# 高等计算机体系结构
# 第六讲: 流水线和数据相关

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
2021-04-02

1

## 提醒：作业

- 作业 2
  - 3月26日已发布，4月9日上课前截止
  - 单周期与多周期微体系结构
- 作业 3
  - 4月9日发布
  - 流水线

2

2

## 提醒：实验 1

- 4月16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





# 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





# 单周期和多周期性能分析

数据通路各部分以及各类指令的执行时间

| Instruction class | Instruction memory | Register read | ALU operation | Data memory | Register write | Total |
|---|---|---|---|---|---|---|
| R-type | 200 | 50 | 100 | 0 | 50 | 400 ps |
| Load word | 200 | 50 | 100 | 200 | 50 | 600 ps |
| Store word | 200 | 50 | 100 | 200 | | 550 ps |
| Branch | 200 | 50 | 100 | 0 | | 350 ps |
| Jump | 200 | | | | | 200 ps |

- 采用单周期，即所有指令周期固定为单一时钟周期
  - 指令平均周期 = 600ps
- 采用多周期，各型指令所需的时间和假设出现的频率

  R型指令：800ps， 45%

  lw指令 ：1000ps， 25%

  sw指令 ：800ps， 10%

  beq指令 ：600ps， 15%

  j指令 ：600ps， 5%
- 平均CPI： 5*25%+4*10%+4*45%+3*15%+3*5% = 4.05
- 指令平均执行时间：1000*25%+800*10%+800*45%+600*15%+600*5% = 810ps



# 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

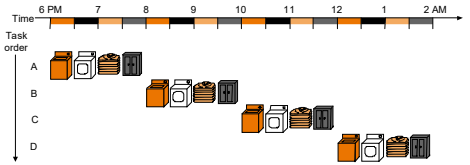
# 回顾：流水线的基本思想

- 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？





# 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”

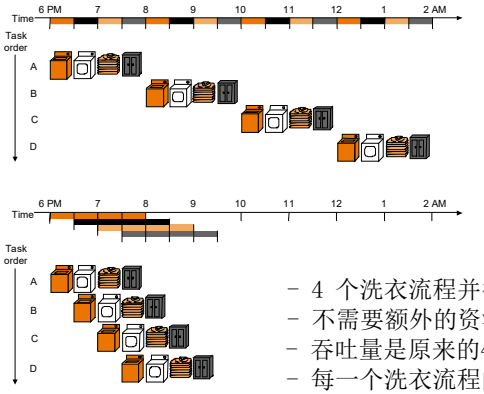
- 同一个洗衣流程的各个步骤是顺序相关的
- 不同的洗衣流程是互相无关的
- 不同的步骤不共享资源







# 多个洗衣流程流水



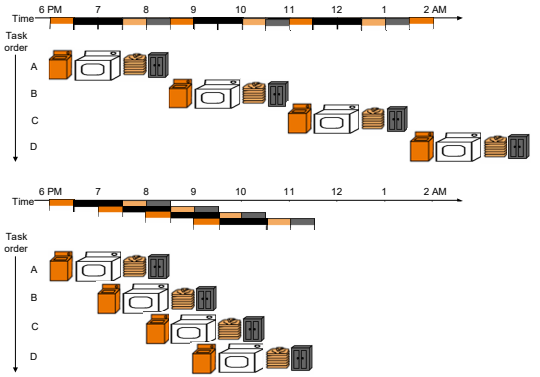

- 4 个洗衣流程并行
- 不需要额外的资源
- 吞吐量是原来的4倍
- 每一个洗衣流程的延迟不变





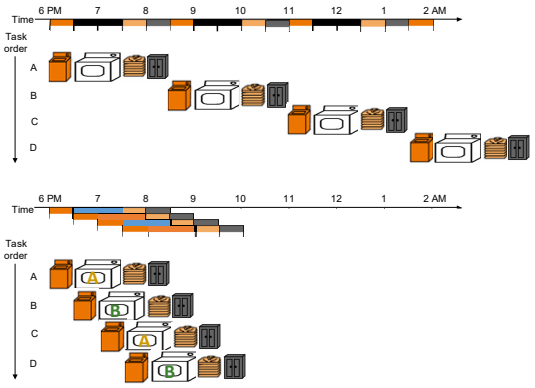


# 多个洗衣流程流水：可能的实际情况




最慢的步骤决定吞吐量

## 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



13

13

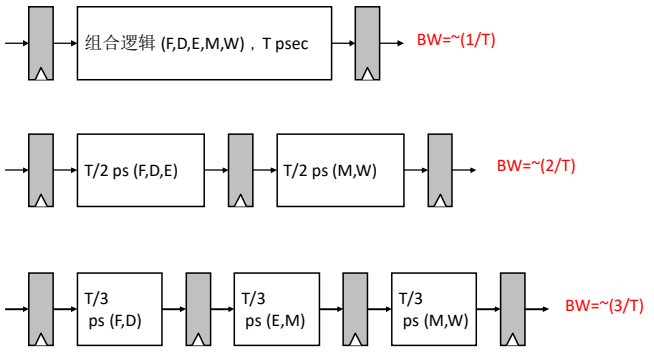
## 理想的流水线

- 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”?

14

14

## 理想的流水线




15

15

## 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 )


16

16

# 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


17

17

# 流水线指令处理

18

18

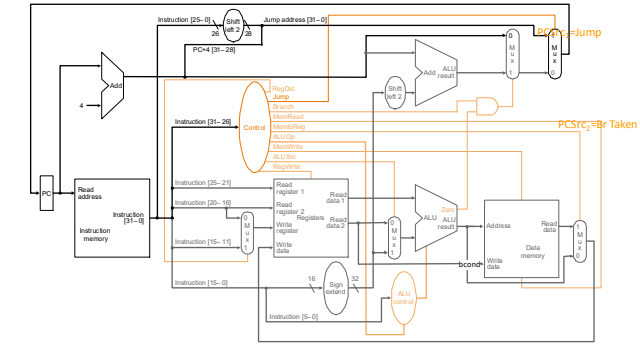
# 回顾：指令处理周期

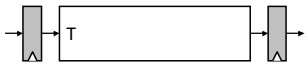
1. 取指令（IF）
2. 指令译码和取寄存器操作数（ID/RF）
3. 执行/计算内存地址（EX/AG）
4. 取内存操作数（MEM）
5. 保存/写回结果（WB）

19

19

# 回顾单周期微体系结构



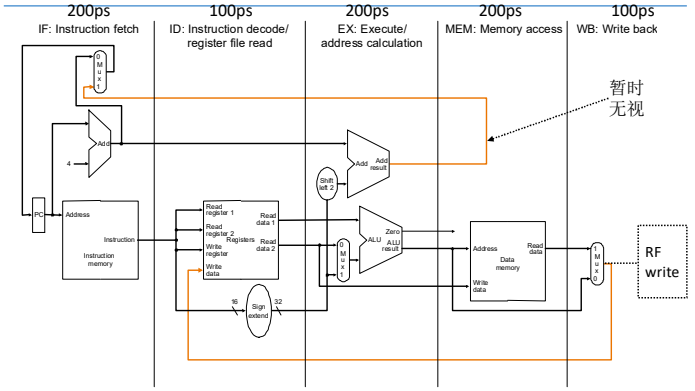

BW=~(1/T)



20

20

# 划分阶段




这样划分正确吗?

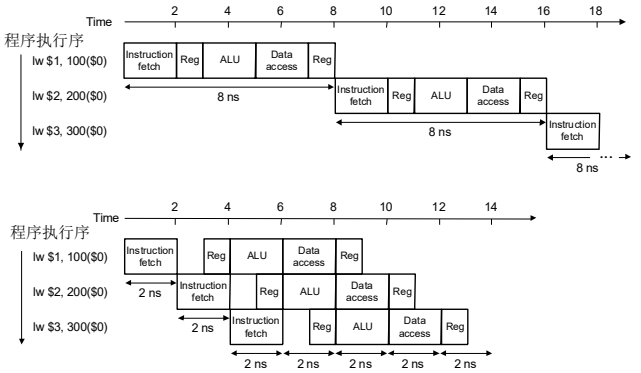
为什么不是4个或者6个阶段?



21

21

# 指令流水线吞吐



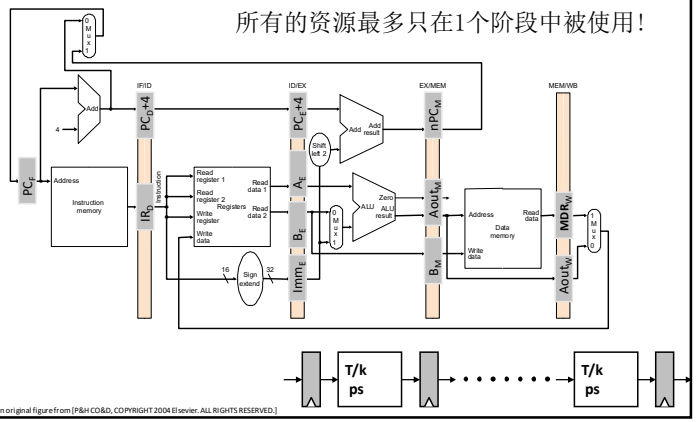

5阶段的加速比不是理想模型中预计的5… 为什么?

22

22

# 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!

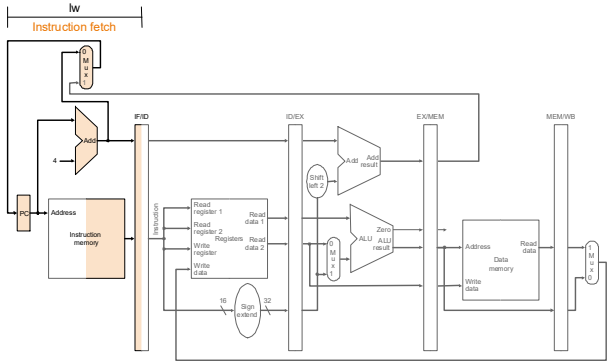





23

23

# 流水线操作示例

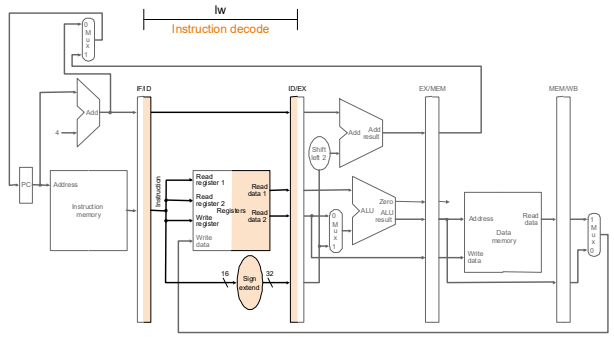





24

24

# 流水线操作示例

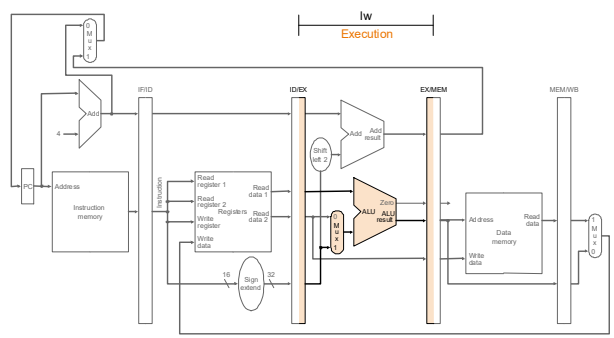





25

25

# 流水线操作示例

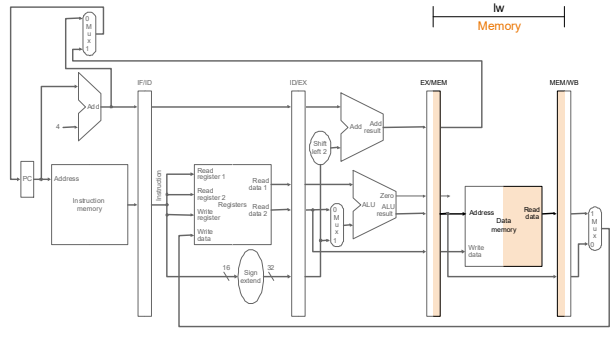





26

26

# 流水线操作示例

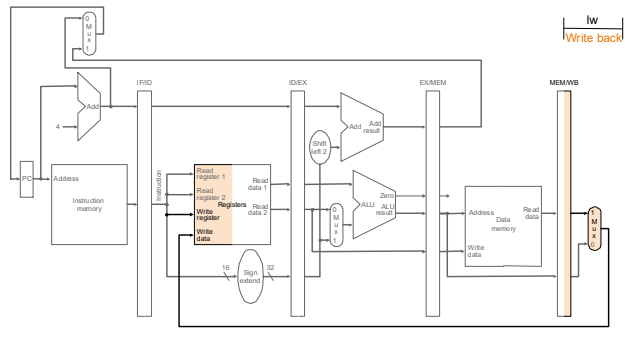





27

27

# 流水线操作示例

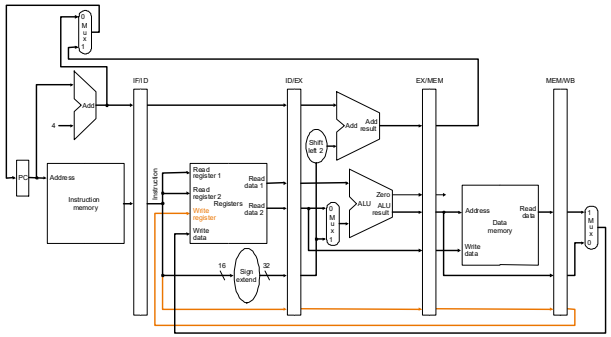





28

28

# 流水线操作示例



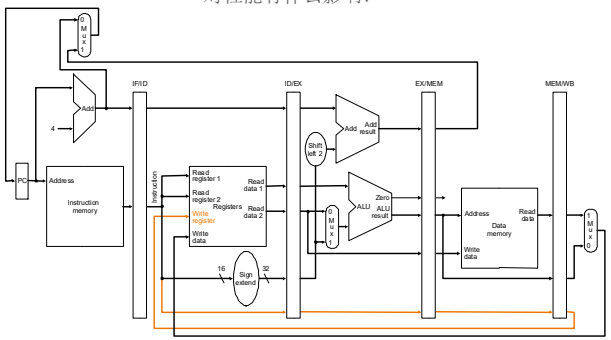



29

29

# 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?

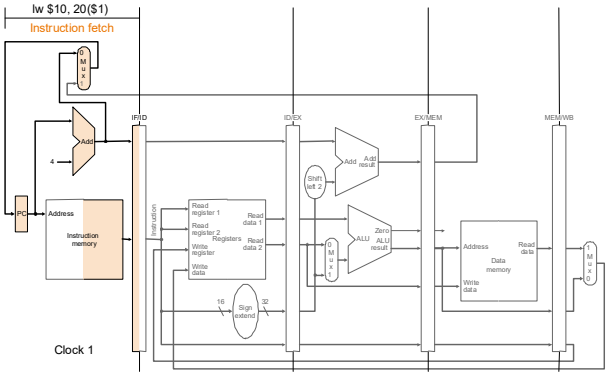





30

30

# 流水线操作示例

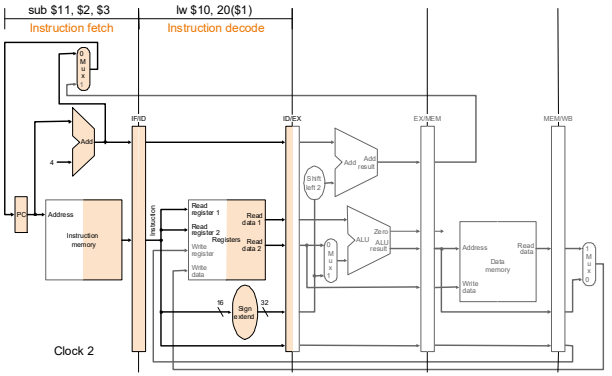





31

31

# 流水线操作示例

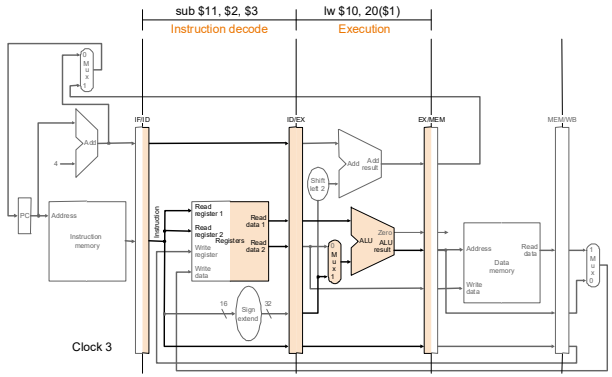





32

32

# 流水线操作示例

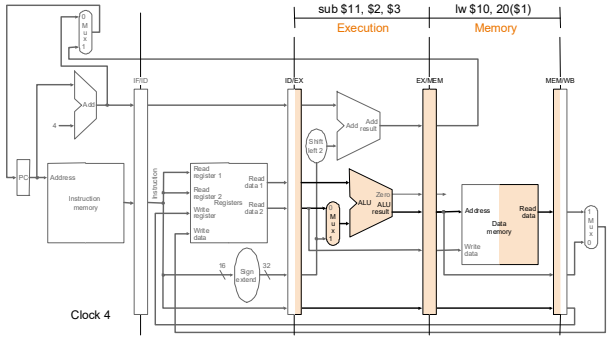





33

33

# 流水线操作示例

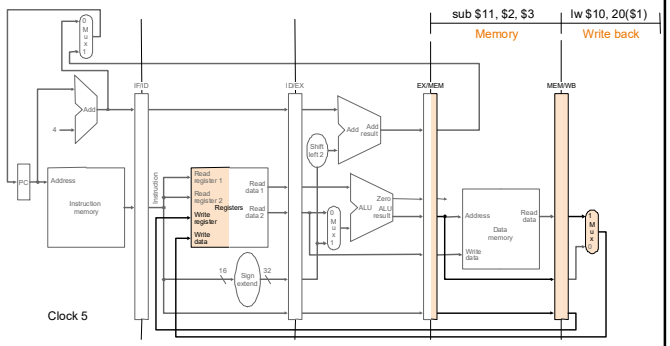





34

34

# 流水线操作示例

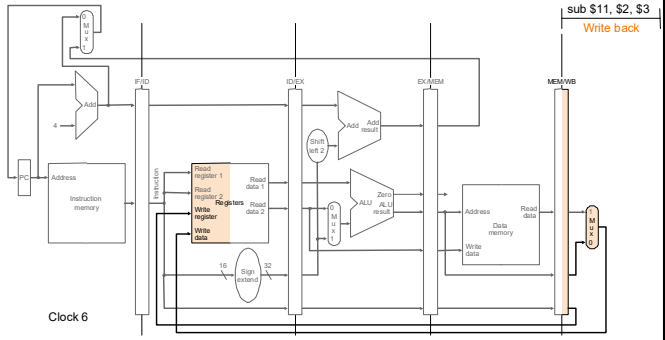





35

35

# 流水线操作示例

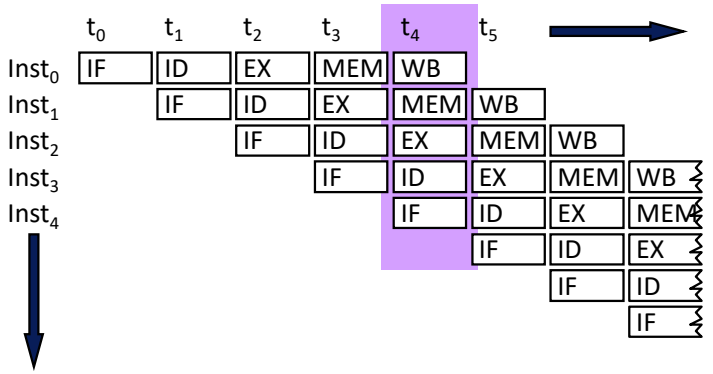





36

36

## 图解流水线操作：操作视图




37

37

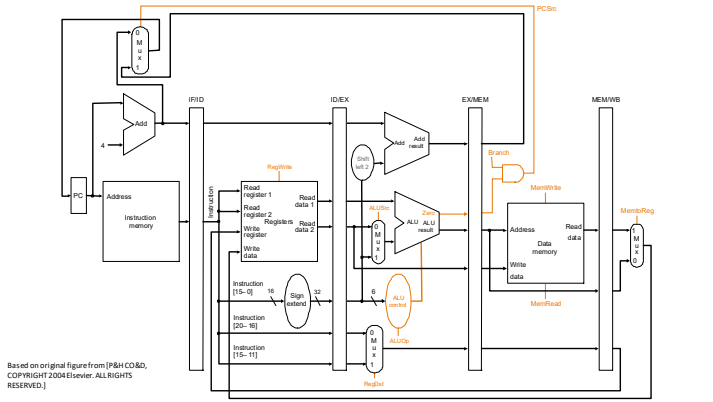
## 图解流水线操作：资源视图

| | $t_0$ | $t_1$ | $t_2$ | $t_3$ | $t_4$ | $t_5$ | $t_6$ | $t_7$ | $t_8$ | $t_9$ | $t_{10}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 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 | | | | | $I_0$ | $I_1$ | $I_2$ | $I_3$ | $I_4$ | $I_5$ | $I_6$ |

38

38

## 流水线中的控制点




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

和单周期相同的控制点集合!!

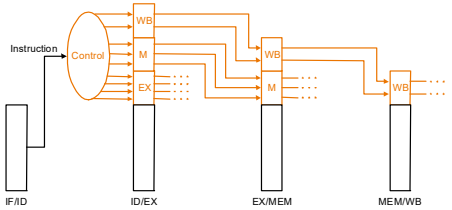
39

39

## 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



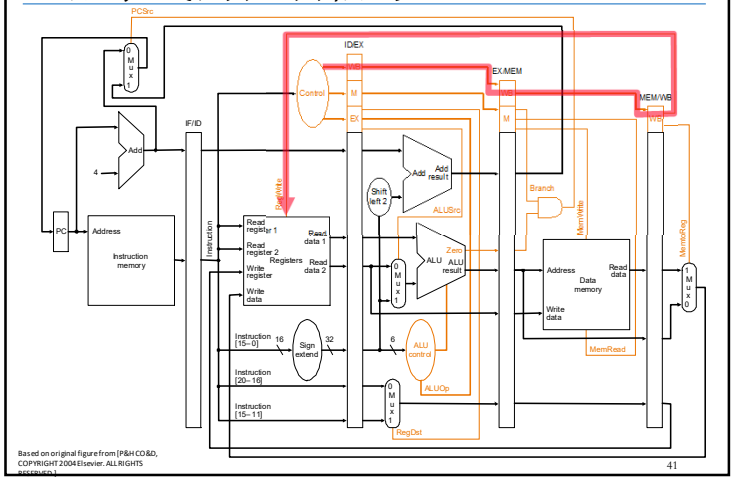

⇒或者携带相关的“指令字/字段”流经流水线，在每个流水段内部译码（仍然使用相同的逻辑）

哪一种更好?

40

40

# 流水线的控制信号




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)

42

42

# 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)

43

43

# 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

44

44

## 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

45

45

## 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

46

46

## 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?

47

47

## 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***

48

48

## 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 $r_7$

49

49

## 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

50

50

## 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

51

51

## 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

52

52

## 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?

53

53

## 相关性的检测方法(II)

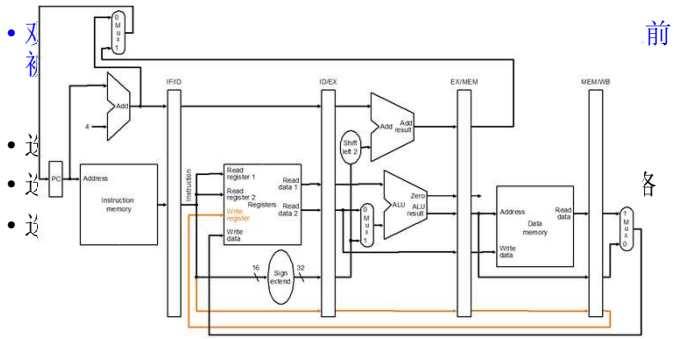
- 相关性检查逻辑（组合逻辑）
  - 用特殊的逻辑检查是否有任何前序指令会写任何当前译码指令的源操作数寄存器
  - 是：该指令/流水线停顿
  - 否：无需停顿… 没有流相关
- 优点：
  - 反相关和输出相关不会导致停顿
- 缺点：
  - 逻辑比计分板更复杂
  - 当我们设计的流水线结构更深更宽时逻辑会变得越发复杂

54

54

## 一旦检测出相关性

- 接下来该怎么做?

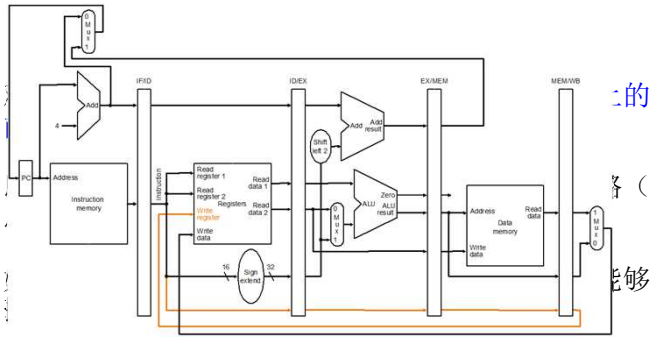



55

55

## 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




56

56

## 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

57

57

## 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?

58

58

## 处理数据相关：
## *深入探索和实现方法*

59

59

## 回顾：如何处理数据相关

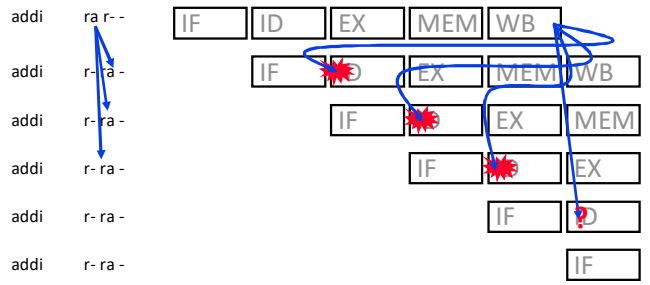
- 五种处理流相关的基本方法
  - 检测并等待直到值在寄存器堆中可以访问
  - 检测并转发/旁路数据给相关的指令
  - 检测并消除相关性（在软件层面）
    - 不需要硬件检测相关性
  - 预测 需要的值， “投机”执行，并且验证
  - 其它(细粒度多线程)
    - 不需要检测

60

60

# 处理写后读相关

- 下面的流相关导致5阶段流水线的冲突




61

61

# 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 | | | | |

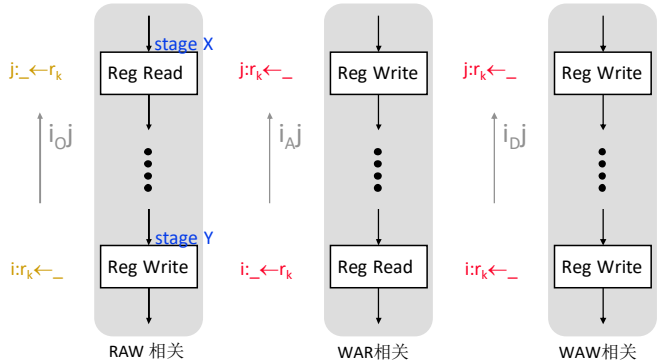
■对给定的流水线，什么情况下2条数据相关指令有潜在的冲突可能?

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

62

62

# 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

63

63

# 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

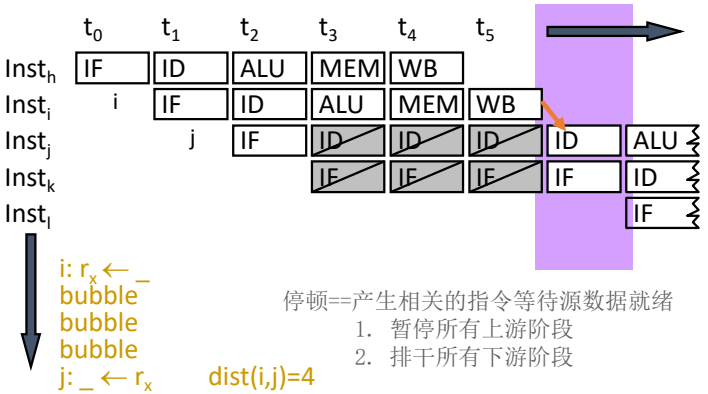
WAW 和 WAR 相关?

内存数据相关?

64

64

# 流水线停顿：解决数据相关

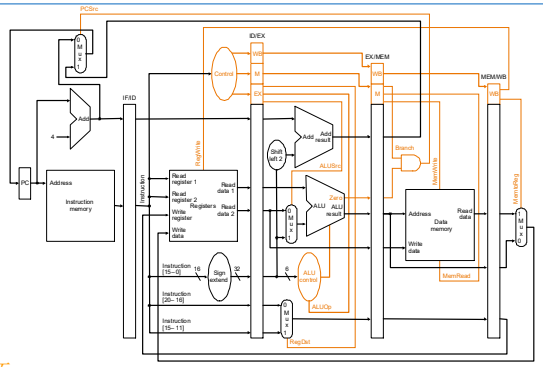



停顿==产生相关的指令等待源数据就绪

1. 暂停所有上游阶段
2. 排干所有下游阶段

65

# 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

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

66

66

# 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
  - $dist(I_A, I_B) \le dist(ID, WB) = 3$
- 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

67

67

# 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

68

68

## 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

69

69

## 示例

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

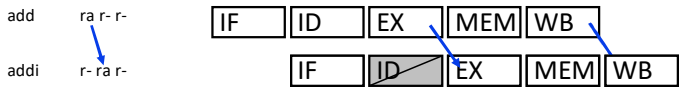
```
for2tst:  addi  $s1, $s0, -1      3 stalls
          slti  $t0, $s1, 0       3 stalls
          bne   $t0, $zero, exit2
          sll   $t1, $s1, 2       3 stalls
          add   $t2, $a0, $t1     3 stalls
          lw    $t3, 0($t2)
          lw    $t4, 4($t2)       3 stalls
          slt   $t0, $t4, $t3     3 stalls
          beq   $t0, $zero, exit2
          .........
          addi  $s1, $s1, -1
          j     for2tst
exit2:
```

70

70

## 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




71

71

## 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

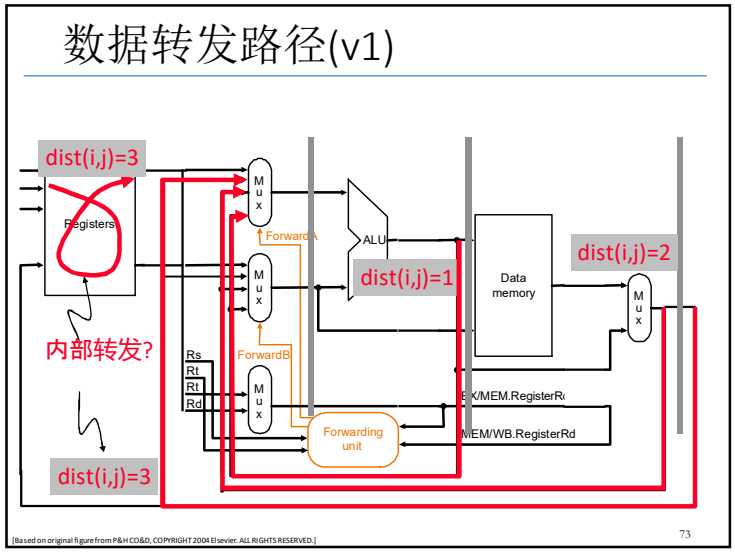
  ⇒ 从数据通路上而不是从RF中取回操作数

  ⇒ 当有多个未完成的操作数定义时取回最新产生的那一个

72

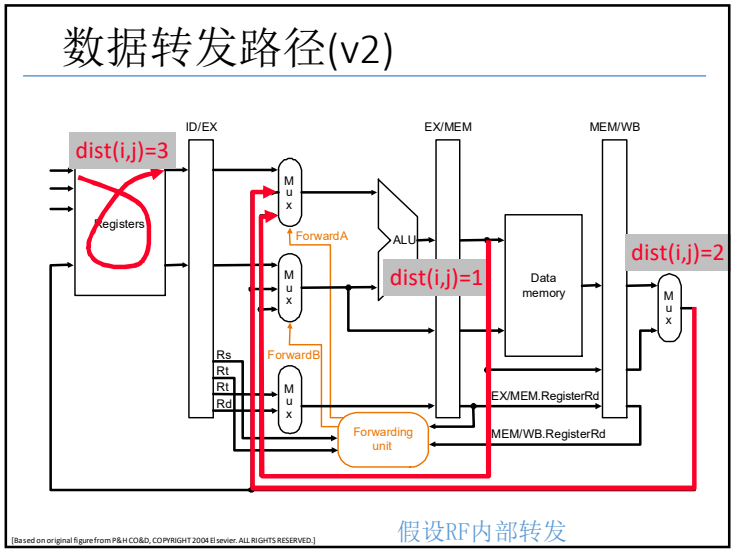
72

# 数据转发路径(v1)




73

# 数据转发路径(v2)




74