线的控制信号



58

58

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)
- 类似的例子：汽车装配线，洗衣
  - 指令处理“周期”?

59

59

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)

60

60

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

61

61

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

62

62

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

63

63

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?

64

64

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***

65

65

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

66

66

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

67

67

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

68

68

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

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?

相关性的检测方法(II)

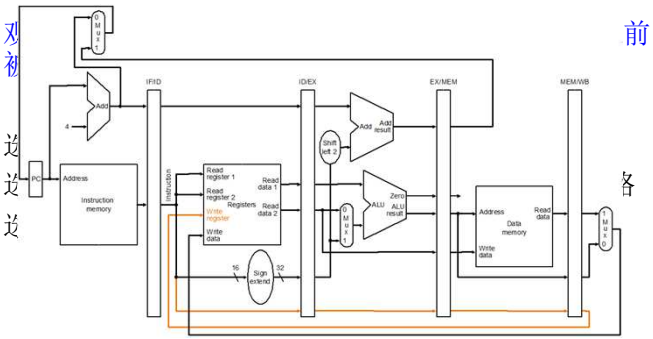
- 相关性检查逻辑（组合逻辑）
  - 用特殊的逻辑检查是否有任何前序指令会写任何当前译码指令的源操作数寄存器
  - 是：该指令/流水线停顿
  - 否：无需停顿… 没有流相关

- 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

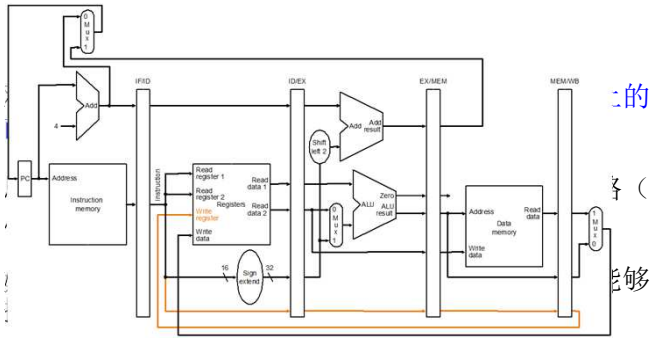
一旦检测出相关性

- 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



73

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

74

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?

75