计算机体系结构

周学海
xhzhou@ustc.edu.cn
0551-63492149
中国科学技术大学

review

- **ISA的功能设计：任务为确定硬件支持哪些操作。方法是统计的方法。存在CISC和RISC两种设计理念**
  - CISC（Complex Instruction Set Computer）
    - 目标：强化指令功能，减少指令的指令条数，以提高系统性能
    - 基本方法：面向目标程序的优化，面向高级语言和编译器的优化
  - RISC（Reduced Instruction Set Computer）
    - 目标：通过简化指令系统，用最高效的方法实现最常用的指令
    - 主要手段：充分发挥流水线的效率，降低（优化）CPI
- **典型ISA的特色**
  - 在当时比较有特色的设计
    - 如MIPS的不对齐的存储器访问，分支指令后的延迟槽。
    - SPARC的寄存器窗口重叠技术；
    - PA-RISC 的Nullification指令；
    - Alpha仅允许按字访问存储器等
  - 基于目前硬件技术现状，存在一些已经过时的决定

Review

- **RISC-V: 面向“全场景”的RISC-Style ISA**
  - 模块化的指令集、可定制化的扩展，优雅的压缩指令子集（**通用 vs. 专用**）
  - 规整的指令格式、简洁的寻址方式、……(**简单**)
- **微程序控制器： 控制信号基于控制存储器中存储的信息**
  - 每条“宏指令”是通过执行一段由若干条“微指令”构成的微程序来完成
  - 微程序技术对计算机技术与产业发展起到了巨大的推动作用
- **可以采用简单的数据通路+微程序控制器 实现 复杂的指令（指令集）**

Chapter3 基本流水线

- **3.1 基本流水线**
  - 流水线的基本概念
  - 流水线中的相关
  - 流水线的性能分析
- **3.2 基本流水线的扩展**
  - 异常处理
  - 多周期操作处理

3.1 基本流水线

基本概念 → 核心问题 Hazards → 性能分析

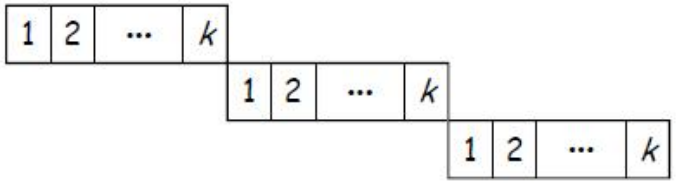
指令流水线技术起源

- **基本思想源于：**
  - 工业上的一种生产方式：流水线又称装配线。指每一个生产单位只专注处理某一个片段的工作，以提高工作效率及产量
  - 1769年，英国人乔赛亚·韦奇伍德开办埃特鲁利亚陶瓷工厂，在场内实行精细的劳动分工，把原来由一个人从头到尾完成的制陶流程分成几十道专门工序，分别由专人完成。
  - 另一说法：20世纪初亨利·福特发明了流水线装配工艺
- **指令流水线起源于IBM7030（Stretch），1961年发布（1956-61）**
  - Stretch是IBM7030的别名，希望在当时的计算机技术上有大踏步的创新（性能是IBM704的100倍）
  - 四级流水
- **20世纪60年代早期开发的CDC6600，还引入了一些流水线方面增强技术**
- **70年代末和80年代初的有关流水线的术语和描述，涵盖了简单流水线中使用的基本技术。**
- **RISC机器设计最初考虑：易于流水线的实现**
  - 第一个RISC项目是在70年代末和80年代初来自IBM、斯坦福和加州大学伯克利分校。IBM 801，Stanford MIPS和Berkeley RISC 1和RISC 2
- **Intel CPU i486 （1989）引入5级流水线**

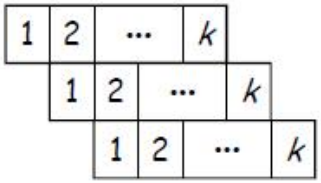
采用流水线技术实现ISA的思想，早于RISC

流水线的基本概念

- **一个任务可以分解为 k 个子任务**
  - K个子任务在 K 个不同阶段（**使用不同的资源**）运行
  - 每个子任务执行需要1个单位时间
  - 整个任务的执行时间为 K倍单位时间
- **流水线执行模式是重叠执行模式**
  - K个流水段并行执行K个不同任务
  - 每个单位时间进入/离开流水线一个任务



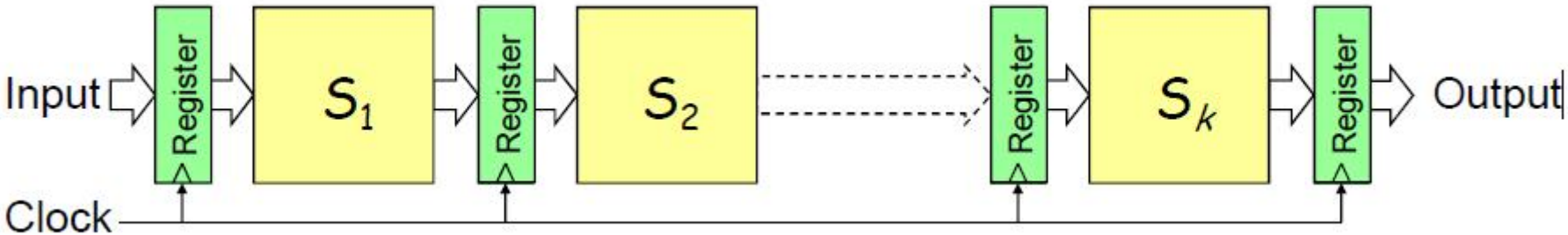
Serial Execution
One completion every k time units



Pipelined Execution
One completion every 1 time unit

同步流水线

- **流水段之间采用时钟控制的寄存器文件（clocked registers）**
- **时钟上升沿到达时…**
  - 所有寄存器同时保存前一流水段的结果
- **流水段是组合逻辑电路**
- **流水线设计中希望各段相对平衡**
  - 即所有段的延迟时间大致相等
- **时钟周期取决于延迟最长的流水段**



流水线的性能

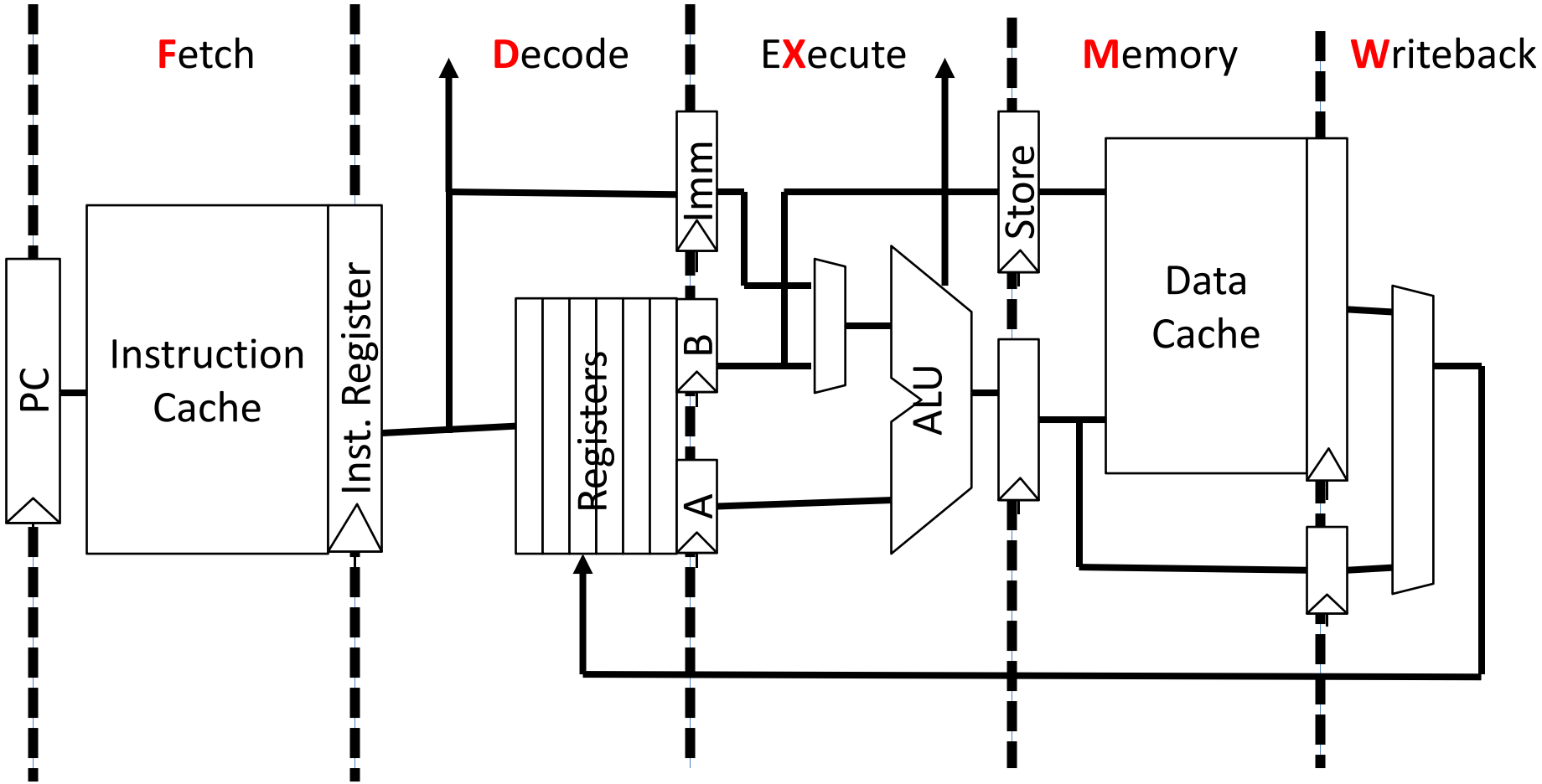
- 设 τ_i = time delay in stage S_i, $i=1..k$
- 时钟周期 τ = max(τ_i) 为最长的流水段延迟
- 时钟频率 f = 1/τ = 1/max(τ_i)
- 流水线可以在k+n-1个时钟周期内完成n个任务
  - 完成第一个任务需要 k个时钟周期
  - 其他n-1个任务需要n-1个时钟周期完成
- K-段流水线的理想加速比（相对于串行执行）

$$S_k = \frac{\text{Serial execution in cycles}}{\text{Pipelined execution in cycles}} = \frac{nk}{k+n-1} \qquad S_k \to k \text{ for large } n$$

典型的RISC　5段指令流水线

- **5个流水段，每段的延迟为1个cycle**
- **IF: 取值阶段**
  - 选择地址：下一条指令地址、转移地址
- **ID: 译码阶段**
  - 确定控制信号 并从寄存器文件中读取寄存器值
- **EX: 执行**
  - Load 、Store：计算有效地址
  - Branch：计算转移地址并确定转移方向
- **MEM: 存储器访问（仅Load和Store)**
- **WB: 结果写回**

典型的 RISC 5段流水线

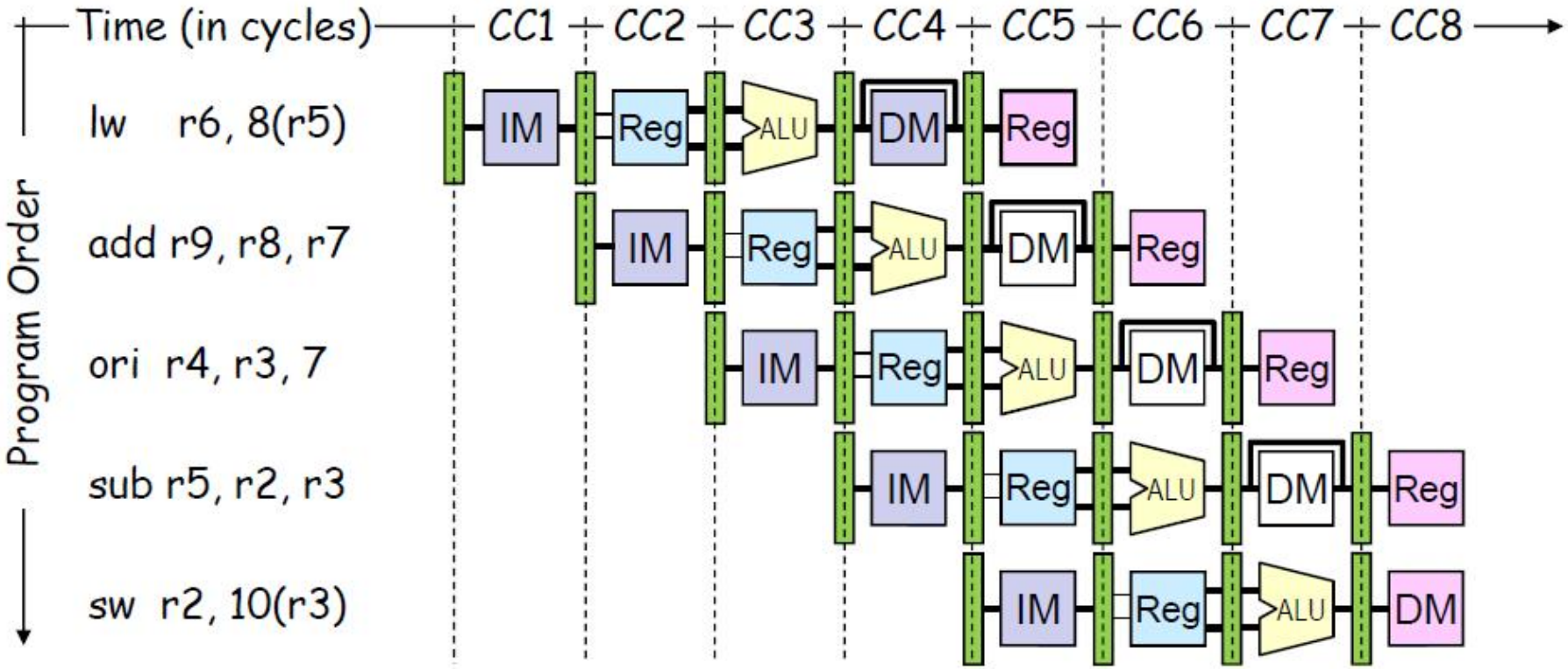


This version designed for regfiles/memories with synchronous reads and writes.

流水线的可视化表示

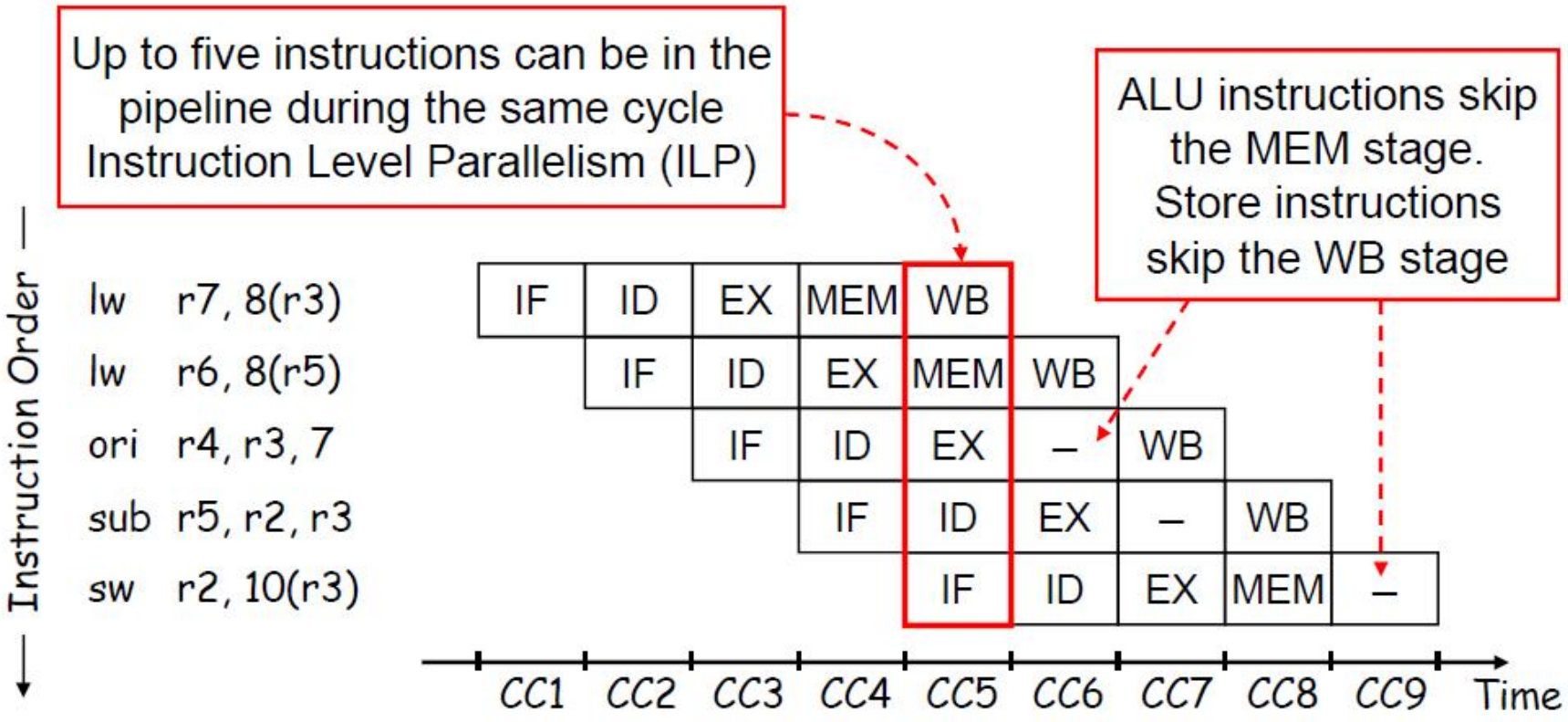
- **多条指令执行多个时钟周期**
  - 指令按程序序从上到下排列
  - 图中展示了每一时钟周期资源的使用情况
  - 不同指令相邻阶段之间没有干扰

基本假设：
1、存储器分为IM和DM
2、寄存器文件的写-读同一寄存器时，将要写的值直接传递
Why?



指令流时序

- **时序图展示：**
  - 每个时钟周期指令所使用的流水段情况
- **指令流在采用5段流水线执行模式的执行情况**



单周期、多周期、流水线控制性能比较

- **假设5段指令执行流水线**

Instruction	Fetch	Reg Read	ALU	Memory	Reg Wr	Time
Load	350 ps	250 ps	350 ps	350 ps	250 ps	1550 ps
Store	350 ps	250 ps	350 ps	350 ps		1300 ps
ALU	350 ps	250 ps	350 ps		250 ps	1200 ps
Branch	350 ps	250 ps	350 ps			950 ps

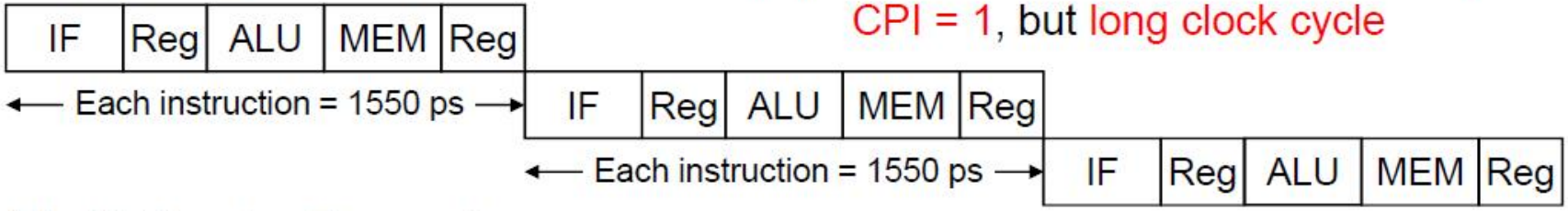
- **某一程序段假设：**
  - 20% load, 10% store, 40% ALU, and 30% branch
- **比较三种执行模式的性能**

单周期、多周期、流水线控制性能比较

Single-Cycle Execution:

T_{clock} = 350+250+350+350+250 = 1550 ps

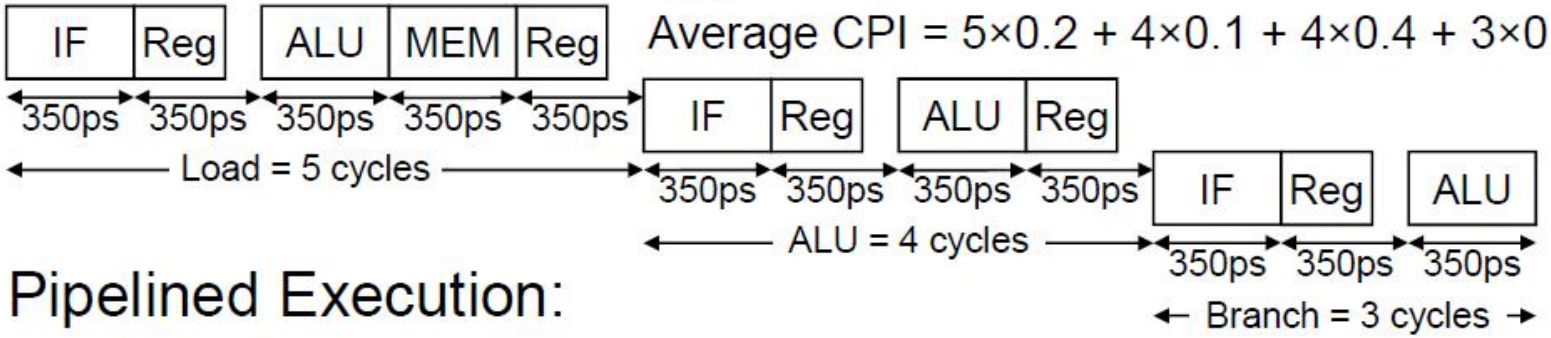
CPI = 1, but long clock cycle



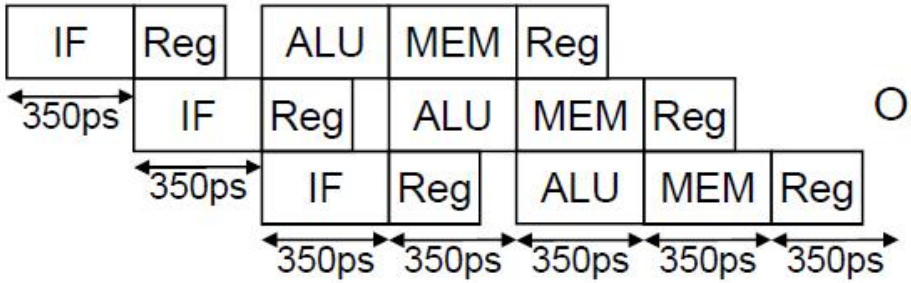
Multi-Cycle Execution:

T_{clock} = 350 ps

Average CPI = 5×0.2 + 4×0.1 + 4×0.4 + 3×0.3 = 3.9



Pipelined Execution:



T_{clock} = 350 ps = max(350, 250)

One instruction completes each cycle

Average CPI = 1

Ignore time to fill pipeline

Recap：流水线技术要点

- **流水线的基本概念**
  - 多个任务重叠（并发/并行）执行，但使用不同的资源
  - 流水线技术提高整个系统的吞吐率，不能缩短单个任务的执行时间
  - 其潜在的加速比 = 流水线的级数
- **流水线正常工作的基本条件**
  - 增加寄存器文件保存当前段传送到下一段的数据和控制信息
  - 需要更高的存储器带宽
- **流水线的相关（hazards)问题**
  - 由于存在相关(hazards)问题，会导致流水线停顿
  - Hazards 问题：流水线的执行可能会导致对资源的访问冲突，或破坏对资源的访问顺序

3.1 基本流水线

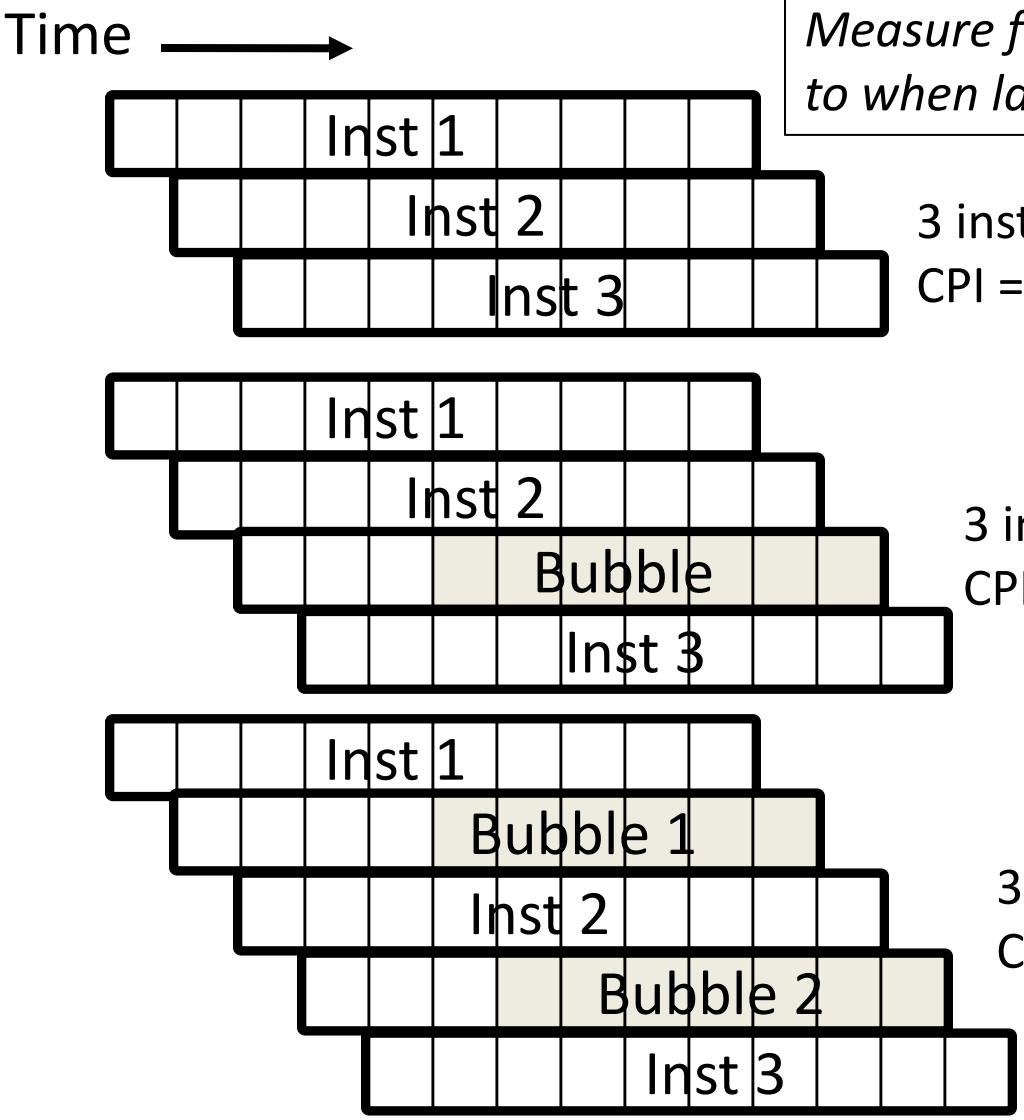
基本概念 → 核心问题 Hazards → 性能分析

流水线的相关（Hazards)

- **结构相关：流水线中一条指令可能需要另一条指令使用的资源**
- **数据和控制相关：一条指令可能依赖于先前的指令生成的内容**
  - 数据相关：依赖先前指令产生的结果（数据）值
  - 控制相关：依赖关系是如何确定下一条指令地址 (branches, exceptions)
- **处理相关的一般方法是插入bubble，导致CPI>1 (单发射理想CPI)**

Pipeline CPI Examples

Measure from when first instruction finishes to when last instruction in sequence finishes.



消减结构相关

- **当两条指令同时需要相同的硬件资源时，就会发生结构相关（冲突）**
  - 方法1：通过将新指令延迟到前一条指令执行完（释放资源后）执行
  - 方法2：增加新的资源
    - E.g., 如果两条指令同时需要操作存储器，可以通过增加到两个存储器操作端口来避免结构冲突
- **经典的 RISC 5-段整型数流水线通过设计可以没有结构相关**
  - 很多RISC实现在**多周期操作**时存在结构相关
    - 例如多周期操作的multipliers, dividers, floating-point units等，由于没有多个寄存器文件写端口 导致 结构冲突

三种基本的数据相关

- **写后读相关(Read After Write (RAW))**

```
I: add x1,x2,x3
J: sub x4,x1,x3
```

  - 由于实际的数据交换需求而引起的
- **读后写相关（Write After Read (WAR)**

```
I: sub x4,x1,x3
J: add x1,x2,x3
```

  - 编译器编写者称之为“anti-dependence”（反相关），是由于重复使用寄存器名“x1”引起的.
- **写后写相关（Write After Write (WAW)）**

```
I: sub x1,x4,x3
J: add x1,x2,x3
```

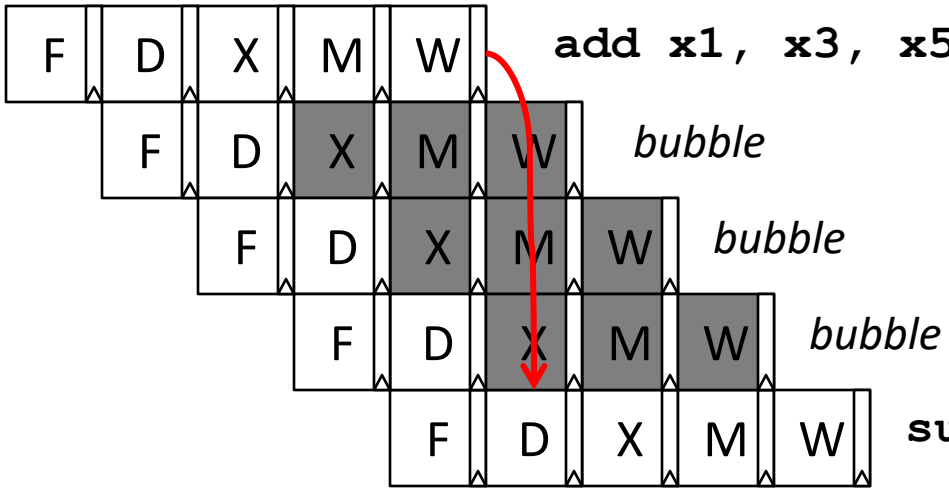
  - 编译器编写者称之为“output dependence”，也是由于重复使用寄存器名“x1”引起的.
  - 在后面的复杂的流水线中我们将会看到 WAR 和WAW 相关

消减数据相关的三种策略

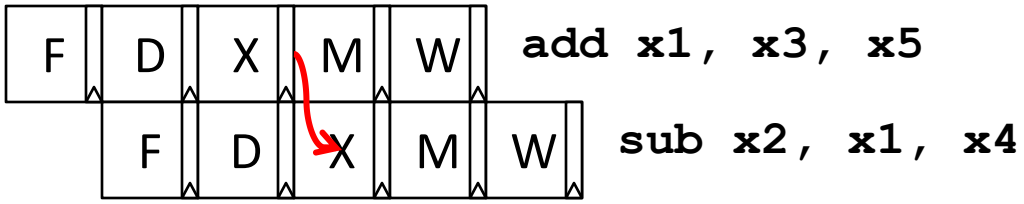
- **联锁机制（Interlock)**
  - 在issue阶段保持当前相关指令，等待相关解除
- **设置旁路定向路径（Bypass or Forwarding)**
  - 只要结果可用，通过旁路尽快传递数据
- **投机（Speculate)**
  - 猜测一个值继续，如果猜测错了再更正

Interlocking Versus Bypassing

```
add x1, x3, x5
sub x2, x1, x4
```

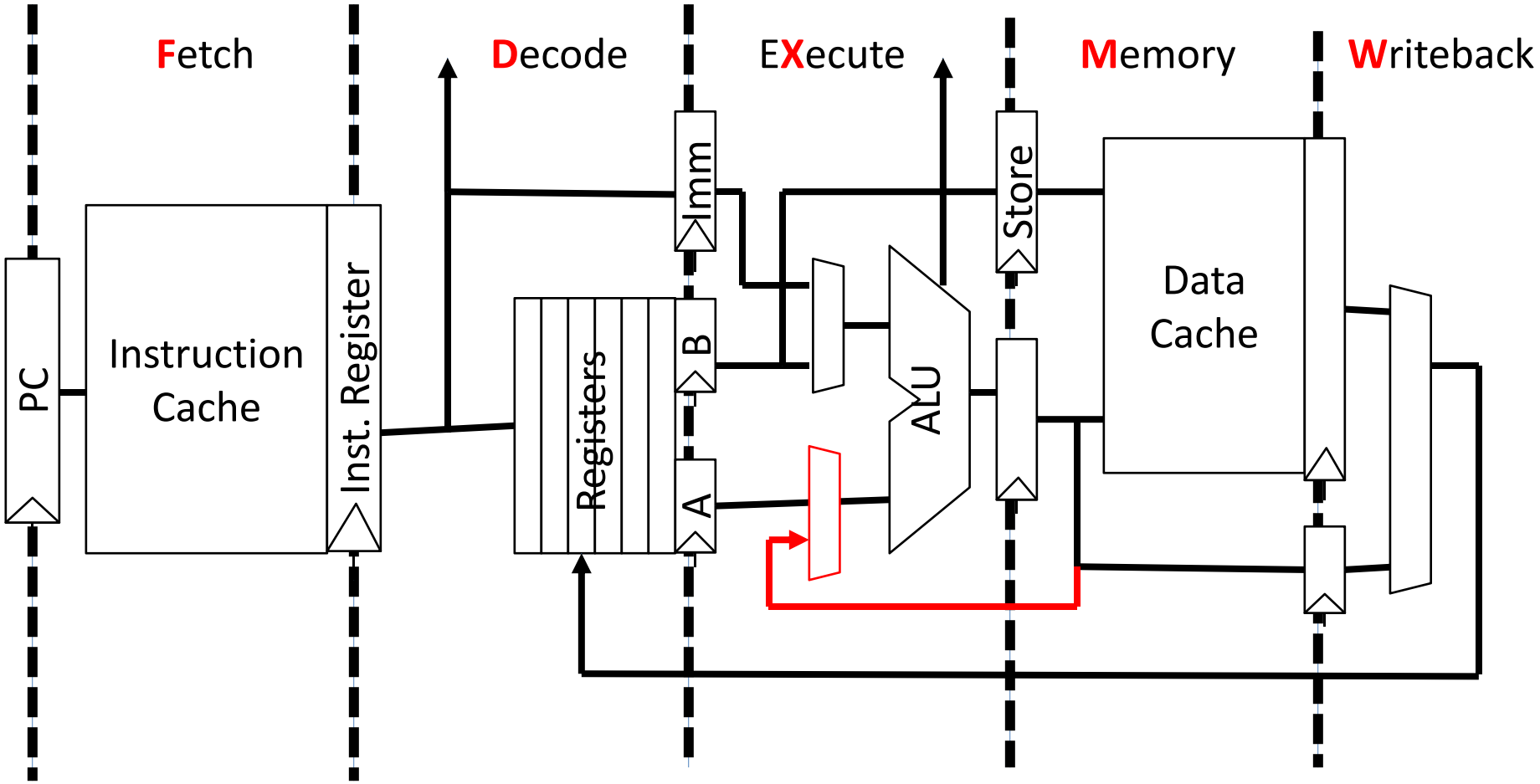


Instruction interlocked in decode stage

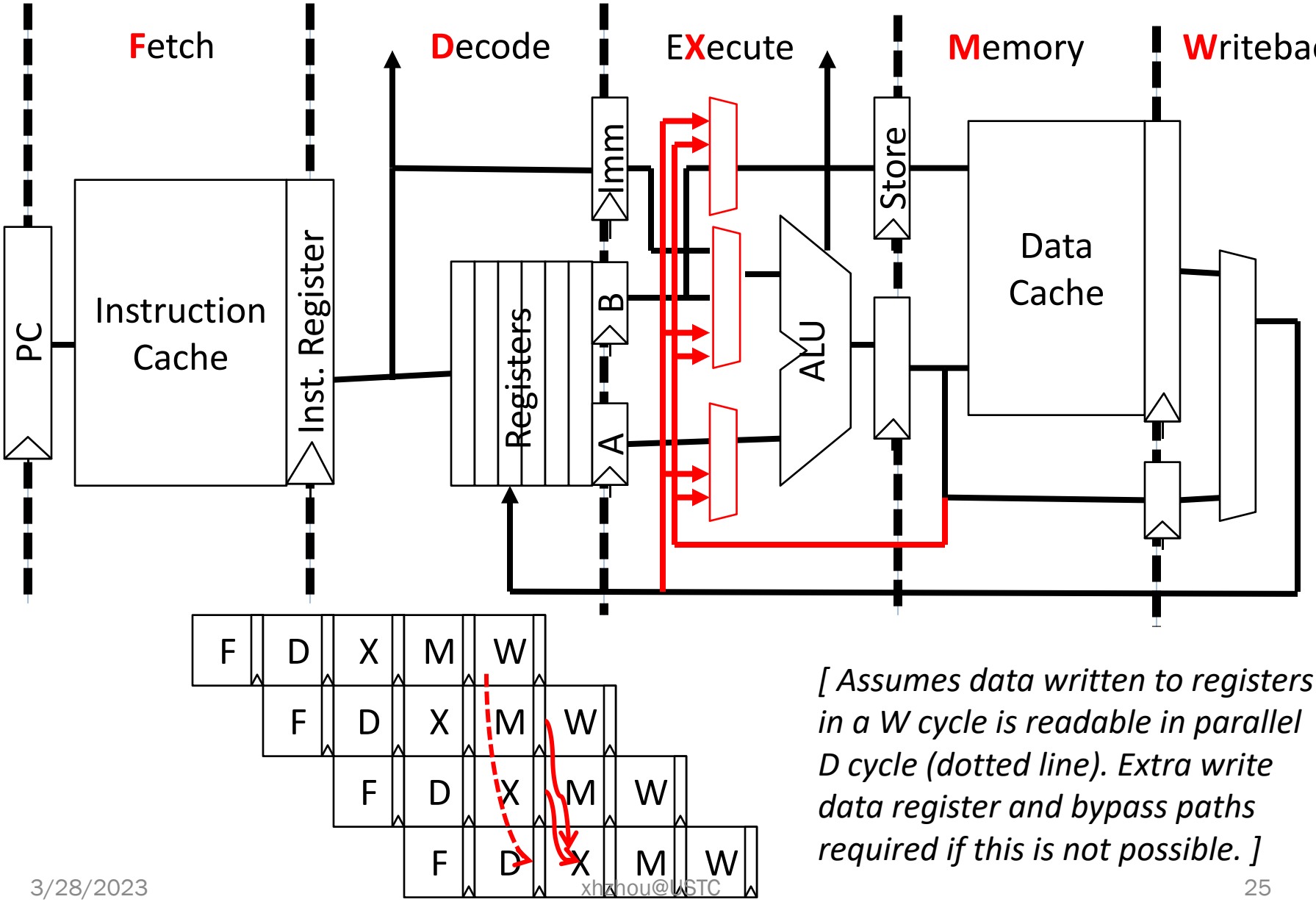


Bypass around ALU with no bubbles

Example Bypass Path



Fully Bypassed Data Path



[Assumes data written to registers in a W cycle is readable in parallel D cycle (dotted line). Extra write data register and bypass paths required if this is not possible.]

针对数据相关的值猜测执行

- **不等待产生结果的指令产生值，直接猜测值继续**

- **这种技术，仅在某些情况下可以使用:**
  - 分支预测
  - 堆栈指针更新
  - 存储器地址消除歧义（Memory address disambiguation）

采用软件方法避免数据相关

Try producing fast code for

a = b + c;

d = e – f;

assuming a, b, c, d ,e, and f in memory.

Slow code:

```
LW  Rb,b
LW  Rc,c
ADD     Ra,Rb,Rc
SW      a,Ra
LW  Re,e
LW  Rf,f
SUB     Rd,Re,Rf
SW d,Rd
```

Fast code:

```
LW      Rb,b
LW      Rc,c
LW      Re,e
ADD     Ra,Rb,Rc
LW      Rf,f
SW      a,Ra
SUB     Rd,Re,Rf
SW      d,Rd
```

假设：

1、使用Load的结果中间需要至少1条指令的间隔

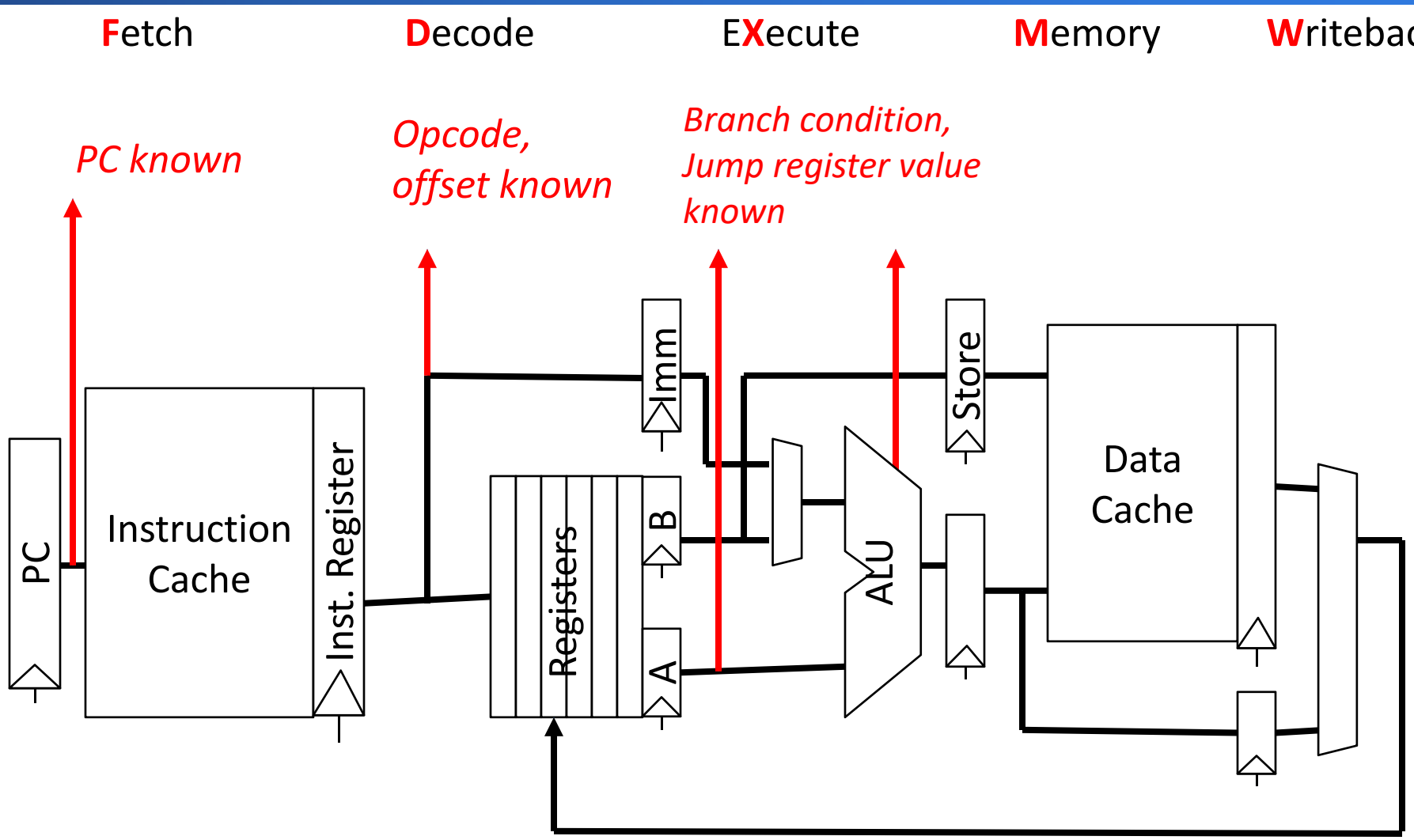
2、运算的结果，写入内存至少需要1条指令的间隔

Control Hazards

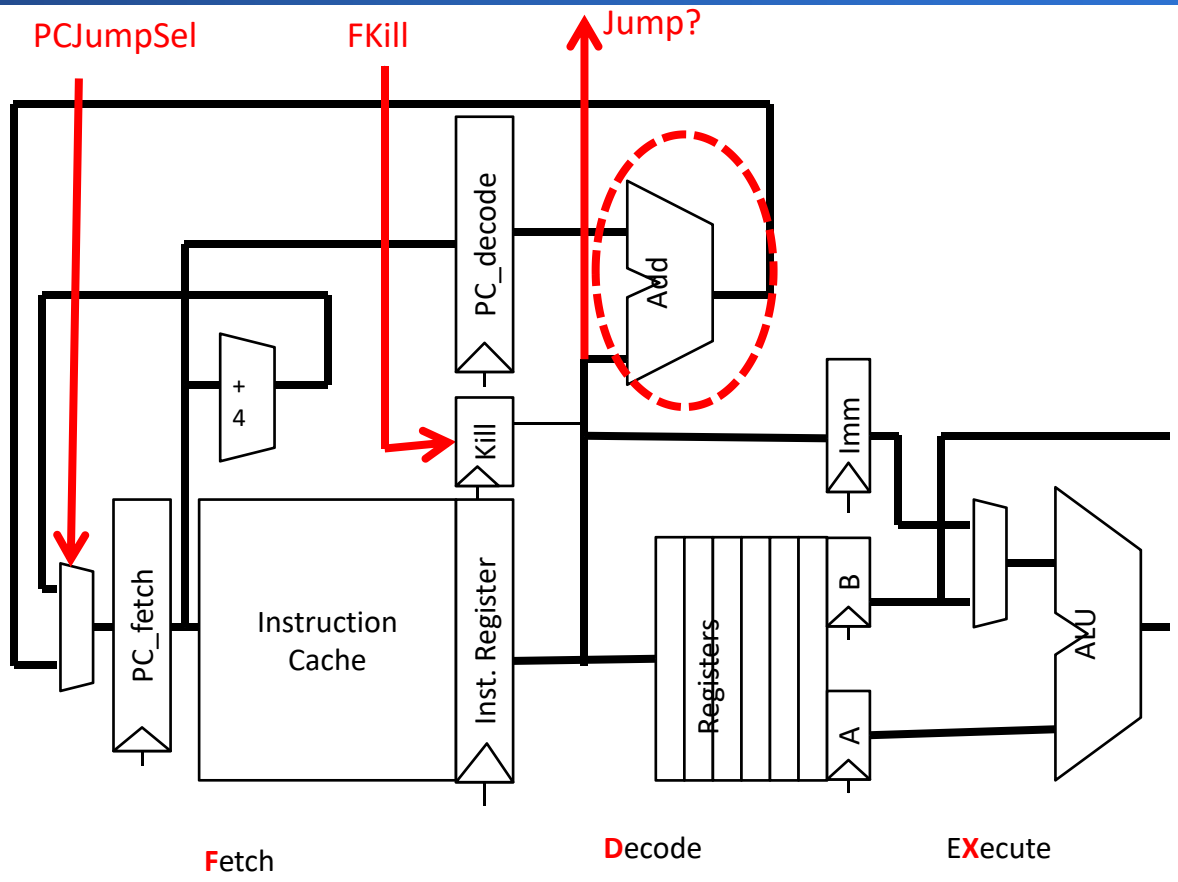
如何计算下一条指令地址（next PC)

- **无条件直接转移**
  - Opcode, PC, and offset
- **基于基址寄存器的无条件转移**
  - Opcode, Register value, and offset
- **条件转移**
  - Opcode, Register (for condition), PC and offset
- **其他指令**
  - Opcode and PC (and have to know it’s not one of above)

Control flow information in pipeline

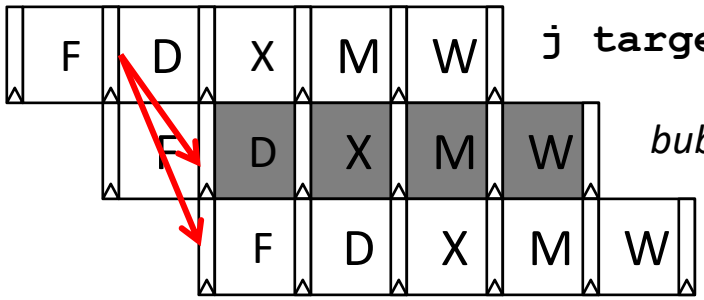


RISC-V Unconditional PC-Relative Jumps



改进的数据通路

[Kill bit turns instruction into a bubble]



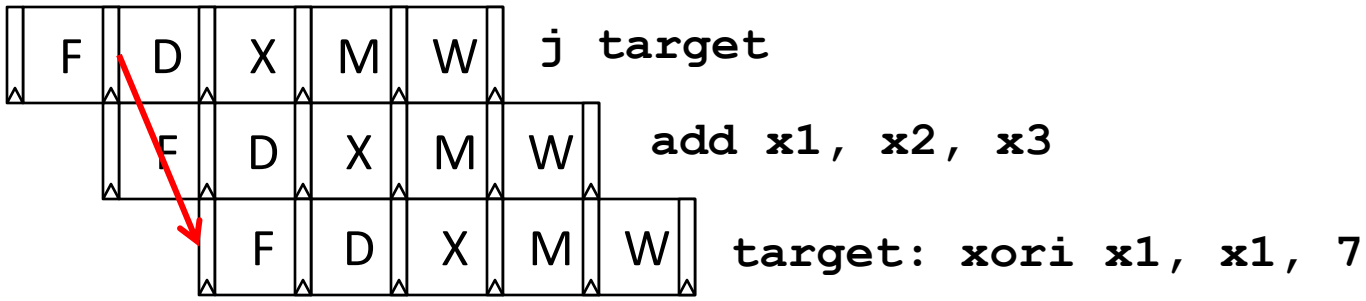

```
j target   ;等同于 jal x0,offset
           ; pc +=sext(offset)
```

bubble

```
target: add x1, x2, x3
```

Branch Delay Slots

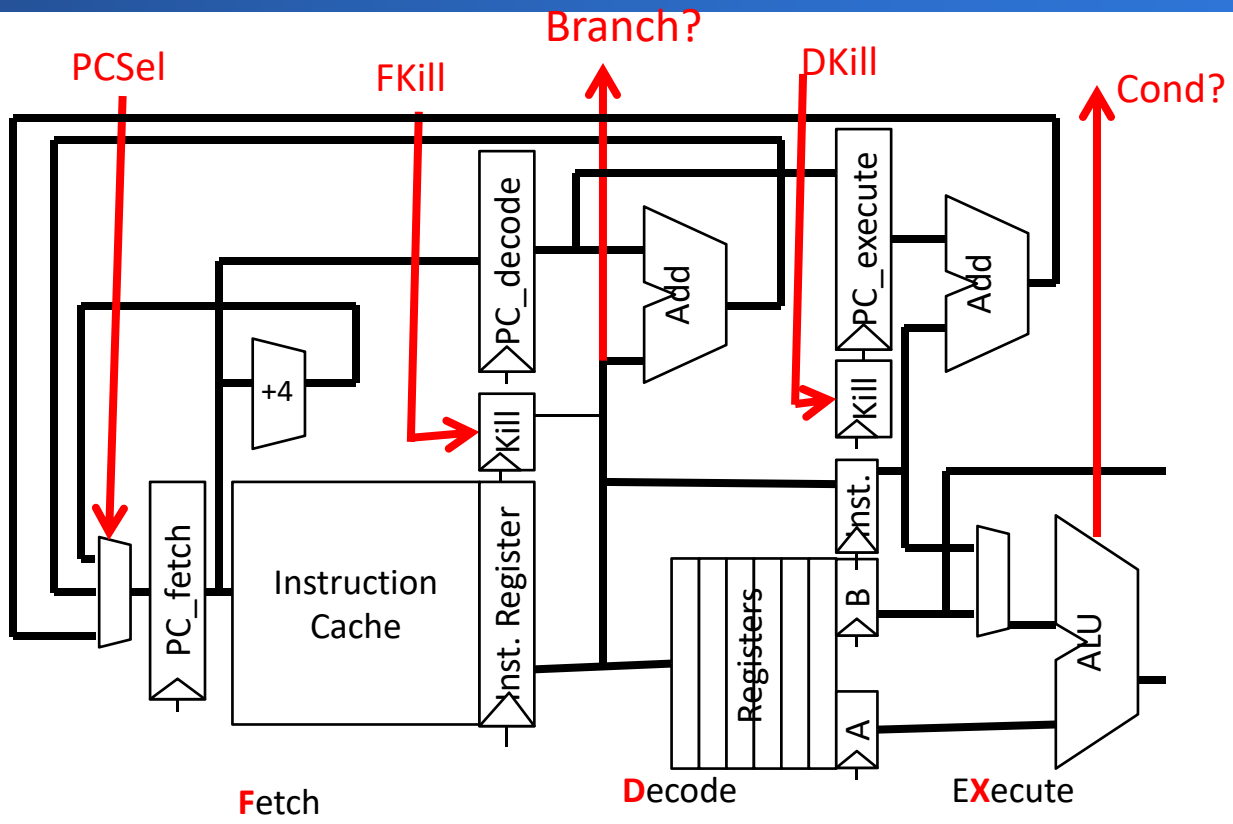
- 早期的RISC机器的延迟槽技术—改变ISA语义，在分支/跳转后的**延迟槽**中指令总是在控制流发生变化之前执行:
  - 0x100 j target
  - 0x104 add x1, x2, x3 // Executed before target
  - …
  - 0x205 target: xori x1, x1, 7
- 软件必须用有用的工作填充延迟槽（delay slots），或者用显式的NOP指令填充延迟槽

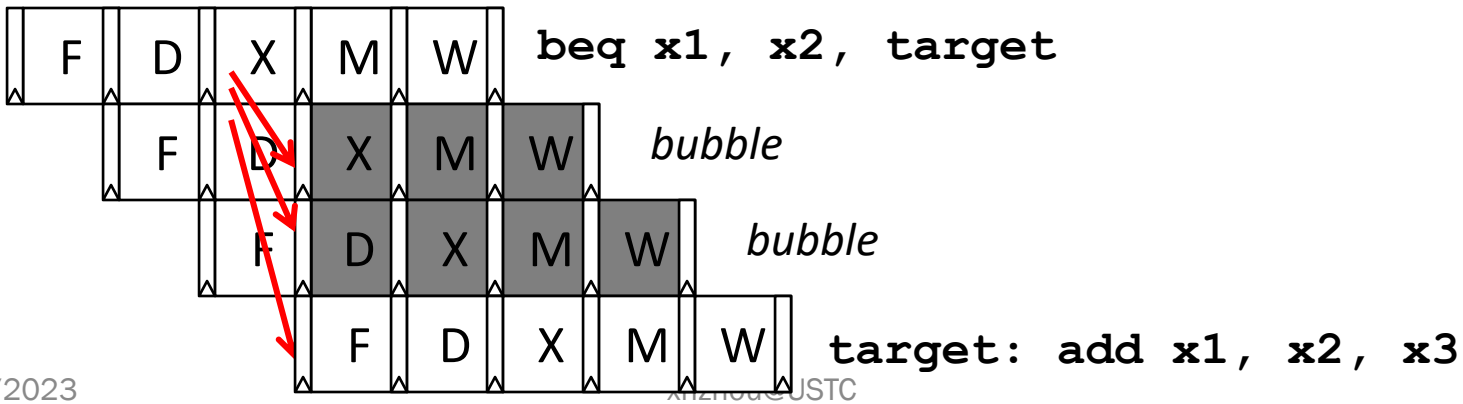


Post-1990 RISC ISAs 取消了延迟槽

- **性能问题**
  - 当延迟槽中填充了NOPs指令后，增加了I-cache的失效率
  - 即使延迟槽中只有一个NOP，I-cache失效导致机器等待
- **使先进的微体系架构复杂化**
  - 例如4发射30段流水线
- **分支预测技术的进步减少了采用延迟槽技术的动力**

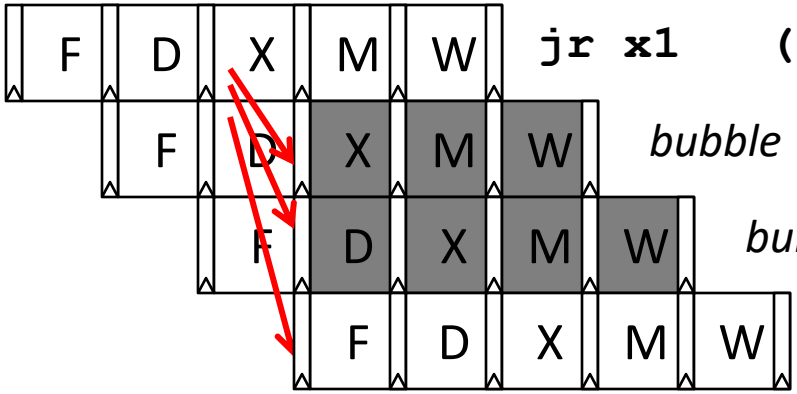
RISC-V Conditional Branches





Pipelining for Jump Register

- **Register value obtained in execute stage**



`jr x1` (等同于 `jalr x0,0(x1); pc=0(x1)`)

bubble

bubble

`target: add x5, x6, x7`

小结：解决控制相关的方法

- **#1: Stall 直到分支方向确定**
  - **可通过修改数据通路，减少stall**
- **#2: 预测分支失败**
  - 直接执行后继指令
  - 如果分支实际情况为分支成功，则撤销流水线中的指令对流水线状态的更新
  - 要保证：分支结果出来之前不会改变处理机的状态，以便一旦猜错时，处理机能够回退到原先的状态。
- **#3: 预测分支成功**
  - 前提：先知道分支目标地址，后知道分支是否成功
- **#4: 延迟转移技术**

为什么在经典的五段流水线中指令不能在每个周期都被分发(CPI>1)

- **采用全定向路径可能代价太高而无法实现**
  - 通常提供经常使用的定向路径
  - 一些不经常使用的定向路径可能会增加时钟周期的长度，从而抵消降低CPI的好处
- **Load操作有两个时钟周期的延迟**
  - Load指令后的指令不能马上使用Load的结果
  - MIPS-I ISA 定义了延迟槽, 软件可见的流水线冲突 (由编译器调度无关的指令或插入NOP指令避免冲突），MIPS-II中取消了延迟槽语义（硬件上增加流水线interlocks机制)
    - MIPS:“Microprocessor without Interlocked Pipeline Stages”
- **Jumps/Conditional branches 可能会导致流水线断流（bubbles)**
  - 如果没有延迟槽，则stall后续的指令

带有软件可见的延迟槽有可能需要执行大量的由编译器插入的NOP指令
NOP指令降低了CPI，增加了程序中执行的指令条数

3.1 基本流水线

基本概念 → 核心问题 Hazards → 性能分析

流水线的性能分析

- **基本度量参数：①吞吐率，②加速比，③效率**
- **吞吐率：在单位时间内流水线所完成的任务数量或输出结果的数量。**

$$TP = \frac{n}{T_K}$$

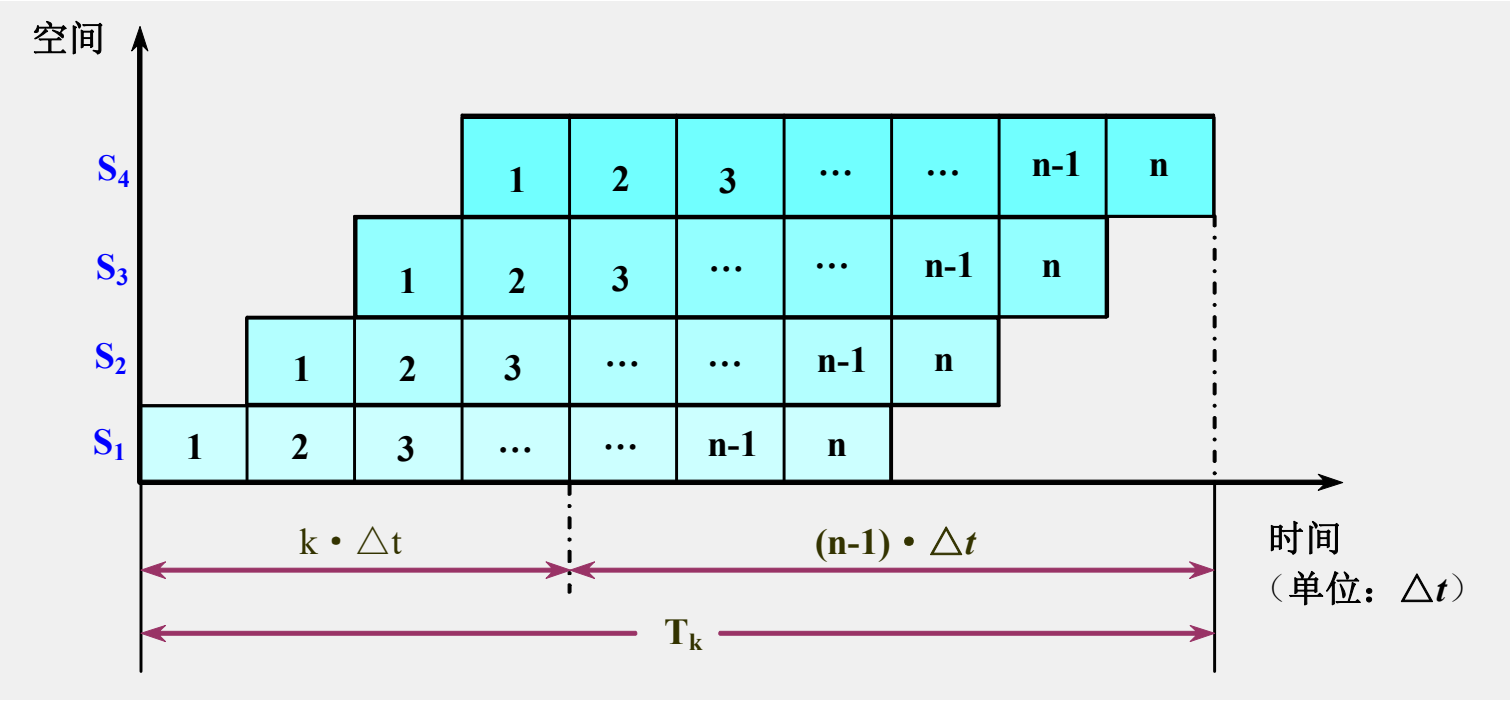
n：任务数

T_k：处理完成n个任务所用的时间

流水线技术提高系统的任务吞吐率

1. 各段时间均相等的流水线

– 各段时间均相等的流水线时空图



吞吐率

- 流水线完成n个连续任务所需要的总时间（假设一条k段线性流水线）

  Tk = kΔt + (n-1)Δt = (k + n-1)Δt

- 流水线的实际吞吐率

$$TP = \frac{n}{(k+n-1)\Delta t}$$

- 最大吞吐率: 流水线在连续流动达到稳定状态后所得到的吞吐率。

$$TP_{\max} = \lim_{n\to\infty} \frac{n}{(k+n-1)\Delta t} = \frac{1}{\Delta t}$$

$$TP = \frac{n}{k+n-1} TP_{\max}$$

S4				1	2	3	4	5	..	..	..	n-1	n
S3			1	2	3	4	5	..	..	..	n-1	n	
S2		1	2	3	4	5	..	..	..	n-1	n		
S1	1	2	3	4	5	..	..	..	n-1	n			

TP与Tpmax的关系

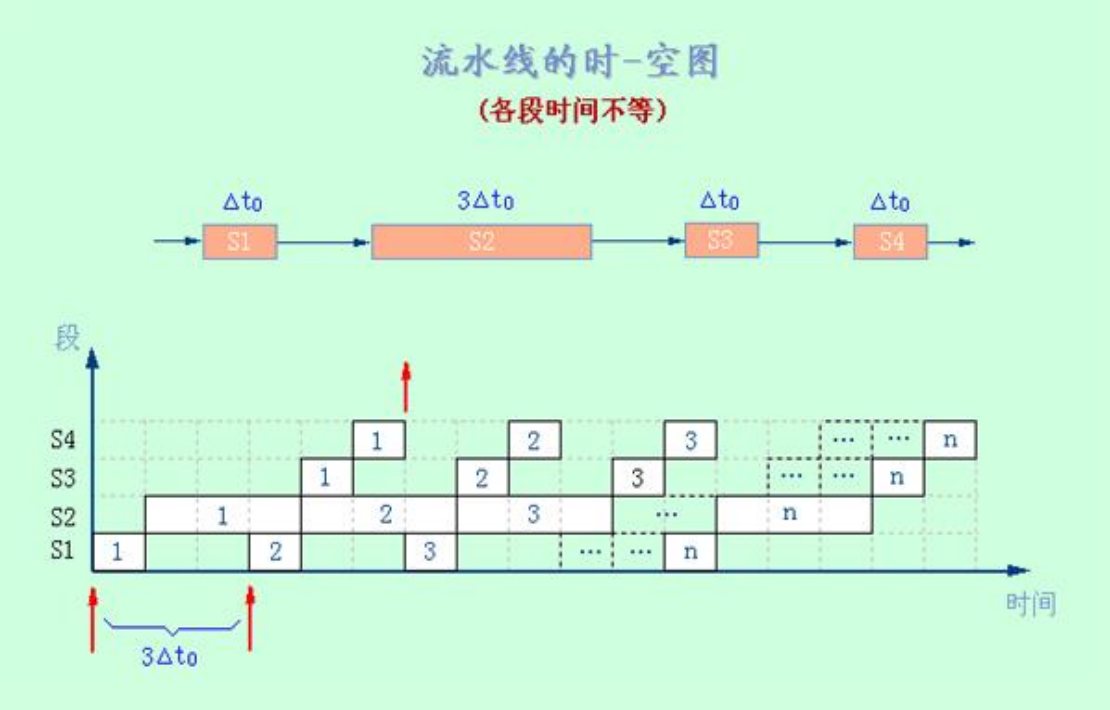
– 最大吞吐率与实际吞吐率的关系

$$TP = \frac{n}{k+n-1} TP_{\max}$$

- 流水线的实际吞吐率小于最大吞吐率
  - 与每个段的时间有关
  - 与流水线的段数k以及输入到流水线中的任务数n有关
- 只有当n>>k时，才有TP≈TPmax。

流水线中的瓶颈——最慢的段

2. 各段时间不完全相等的流水线



- 各段时间不等的流水线及其时空图
  - 一条4段的流水线
  - S1，S3，S4各段的时间：Δt
  - S2的时间：3Δt （瓶颈段）
- 流水线中这种时间最长的段称为流水线的瓶颈段

– 各段时间不等的流水线的实际吞吐率：

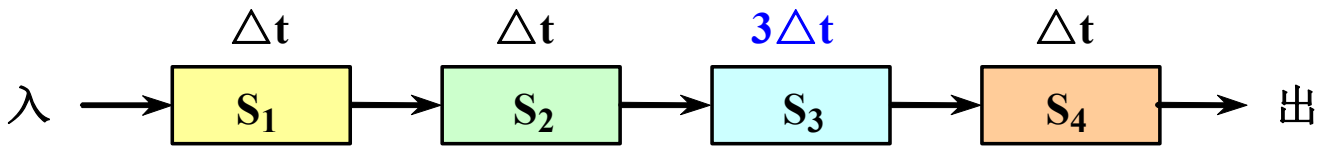
（Δti为第i段的时间，共有k个段）

$$TP = \frac{n}{\sum_{i=1}^{k} \Delta t_i + (n-1)\max(\Delta t_1, \Delta t_2, \cdots, \Delta t_k)}$$

– 流水线的最大吞吐率为

$$TP_{\max} = \frac{1}{\max(\Delta t_1, \Delta t_2, \cdots \Delta t_k)}$$

– 例如：一条4段的流水线中，S1，S2，S4各段的时间都是Δt，唯有S3的时间是3Δt。



最大吞吐率为

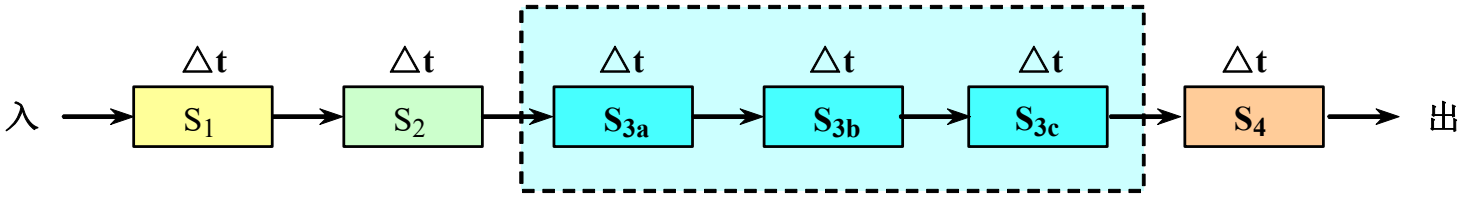
$$TP_{\max} = \frac{1}{3\Delta t}$$

3. 解决流水线瓶颈问题的常用方法

- 细分瓶颈段

  例如：对前面的4段流水线

  把瓶颈段S_3细分为3个子流水线段：S_{3a}，S_{3b}，S_{3c}

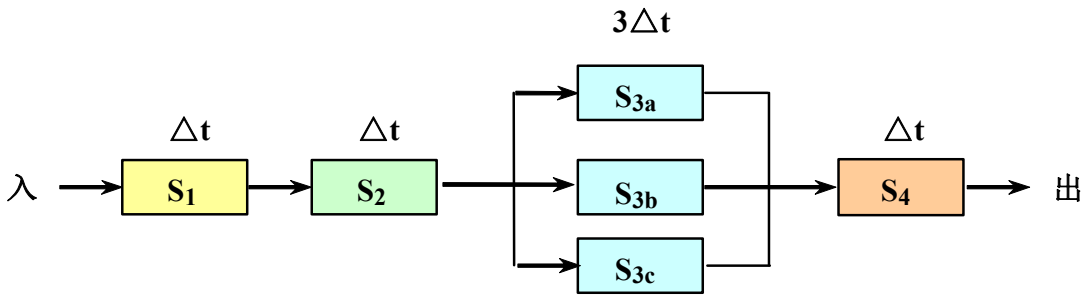


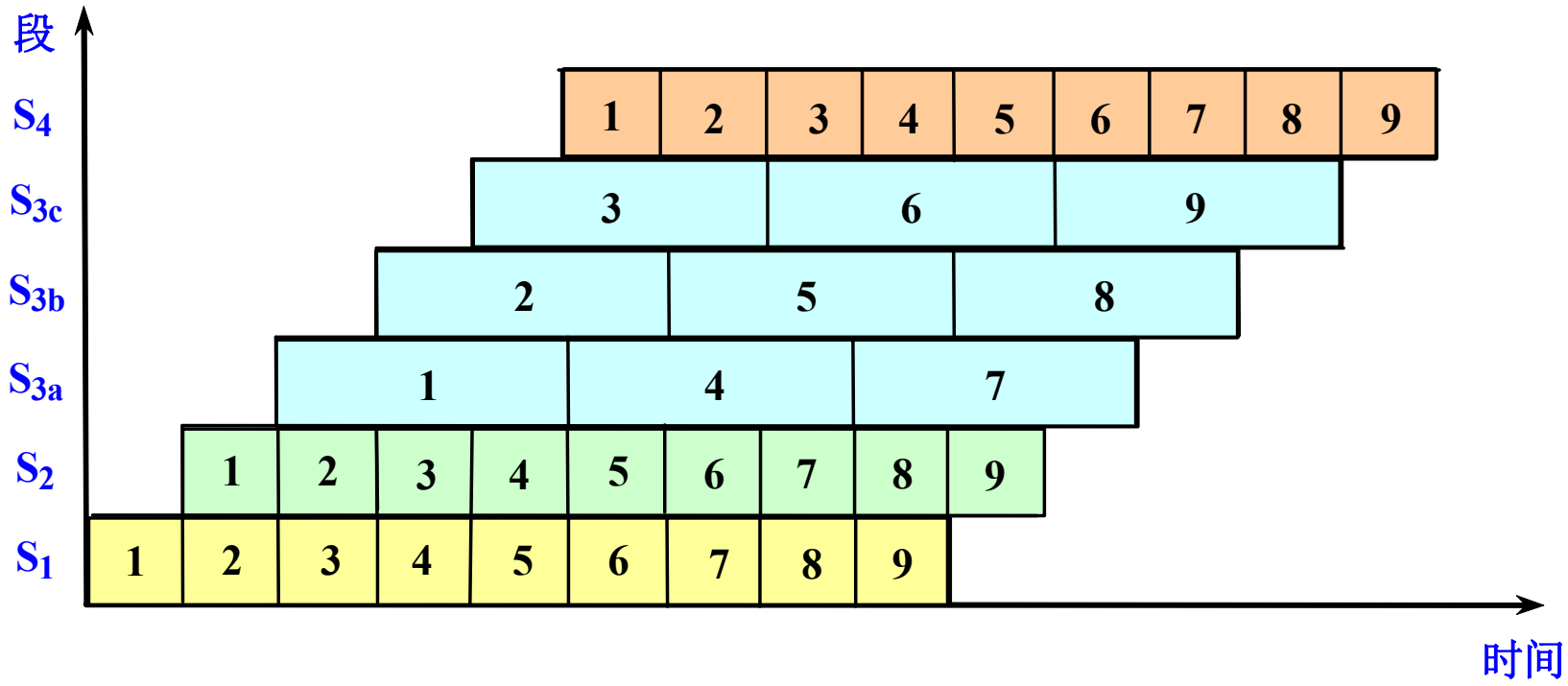
改进后的流水线的吞吐率：

$$TP_{\max} = \frac{1}{\Delta t}$$

– 重复设置瓶颈段

- 缺点：控制逻辑比较复杂，所需的硬件增加了。
- 例如：对前面的4段流水线
- 　　重复设置瓶颈段S3：S3a，S3b，S3c





重复设置瓶颈段后的时空图

加速比

- **加速比：完成同样一批任务，不使用流水线所用的时间与使用流水线所用的时间之比。**
  - 假设：不使用流水线（即顺序执行）所用的间为Ts，使用流水线后所用的时间为Tk，则该流水线的加速比为

$$S = \frac{T_s}{T_k}$$

1. 流水线各段时间相等（都是Δt)

- k段流水线完成n个连续任务所需要的时间为

$$T_k = (k + n - 1)\Delta t$$

- 顺序执行n个任务所需要的时间

$$T_s = nk\Delta t$$

- 流水线的实际加速比为

$$S = \frac{nk}{k + n - 1}$$

- 最大加速比

$$S_{max} = \lim_{n \to \infty} \frac{nk}{k + n - 1} = k$$

- **当*n*>>*k*时，*S* ≈ *k***

思考：流水线的段数愈多愈好?

2. 流水线的各段时间不完全相等时

- 一条k段流水线完成n个连续任务的实际加速比为

$$S=\frac{n\sum_{i=1}^{k}\Delta t_i}{\sum_{i=1}^{k}\Delta t_i+(n-1)\max(\Delta t_1,\Delta t_2,\cdots,\Delta t_k)}$$

效率

- **效率：流水线中的设备实际使用时间与整个运行时间的比值，即流水线设备的利用率。**
  - 由于流水线有通过时间和排空时间，所以在连续完成n个任务的时间内，各段并不是满负荷地工作。
- **各段时间相等**
  - 各段的效率ei相同

$$e_1 = e_2 = \cdots = e_k = \frac{n\Delta t}{T_k} = \frac{n}{k+n-1}$$

– 整条流水线的效率为

$$E=\frac{e_1+e_2+\cdots+e_k}{k}=\frac{ke_1}{k}=\frac{kn\Delta t}{kT_k}$$

可以写成

$$E=\frac{n}{k+n-1}$$

最高效率为

$$E_{\max}=\lim_{n\to\infty}\frac{n}{k+n-1}=1$$

当$n>>k$时，$E\approx1$。

– 当流水线各段时间相等时，流水线的效率与吞吐率成正比。

E=TP△t

$$E=\frac{n}{k+n-1} \qquad TP=\frac{n}{(k+n-1)\Delta t}$$

- **流水线的效率是流水线的实际加速比S与它的最大加速比k的比值。**

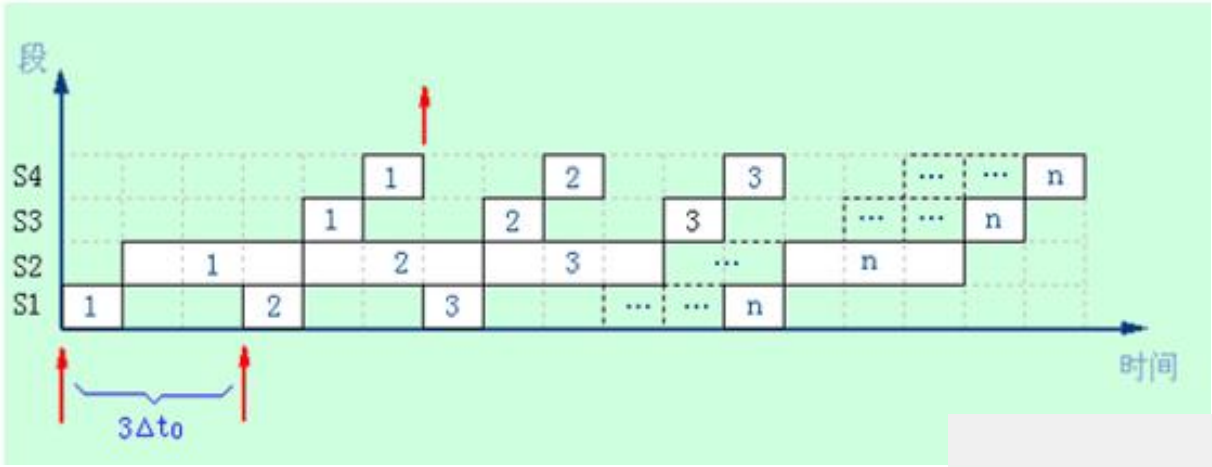
$$S=\frac{nk}{k+n-1} \qquad E=\frac{S}{k}$$

当*E*=1时，*S*=*k*，实际加速比达到最大。

- **从时空图上看，效率就是n个任务占用的时空面积和k个段总的时空面积之比。**

$$E=\frac{n\text{个任务实际占用的时空区}}{k\text{个段总的时空区}}$$

当各段时间不相等时



$$E=\frac{n\cdot\sum_{i=1}^{k}\Delta t_i}{k\left[\sum_{i=1}^{k}\Delta t_i+(n-1)\cdot\max(\Delta t_1,\Delta t_2,\cdots,\Delta t_k)\right]}$$

Summary

- **实际吞吐率：假设k段，完成n个任务，单位时间所实际完成的任务数。**
- **加速比: k段流水线的速度与等功能的非流水线的速度之比。**
- **效率：流水线的设备利用率。**

$$TP_{\max} = \frac{1}{\max\{\Delta t_i\}}$$

$$TP = \frac{n}{\sum_{i=1}^{k} \Delta t_i + (n-1)\Delta t_j}$$

$$S = \frac{n \cdot \sum_{i=1}^{k} \Delta t_i}{\sum_{i=1}^{k} \Delta t_i + (n-1)\Delta t_j}$$

$$E = \frac{n \cdot \sum_{i=1}^{k} \Delta t_j}{k \cdot [\sum_{i=1}^{k} \Delta t_i + (n-1)\Delta t_j]}$$

$$E = TP \cdot \frac{\sum_{i=1}^{k} \Delta t_i}{k}$$

流水线的加速比计算

$$CPI_{pipelined} = \text{Ideal CPI} + \text{Average Stall cycles per Inst}$$

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{unpipelined}}{\text{Cycle Time}_{pipelined}}$$

For simple RISC pipeline, CPI = 1:

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{unpipelined}}{\text{Cycle Time}_{pipelined}}$$

结构相关对性能的影响

- **例如: 如果每条指令平均访存1.3 次，而每个时钟周期只能访存一次，那么**
  - 在其他资源100%利用的前提下，平均 CPI ≥ 1.3

例如： Dual-port vs. Single-port

- **机器A: Dual ported memory（“Harvard Architecture”）**
- **机器B: Single ported memory**
- **存在结构相关的机器B的时钟频率是机器A的时钟频率的1.05倍**
- **Ideal CPI = 1**
- **在机器B中load指令会引起结构相关，所执行的指令中Loads指令占 40%**

Average instruction time = CPI * Clock cycle time

无结构相关的机器A：

Average Instruction time = Clock cycle time

存在结构相关的机器B：

Average Instruction time = (1+0.4*1) * clock cycle time /1.05
= 1.3 * clock cycle time

评估减少分支策略的效果

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

Scheduling scheme	*Branch penalty*	*CPI*	*speedup v. unpipelined*	*speedup v. stall*
Stall pipeline	3	1.42	3.5	1.0
Predict taken	1	1.14	4.4	1.26
Predict not taken	1	1.09	4.5	1.29
Delayed branch	0.5	1.07	4.6	1.31

1.14 = 1 + 1*14%*100%

1.09 = 1+1*14%*65%

1.07 = 1+ 0.5*14%

Conditional & Unconditional = 14%, 65% change PC

小结

- **流水线技术要点**
  - 多个任务重叠（并发/并行）执行，但使用不同的资源
  - 流水线技术提高整个系统的吞吐率，不能缩短单个任务的执行时间
  - 其潜在的加速比 = 流水线的级数
  - 由于存在相关(hazards)问题，会导致流水线停顿
    - Hazards 问题：流水线的执行可能会导致对资源的访问冲突，或破坏对资源的访问顺序
- **指令流水线**
  - 通过指令重叠减小 CPI
  - 充分利用数据通路
    - 当前指令执行时，启动下一条指令
    - 其性能受限于花费时间最长的段
  - 检测和消除相关
  - 正常工作的基本条件
    - 增加寄存器文件保存当前段传送到下一段的数据和控制信息
    - 需要更高的存储器带宽

小结

- **流水线的性能评估**
  - 实际吞吐率：假设k段，完成n个任务，单位时间所实际完成的任务数。
  - 加速比: k段流水线的速度与等功能的非流水线的速度之比。
  - 效率：流水线的设备利用率
- **影响流水线性能的因素**
  - 流水线中的瓶颈——最慢的那一段
  - 流水段所需时间不均衡将降低加速比
  - 流水线存在装入时间和排空时间，使得加速比降低
- **如何有利于流水线技术的应用**
  - 指令格式简单
    - 固定长度、很少的指令格式
  - 只用Load/Store来进行存储器访问

Acknowledgements

- **These slides contain material developed and copyright by:**
  - John Kubiatowicz (UCB)
  - Krste Asanovic (UCB)
  - David Patterson (UCB)
  - Chenxi Zhang (Tongji)
- **UCB material derived from course CS152、CS252、CS61C**
- **KFUPM material derived from course COE501、COE502**