



中国科学技术大学
University of Science and Technology of China

计算机体系结构

Topic III: Pipelining



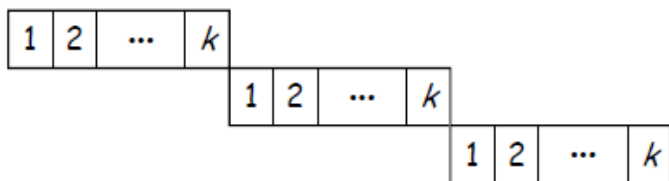
流水线的基本概念

- **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

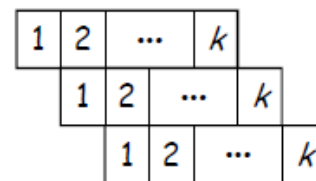
- **流水线执行模式是重叠执行模式**

- 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



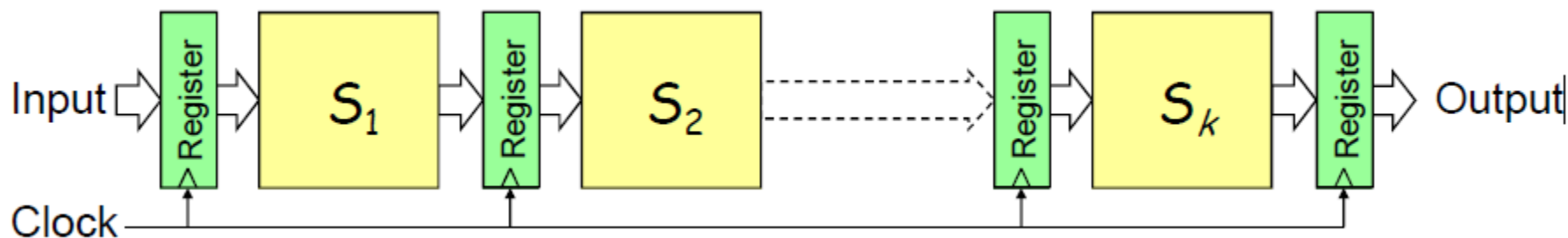
Pipelined Execution

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





流水线的性能

- 设 τ_i = time delay in stage S_i
- 时钟周期 $\tau = \max(\tau_i)$ 为最长的流水段延迟
- 时钟频率 $f = 1/\tau = 1/\max(\tau_i)$
- 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）**

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

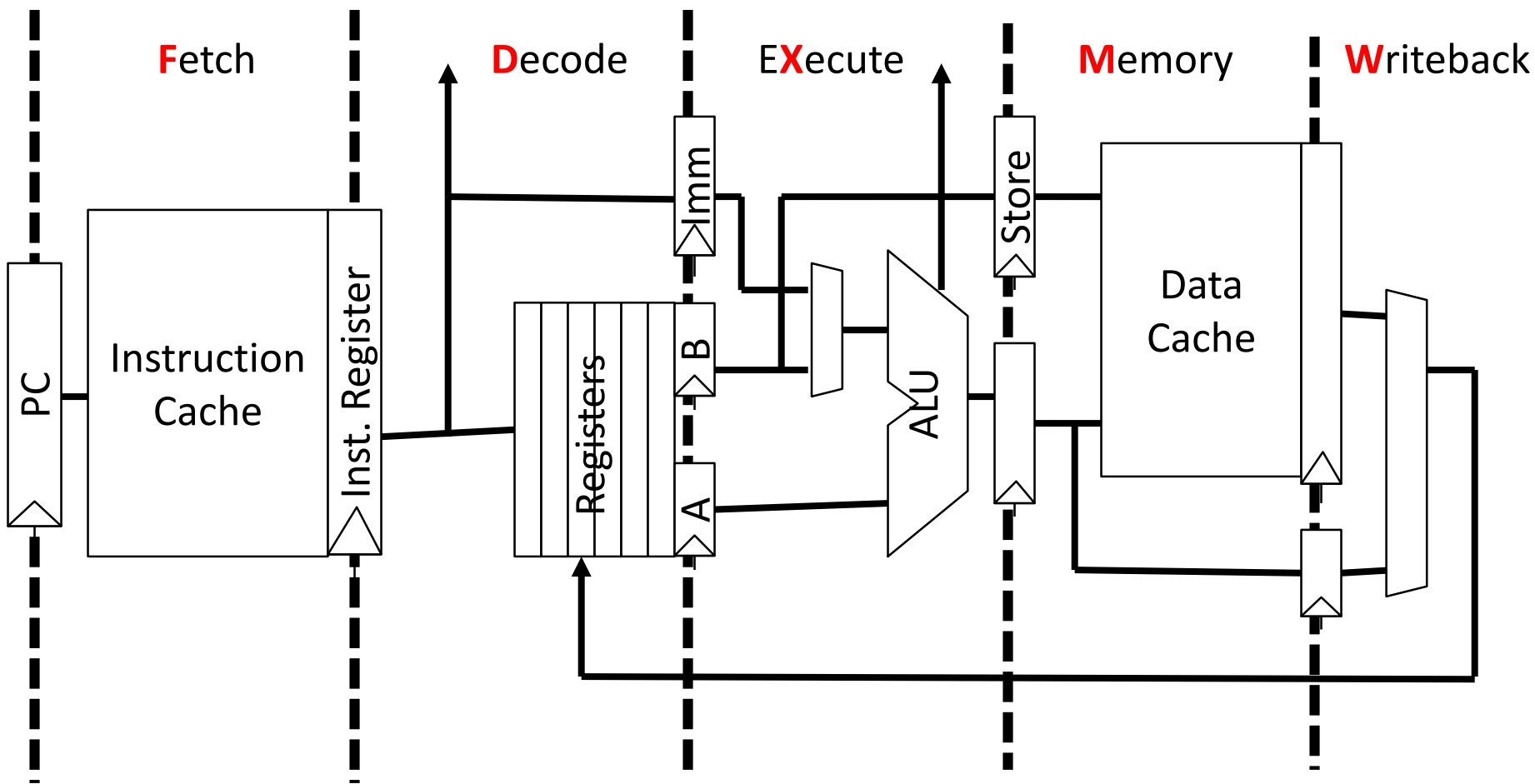


典型的RISC 5段流水线

- **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



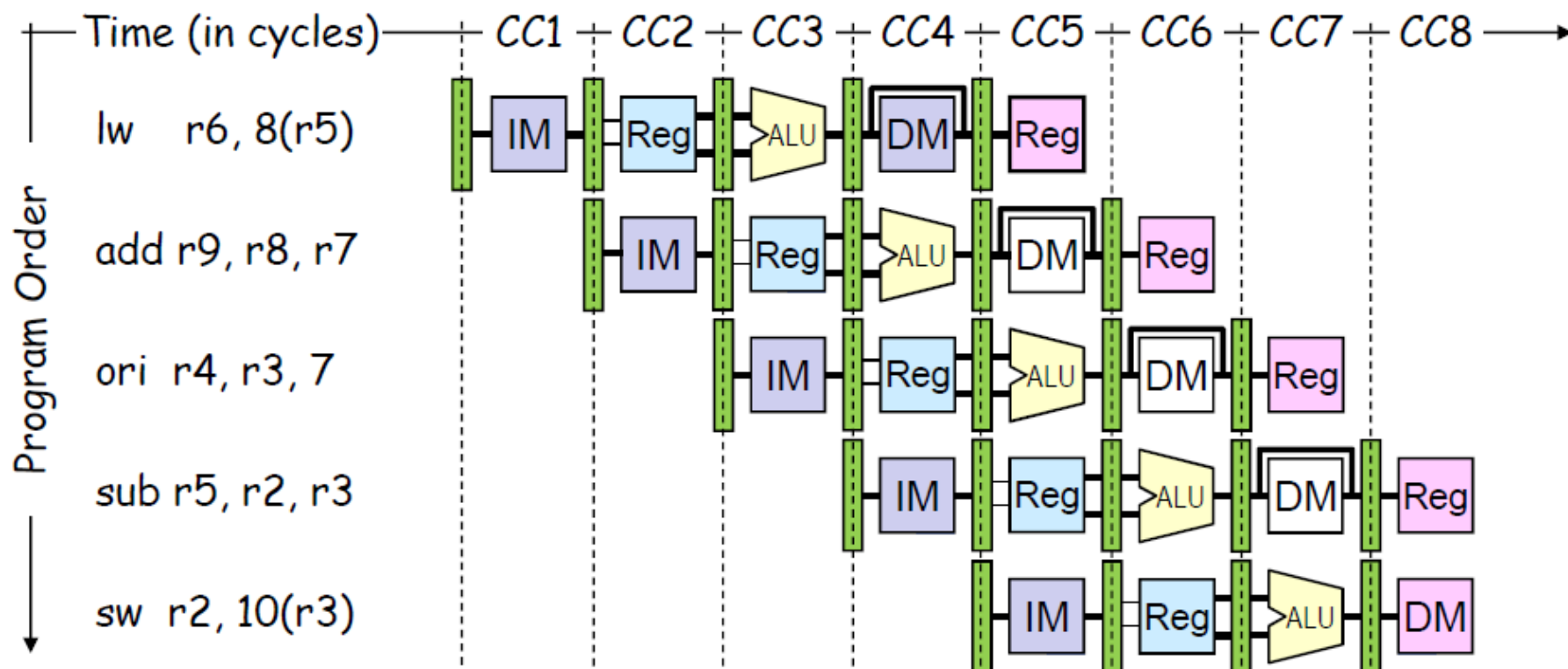
This version designed for regfiles/memories with synchronous reads and writes.



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





Recap: 流水线技术要点

• 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
- 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
- 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

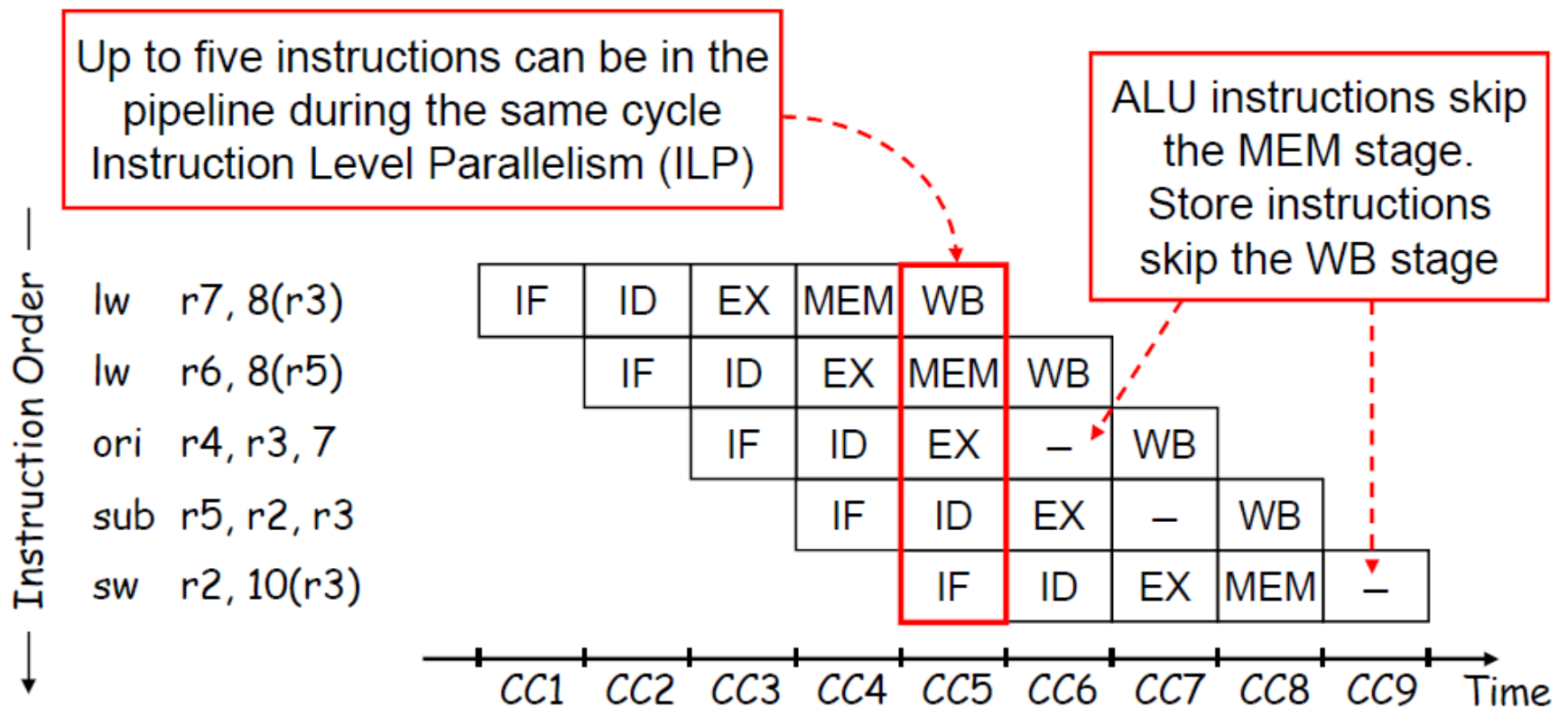
• 流水线正常工作的基本条件

- 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



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**



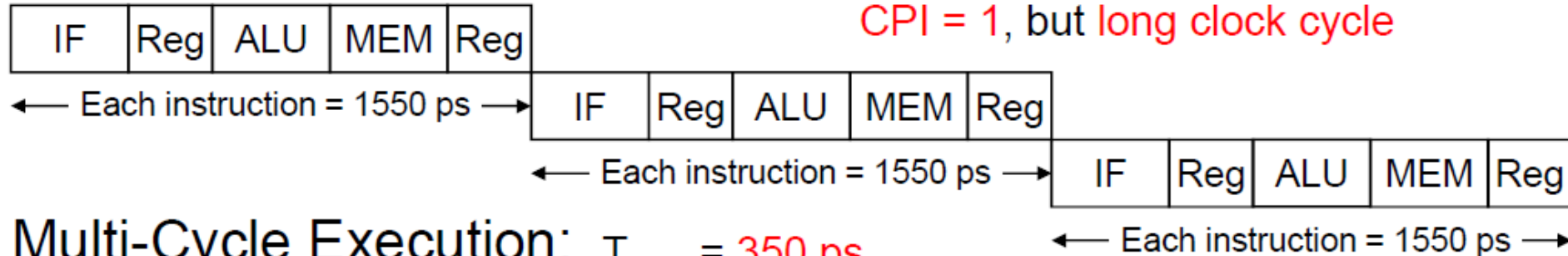


单周期、多周期、流水线控制性能比较

Single-Cycle Execution:

$$T_{\text{clock}} = 350 + 250 + 350 + 350 + 250 = 1550 \text{ ps}$$

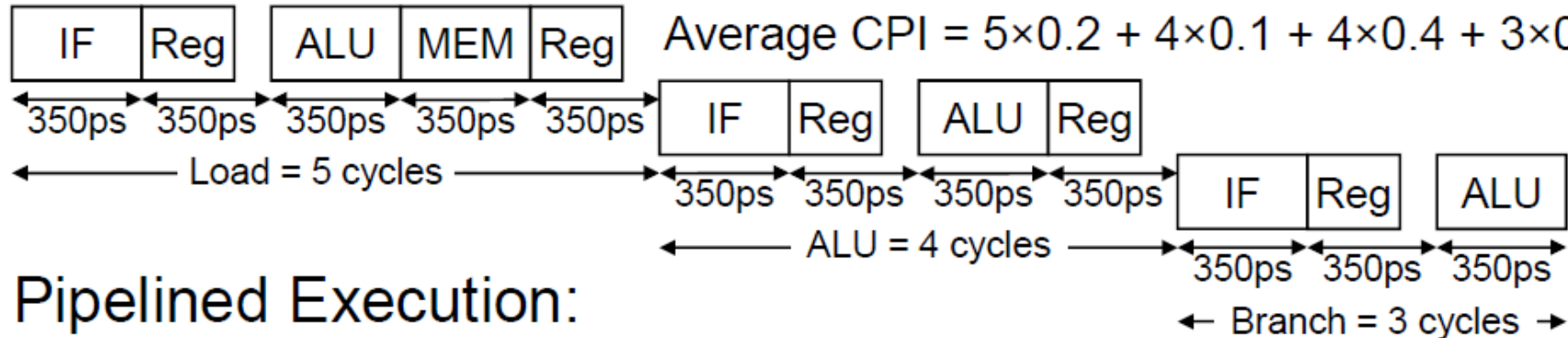
CPI = 1, but long clock cycle



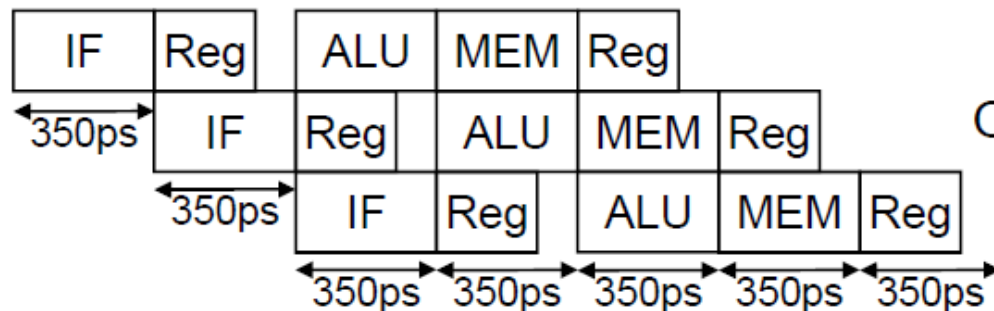
Multi-Cycle Execution:

$$T_{\text{clock}} = 350 \text{ ps}$$

$$\text{Average CPI} = 5 \times 0.2 + 4 \times 0.1 + 4 \times 0.4 + 3 \times 0.3 = 3.9$$



Pipelined Execution:



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

One instruction completes each cycle

Average CPI = 1

Ignore time to fill pipeline



流水线的相关 (Hazards)

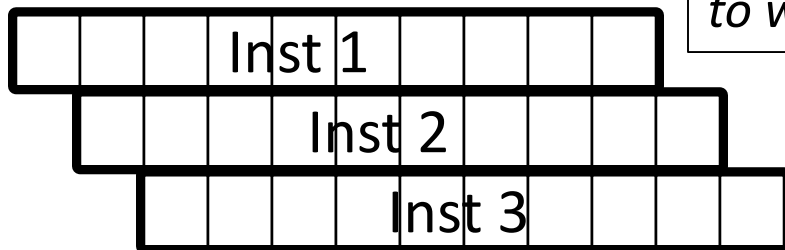
- **结构相关：流水线中一条指令可能需要另一条指令使用的资源**
- **数据和控制相关：一条指令可能依赖于先前的指令生成的内容**
 - 数据相关：依赖先前指令产生的结果（数据）值
 - 控制相关：依赖关系是如何确定下一条指令地址 (branches, exceptions)
- **处理相关的一般方法是插入bubble，导致 $CPI > 1$ (单发射理想CPI)**



Pipeline CPI Examples

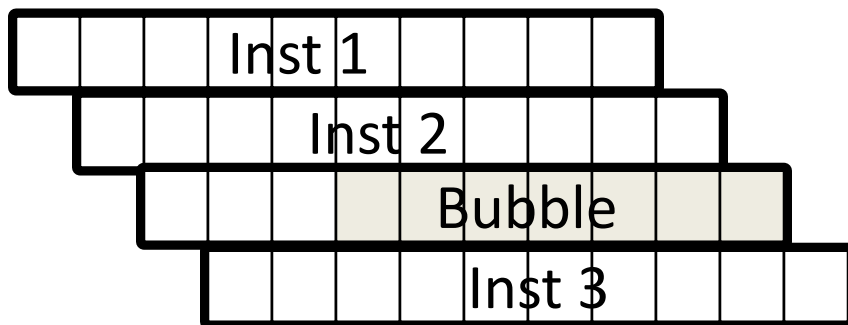
Time →

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



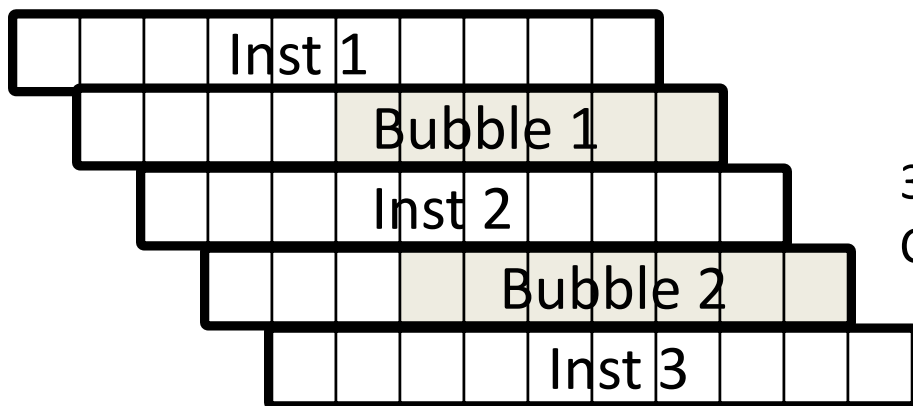
3 instructions finish in 3 cycles

$$\text{CPI} = 3/3 = 1$$



3 instructions finish in 4 cycles

$$\text{CPI} = 4/3 = 1.33$$



3 instructions finish in 5 cycles

$$\text{CPI} = 5/3 = 1.67$$



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

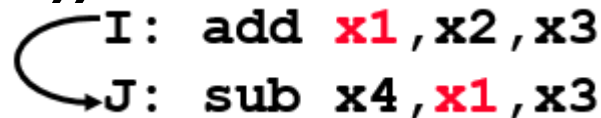


三种基本的数据相关

- 写后读相关(Read After Write (RAW))

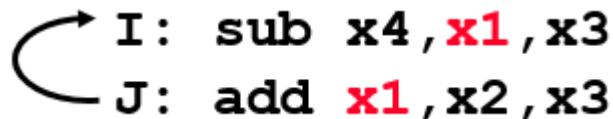
- 由于实际的数据交换需求而引起的

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



- 读后写相关 (Write After Read (WAR))

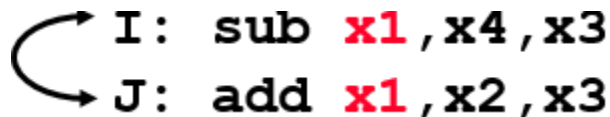
I: sub x4, **x1**, x3
J: add **x1**, x2, x3



- 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



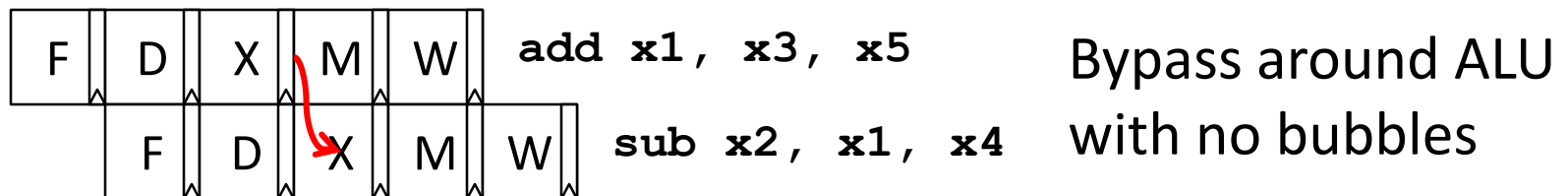
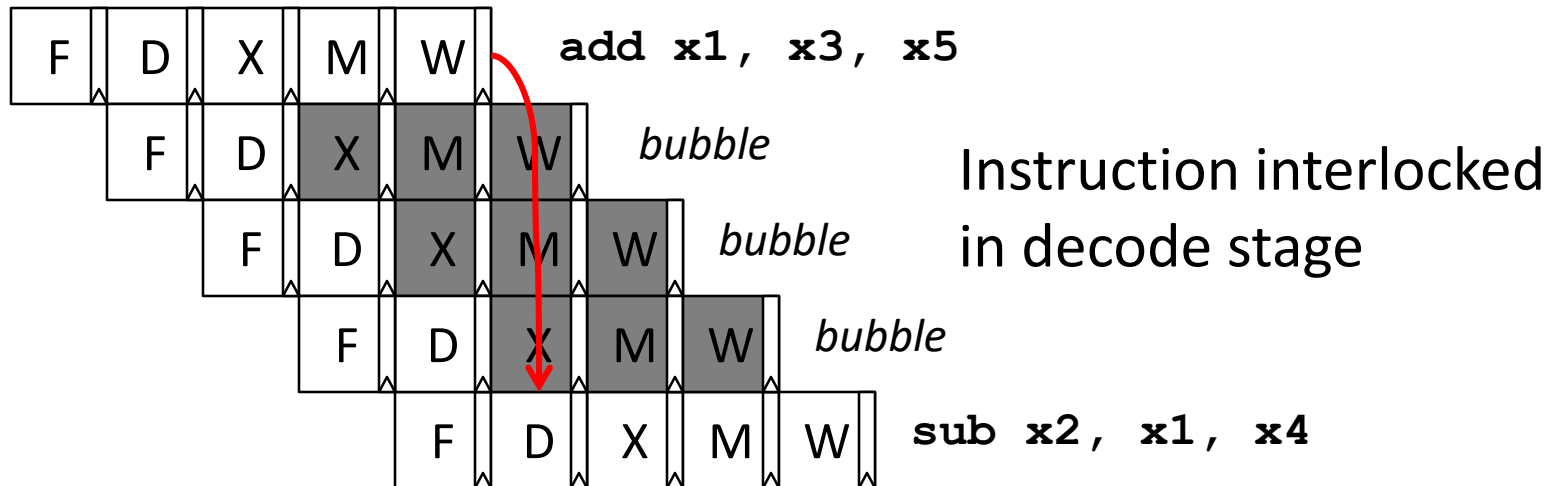
消减数据相关的三种策略

- **连锁机制 (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



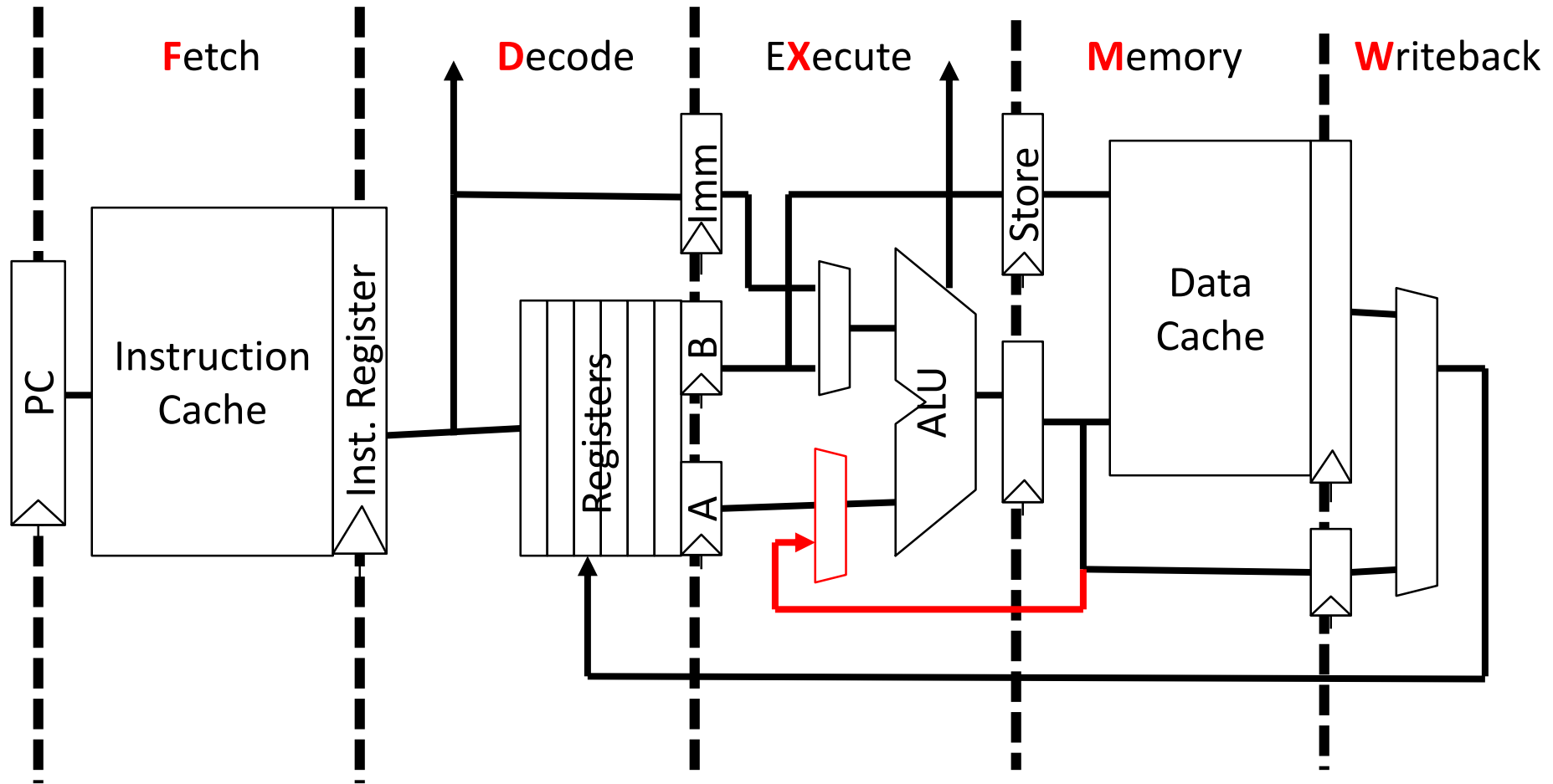
Interlocking Versus Bypassing

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



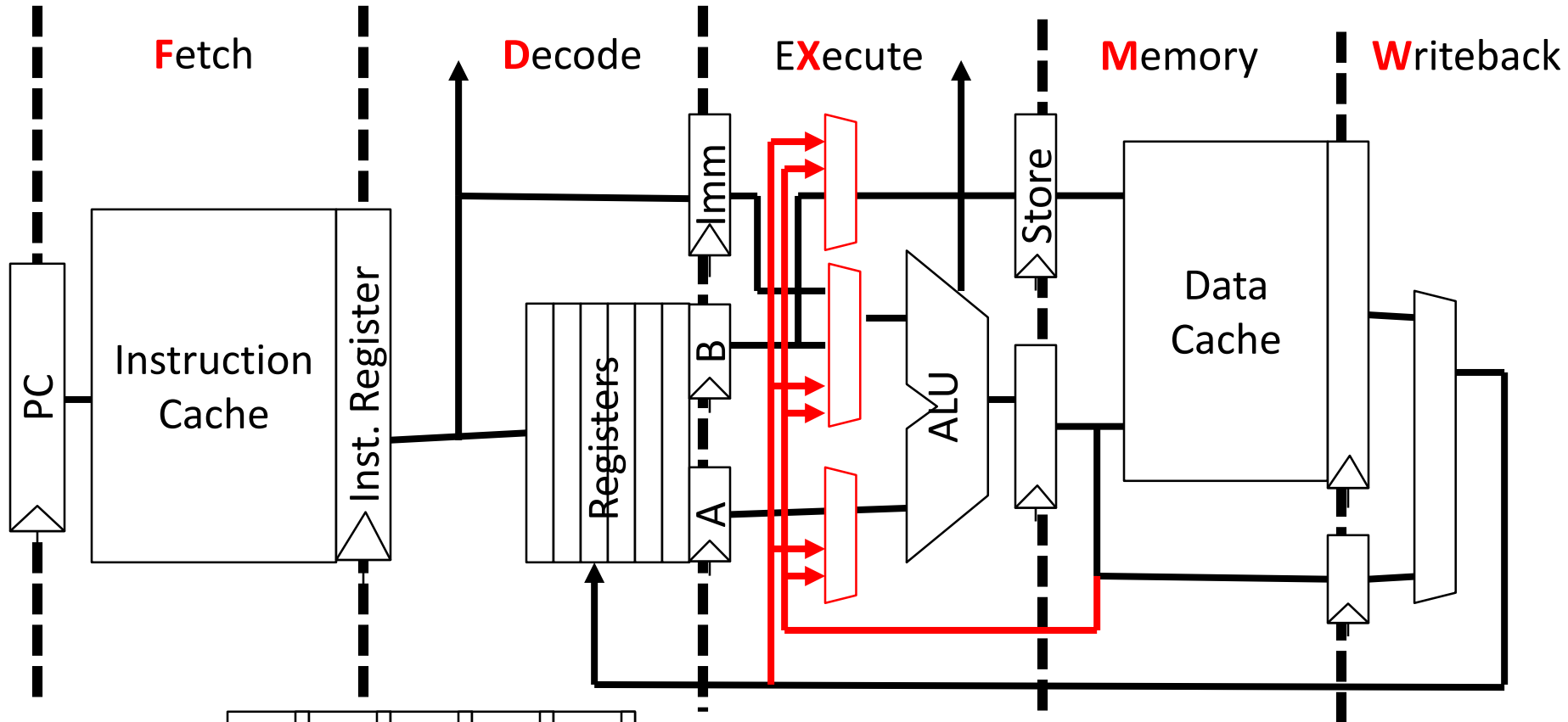


Example Bypass Path

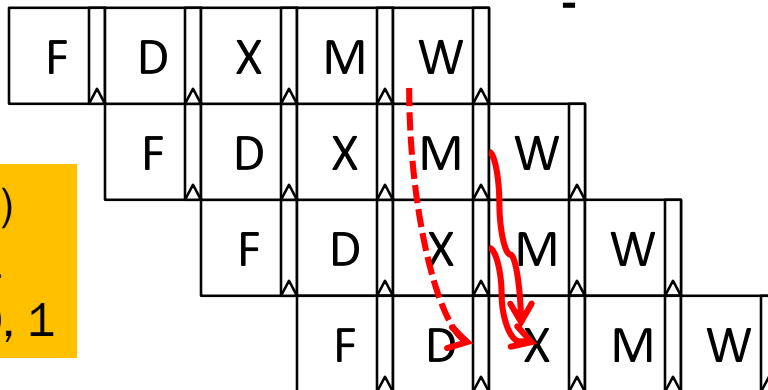




Fully Bypassed Data Path



```
lw $t0, 4($t1)
addi $t1, $t1
addi $t0, $t0, 1
```



[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，直接猜测一个值继续**
- **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

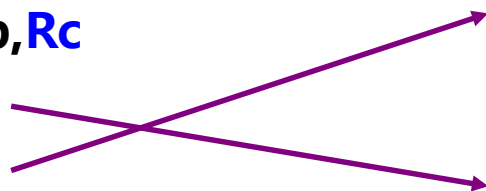
ADD Ra,Rb,Rc

LW Rf,f

SW a,Ra

SUB Rd,Re,Rf

SW d,Rd

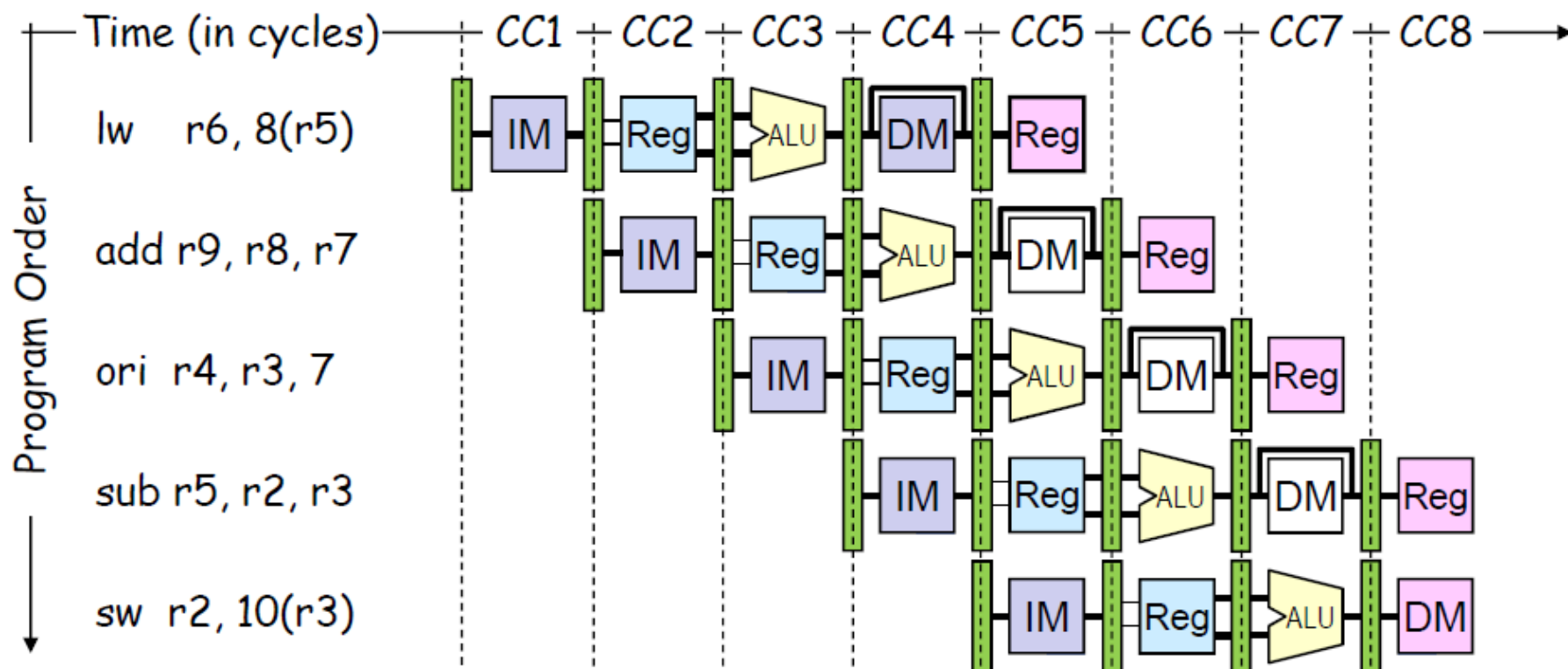




流水线的可视化表示

- **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





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)**



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

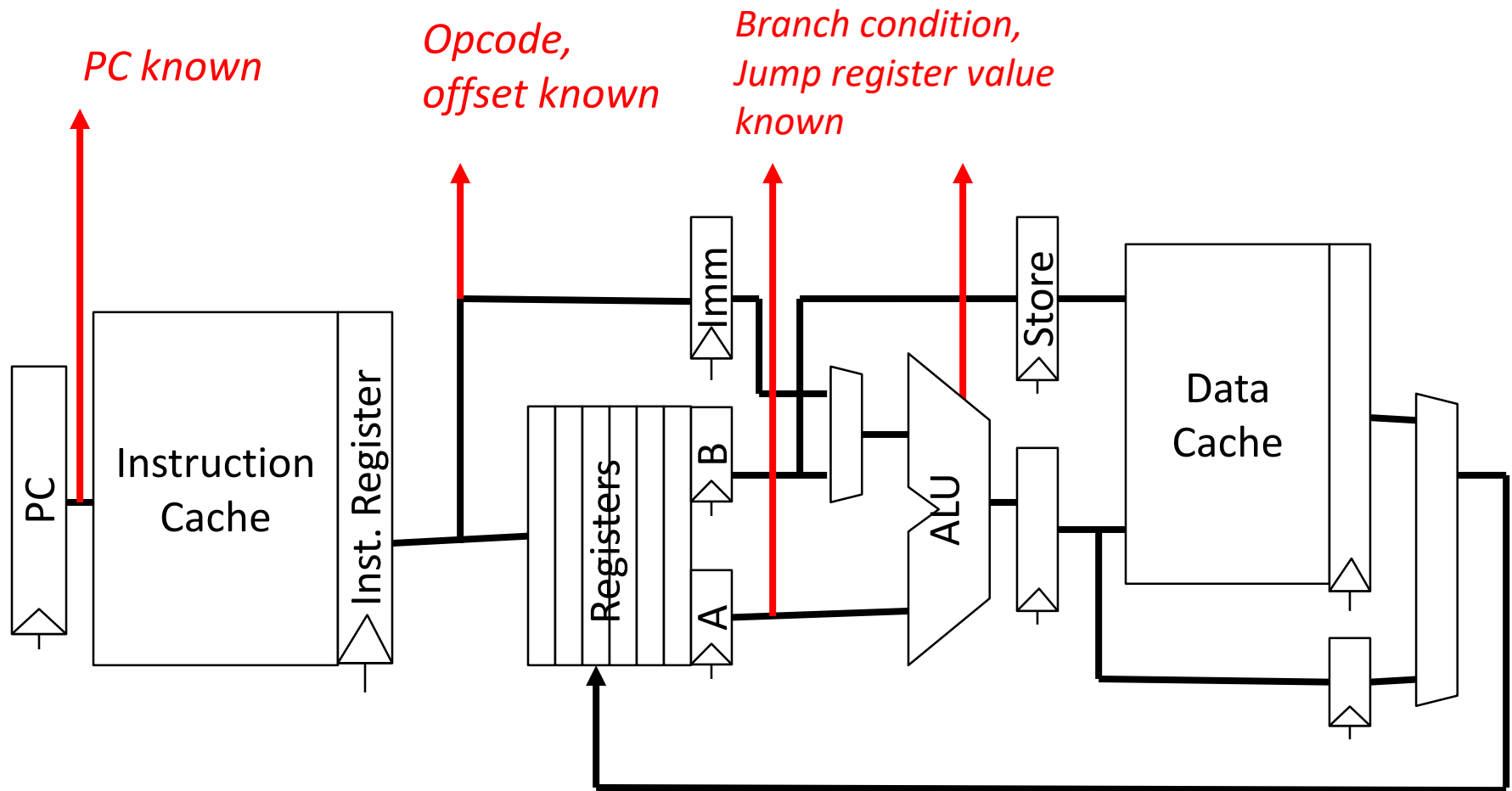
Fetch

Decode

EXecute

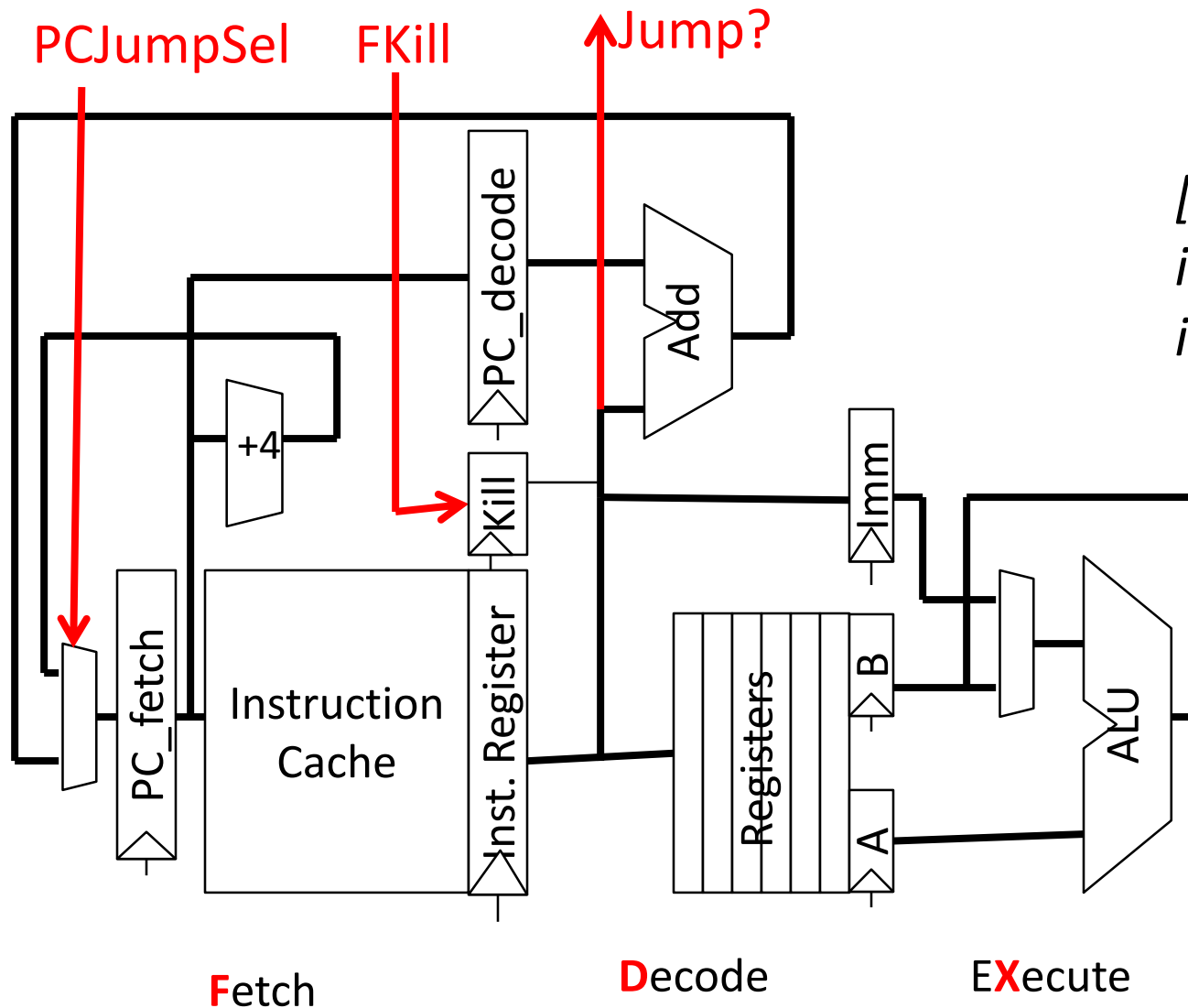
Memory

Writeback





RISC-V Unconditional PC-Relative Jumps



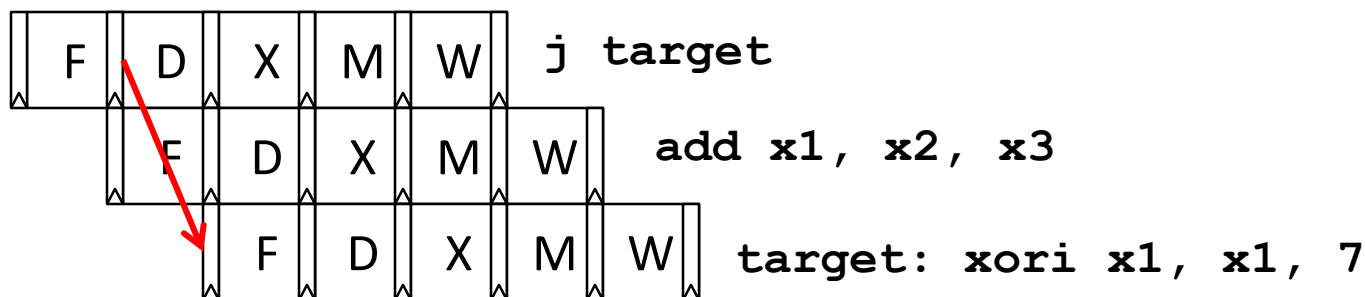
[Kill bit turns instruction into a bubble]





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**

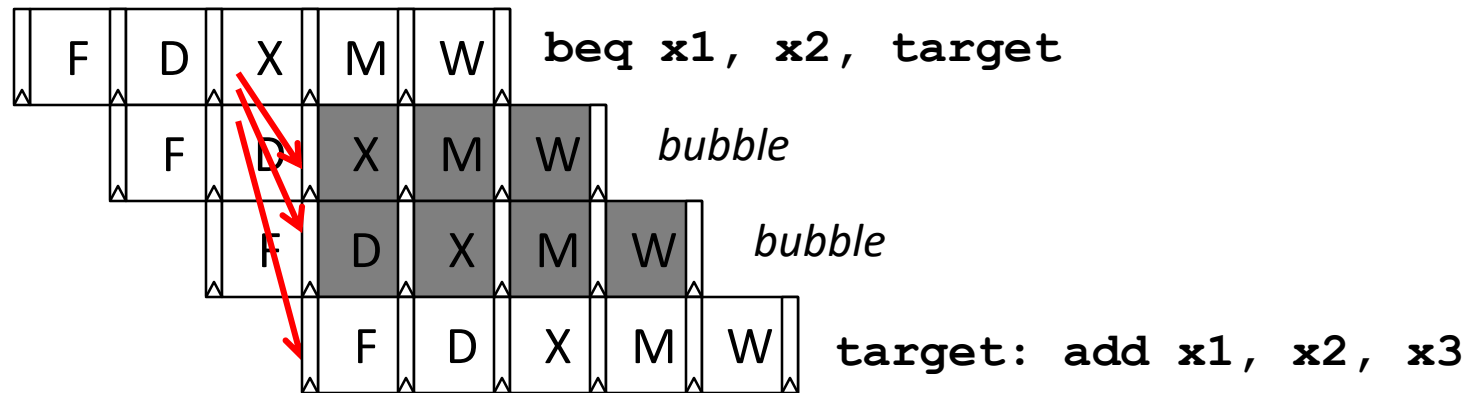
- 例如4发射30段流水线

- **好的分支预测技术减少了采用延迟槽技术的动力**





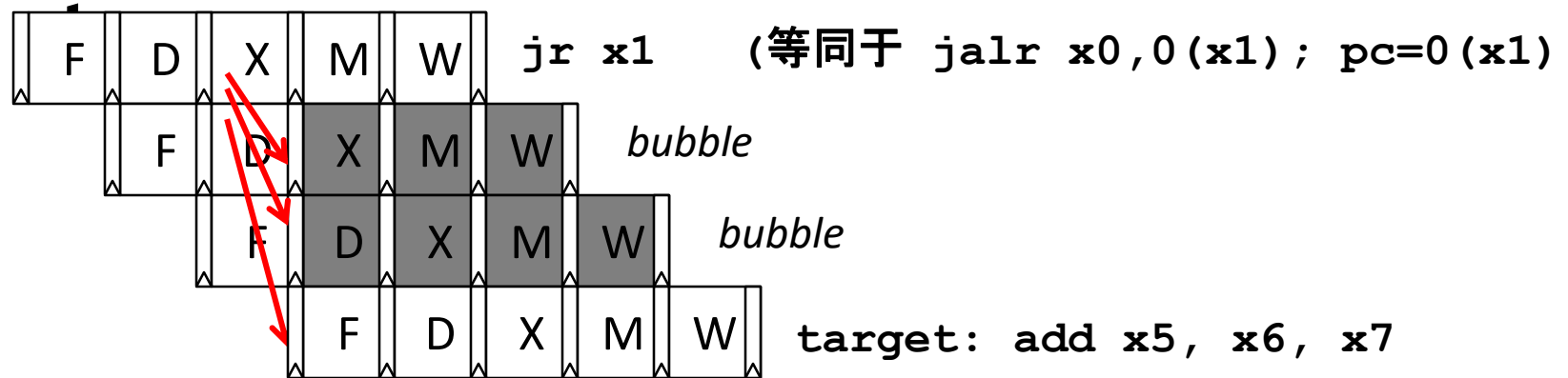
Pipelining for Conditional Branches





Pipelining for Jump Register

- **Target address obtained in execute**





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, 但是增加了程序中执行的指令条数



陷阱和中断

术语说明

- **异常 (Exception)** :由程序执行过程中引起的异常**内部事件**
 - E.g., page fault, arithmetic underflow
- **陷阱 (Trap)** :因异常情况而被迫将控制权移交给监控程序
 - Not all exceptions cause traps (c.f. IEEE 754 floating-point standard)
- **中断 (Interrupt)** :运行程序之外的**外部事件**, 导致控制权转移到监控程序
- **陷阱和中断通常由同样的流水线机制处理**

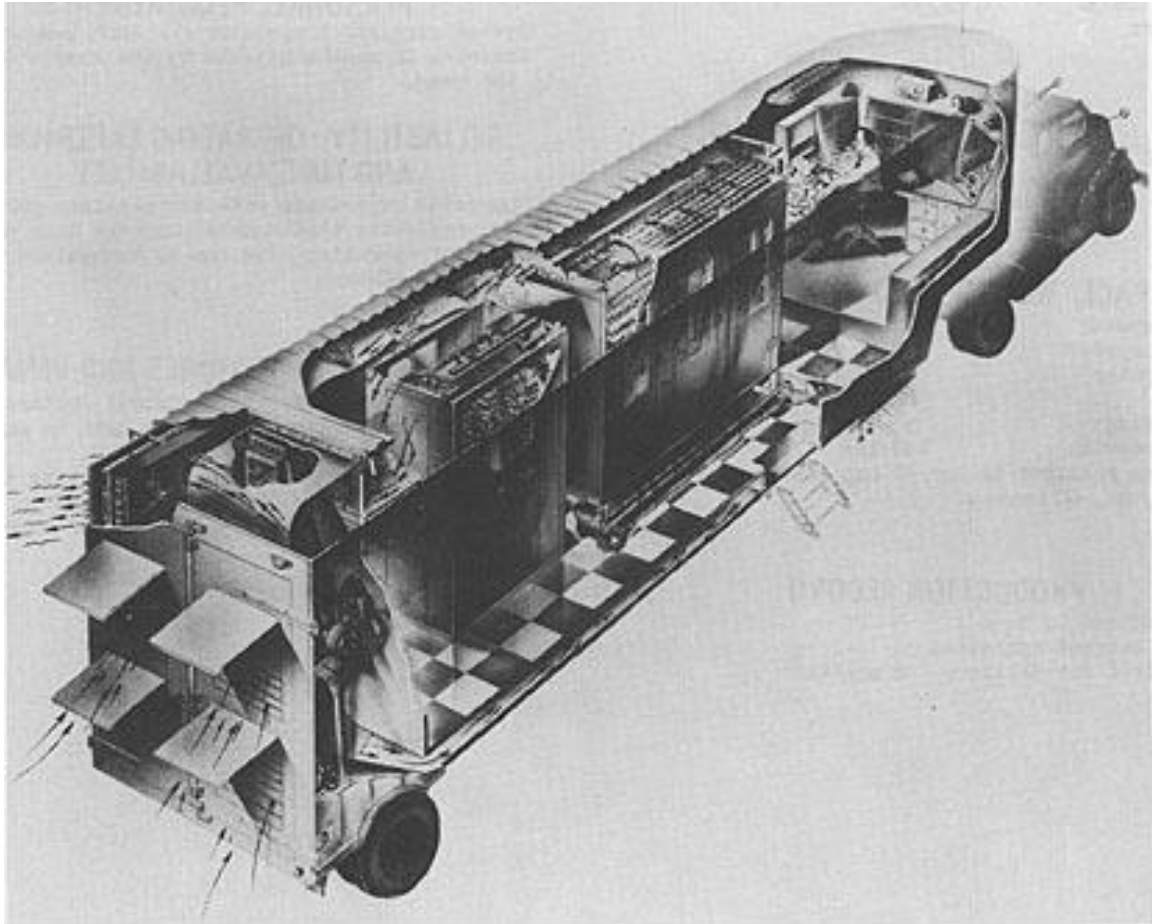


异常处理的发展历史

- **第一个带有陷阱 (traps) 的系统是Univac-I (1951年)**
 - 算术运算溢出有两种选择
 - 1.在地址0处触发一个两条指令的修复例程的执行 或者
 - 2.根据程序员的选择, 使计算机停止
 - 在后来的Univac 1103, 1955, 增加了外部中断机制
 - 用于实时采集风洞数据
- **第一个带有I/O 中断的系统是DYSEAC (1954)**
 - 带有两个程序计数器, 并且I/O信号引起这两个PC间的切换
 - 它也是第一个带有DMA (direct memory access by I/O device) 的系统
 - 同时, 也是第一台移动计算机 (两台半挂牵引车, 12 tons + 8 tons)



DYSEAC, first mobile computer!



- Carried in two tractor trailers, 12 tons + 8 tons
- Built for US Army Signal Corps

[Courtesy Mark Smotherman]

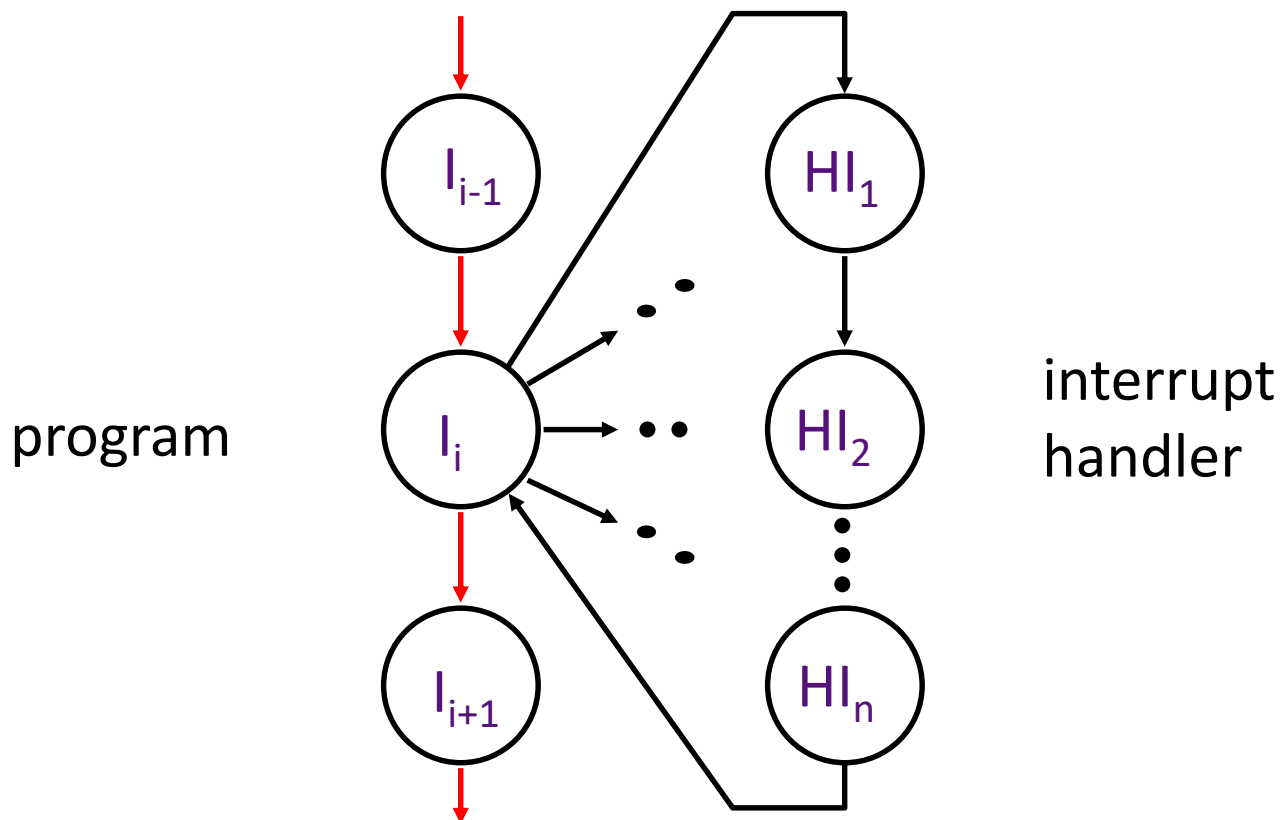


异步中断

- **I/O设备通过发出中断请求信号来请求处理**
- **当处理器决定响应该中断时**
 - 停止当前指令(I_i)的执行, 执行完当前指令前面的指令 (执行完 I_{i-1}) (精确中断)
 - 将 I_i 指令的PC值保存到专门寄存器(EPC)中
 - 关中断并将控制转移到以监控模式运行的指定的中断处理程序



Interrupts: 改变正常的控制流



需要由另一个（系统）程序处理的外部或内部事件。从程序的角度来看，事件通常是意外的或罕见的。



Interrupt Handler

- **允许嵌套中断时，在开中断之前需要保存EPC**



- 需要执行指令将EPC 存入 GPRs
 - 至少在保存EPC之前，需要一种方法来暂时屏蔽进一步的_的中断
- **需要读取记录中断原因的状态寄存器**
- **使用专门的间接跳转指令ERET (return-from-environment) 返回**
 - 开中断
 - 将处理器恢复到用户模式
 - 恢复硬件状态和控制状态

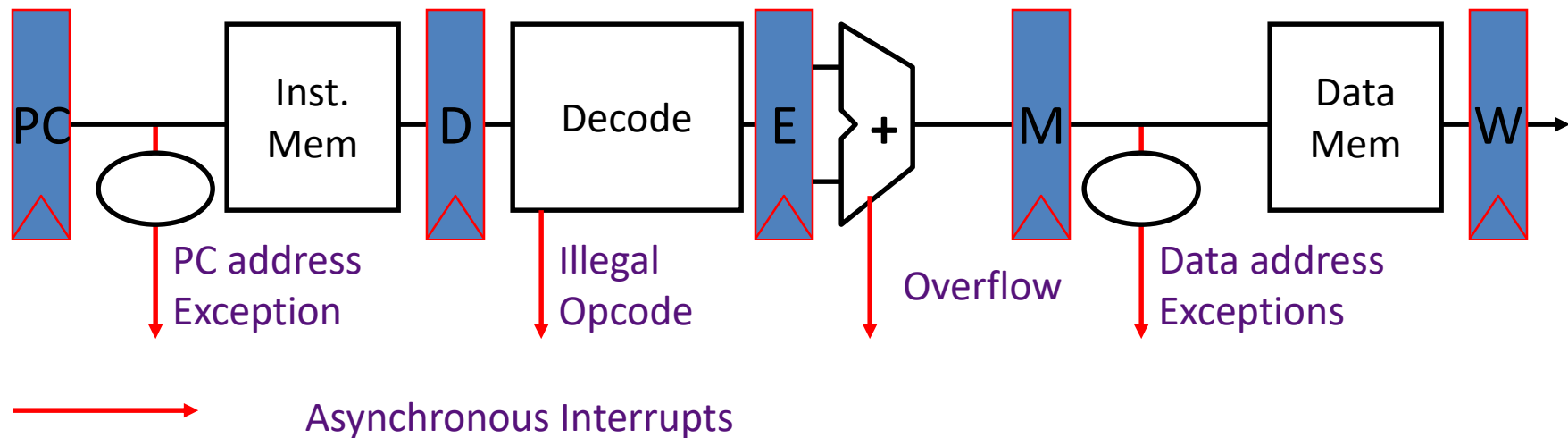


Synchronous Trap

- **同步陷阱是由特定指令执行的异常引起的**
- **通常，该指令无法完成，需要在处理异常之后重新启动**
 - 需要撤消一个或多个部分执行的指令的效果
- **在系统调用(陷阱)的情况下，该指令被认为已经完成**
 - 一种特殊的跳转指令，涉及到切换到特权模式的操作



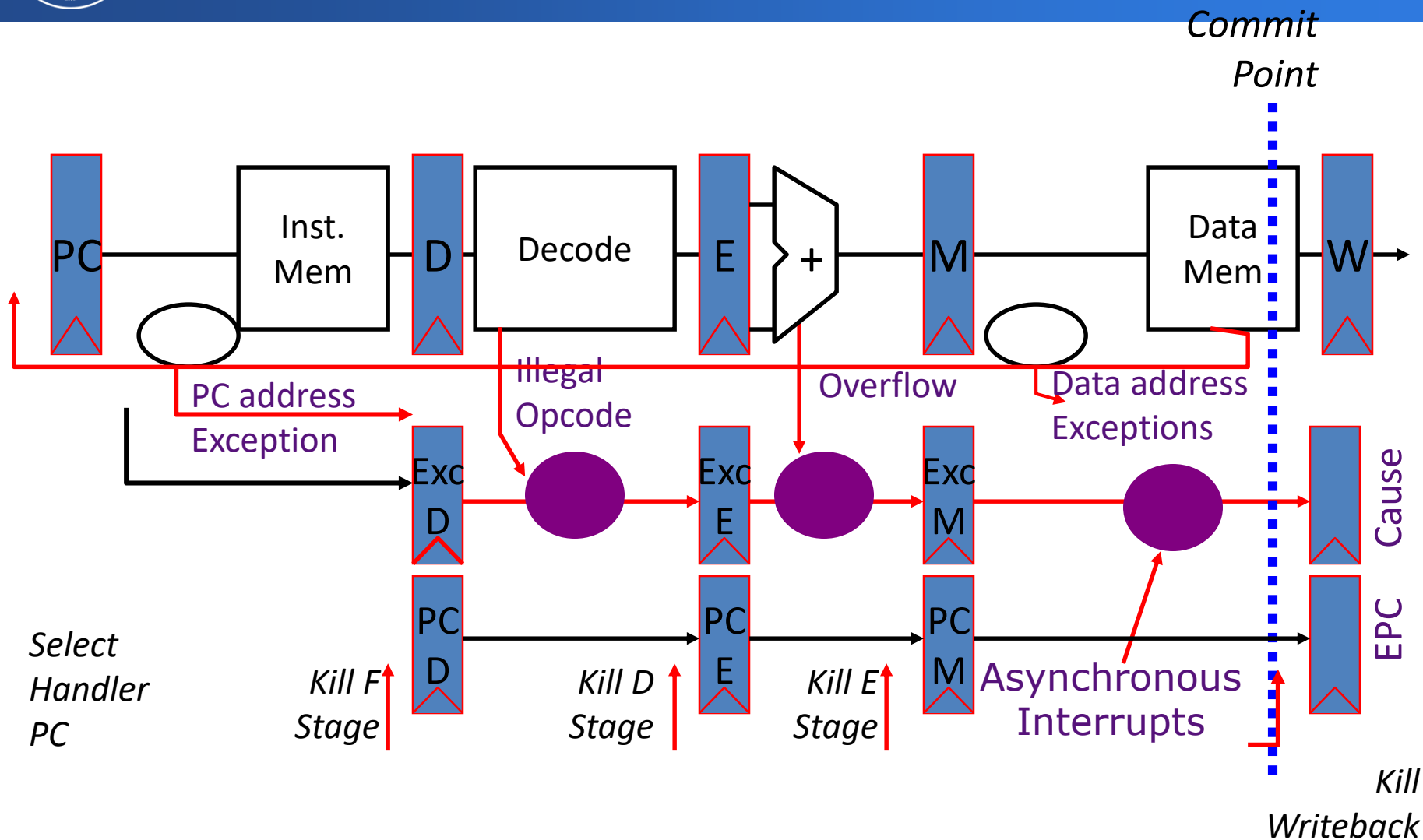
Exception Handling 5-Stage Pipeline



- 如何处理不同流水段的多个并发异常?
- 如何以及在哪里处理外部异步中断?



Exception Handling 5-Stage Pipeline





Exception Handling 5-Stage Pipeline

- 在流水线中将异常标志保留到提交阶段
- 针对某一给定的指令，早期流水阶段中的异常覆盖该指令的后续异常
- **在提交阶段并入异步中断请求（覆盖其他中断）**
- **如果提交时检测到异常：**
 - 更新异常原因及EPC寄存器
 - 终止所有流水段，
 - 将异常处理程序的地址送入PC寄存器，以跳转到处理程序中执行



异常的推测

- **预测机制**
 - 异常总是比较少的，所以简单地预测为没有异常通常是大概率事件
- **检查预测机制**
 - 在指令执行的最后阶段进行异常检测
 - 采用专门的硬件用于检测各种类型的异常
- **恢复执行机制**
 - 仅在提交阶段改变机器状态，因此可以在发生异常后丢弃部分执行的指令
 - 刷新流水线后启动异常处理程序
- **定向路径机制允许后续的指令使用没有提交的指令结果**



流水线的性能分析

- 基本度量参数：**吞吐率**，**加速比**，**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

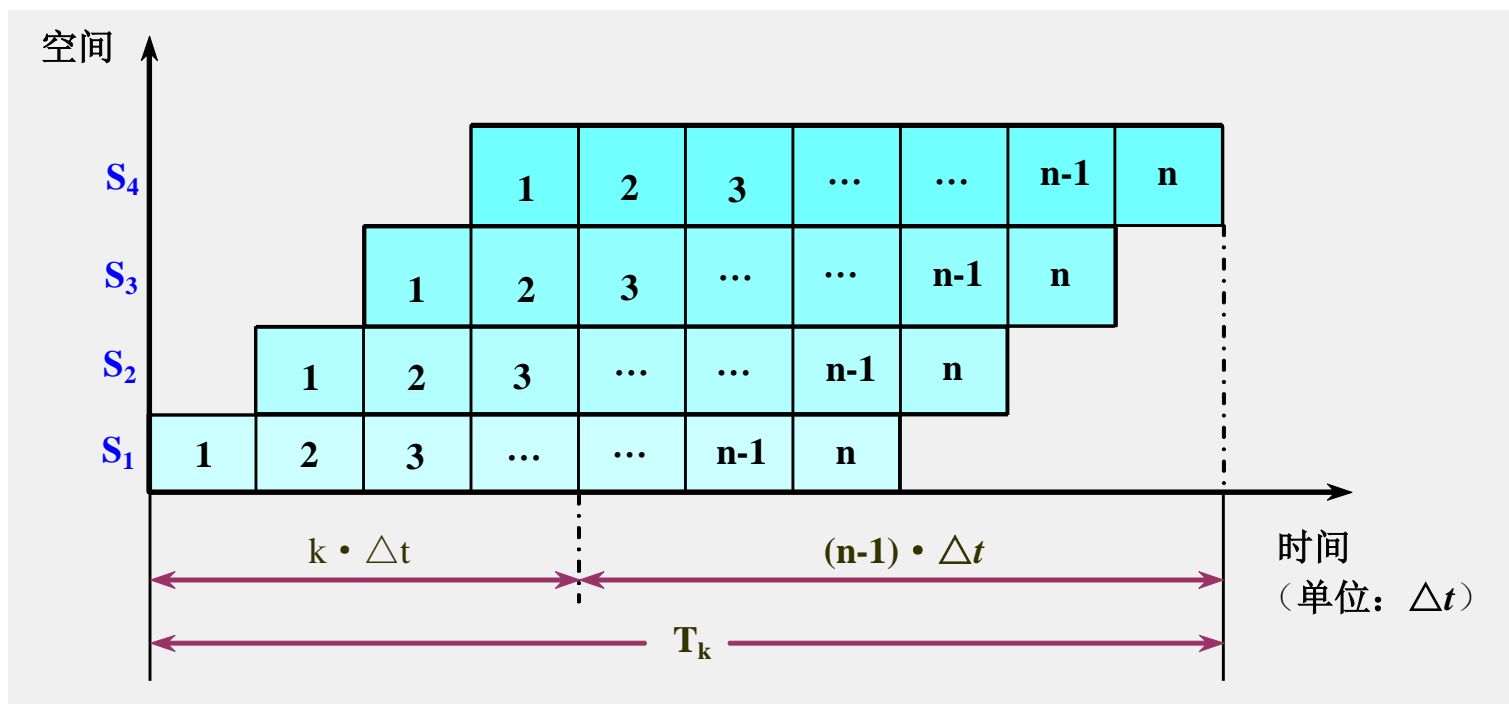
T_k : 处理完成 n 个任务所用的时间



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）

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

- 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。

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

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			

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



TP与Tpmax的关系

– 最大吞吐率与实际吞吐率的关系

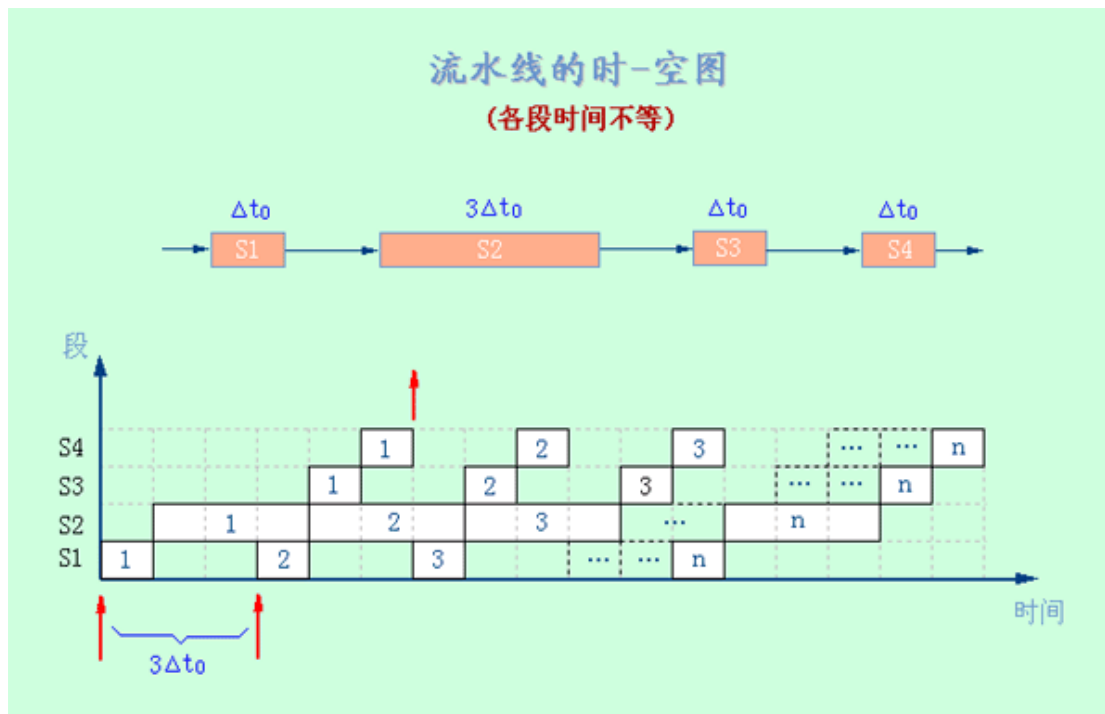
$$TP = \frac{n}{k + n - 1} TP_{\max}$$

- 流水线的实际吞吐率小于最大吞吐率，它除了与每个段的时间有关外，还与流水线的段数 k 以及输入到流水线中的任务数 n 有关。
- 只有当 $n \gg k$ 时，才有 $TP \approx TP_{\max}$ 。



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: $3\Delta t$ (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：
(Δt_i 为第 i 段的时间，共有 k 个段)

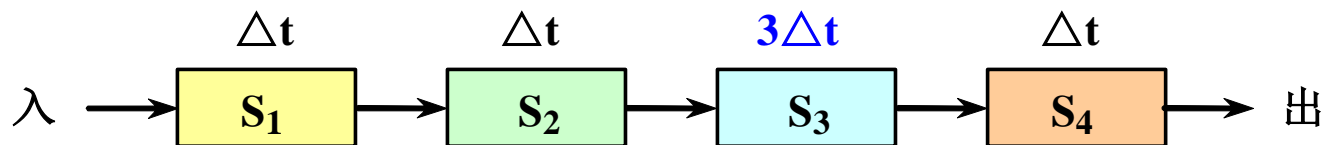
$$TP = \frac{n}{\sum_{i=1}^k \Delta t_i + (n-1) \max(\Delta t_1, \Delta t_2, \dots, \Delta t_k)}$$

- 流水线的最大吞吐率为

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



- 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 $3\Delta t$ 。



最大吞吐率为

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

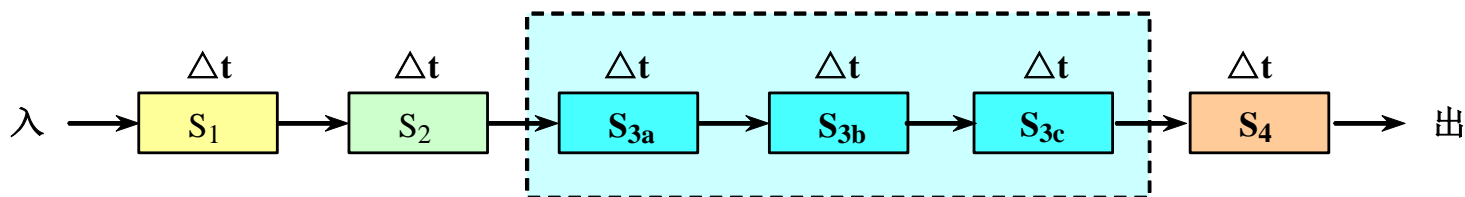


3. 解决流水线瓶颈问题的常用方法

— 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}



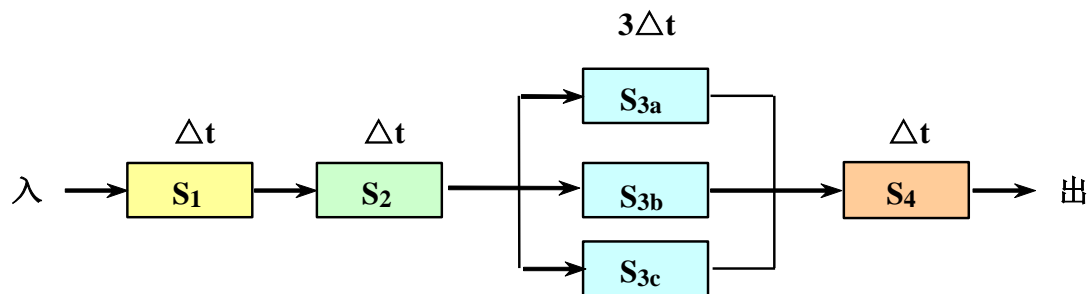
改进后的流水线的吞吐率：

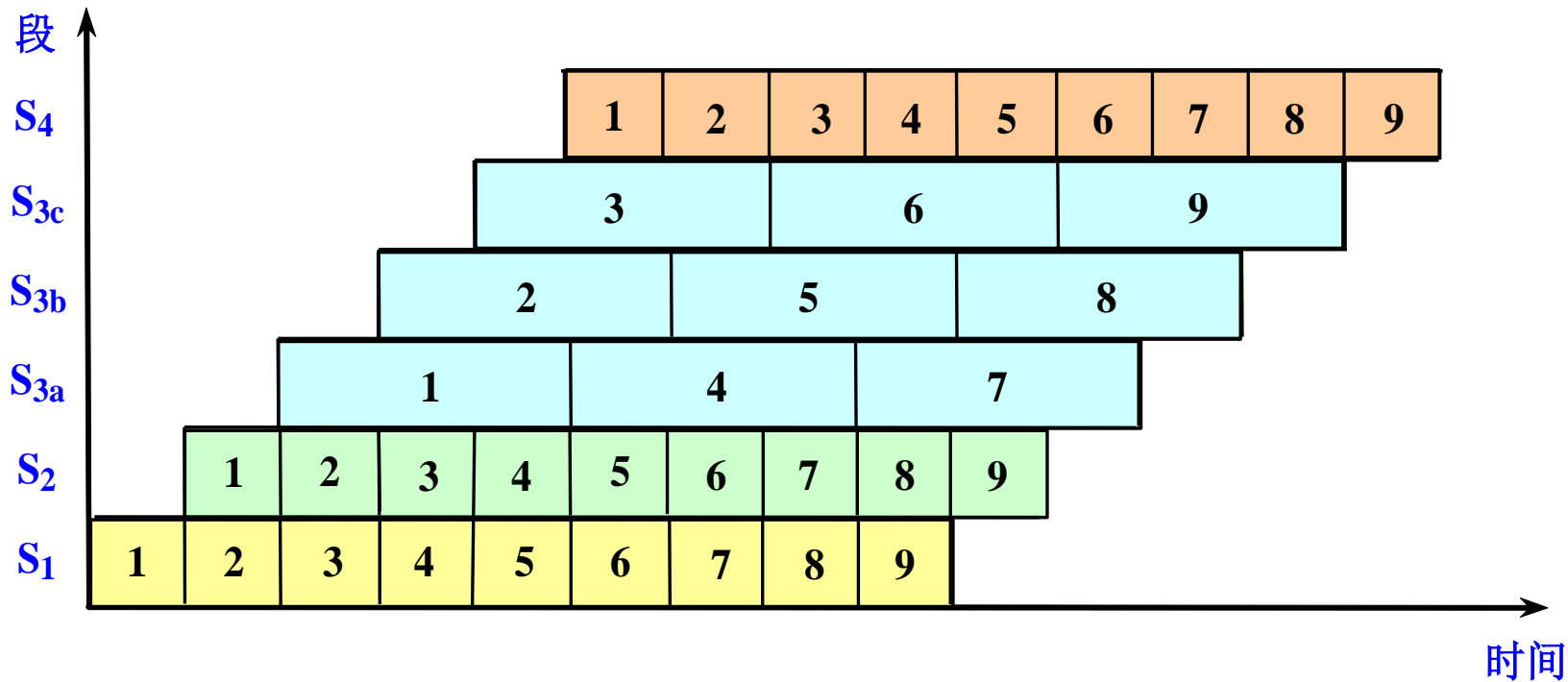
$$TP_{\max} = \frac{1}{\Delta t}$$



— 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）所用的时间为 T_s ，使用流水线后所用的时间为 T_k ，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

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

- 当 $n \gg k$ 时, $S \approx k$

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

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



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 utilization。**
 - 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 e_i 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

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

当 $n \gg k$ 时, $E \approx 1$ 。



- 当流水线各段时间相等时，流水线的效率与吞吐率成正比。

$$E = TP \Delta t$$

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

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

- 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。

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

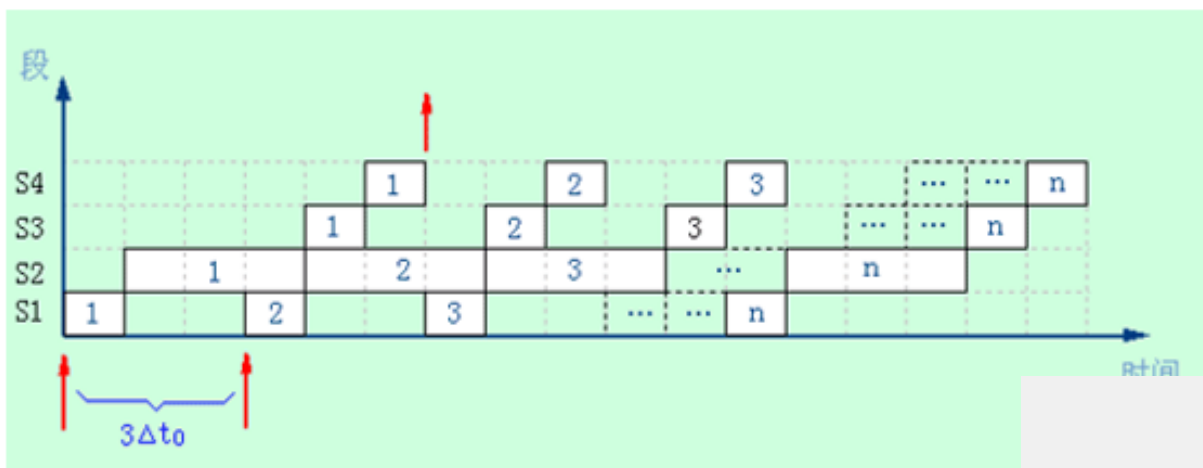
$$E = \frac{S}{k}$$

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



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



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。**



-Review: Pipelining

- **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
- **如何有利于流水线技术的应用**
 - 所有的指令都等长
 - 只有很少的指令格式
 - 只用Load/Store来进行存储器访问



流水线的加速比计算

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

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

For simple RISC pipeline, $CPI = 1$:

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



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 $CPI \geq 1.3$



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 = $(1 + 0.4 * 1) * \text{clock cycle time} / 1.05$
= $1.3 * \text{clock cycle time}$



解决控制相关的方法

- **#1: Stall 直到分支方向确定**
- **#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**



评估减少分支策略的效果

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

<i>Scheduling scheme</i>	<i>Branch penalty</i>	<i>CPI</i>	<i>speedup v. unpipelined</i>	<i>speedup v. stall</i>
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



多周期操作的处理

对整型pipeline增加浮点数处理 RISC-V



多周期操作的处理

- **问题**

- 浮点操作在1 ~ 2个cycles完成是不现实的，一般要花费较长时间
- 在MIPS中如何处理

- **在1到2个cycles时间内完成的处理方法**

- 采用较慢的时钟源，或
- 在FP部件中延迟其EX段

- **现假设FP指令与整数指令采用相同的流水线，那么**

- EX 段需要循环多次来完成FP操作，循环次数取决于操作类型
- 有多个FP功能部件，如果发射出的指令导致结构或数据相关，需暂停



Acknowledgements

- **These slides contain material developed and copyright by:**
 - John Kubiawicz (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**