

计算机体系结构

周学海

xhzhou@ustc.edu.cn

0551-63601556, 63492271

中国科学技术大学

第三章 流水线技术

3.1 流水线的基本概念

3.2 DLX(MIPS) 基本流水线

3.3 流水线的相关

3.4 异常处理

3.5 DLX (MIPS)中多周期操作的处理

3.6 MIPS R4000流水线

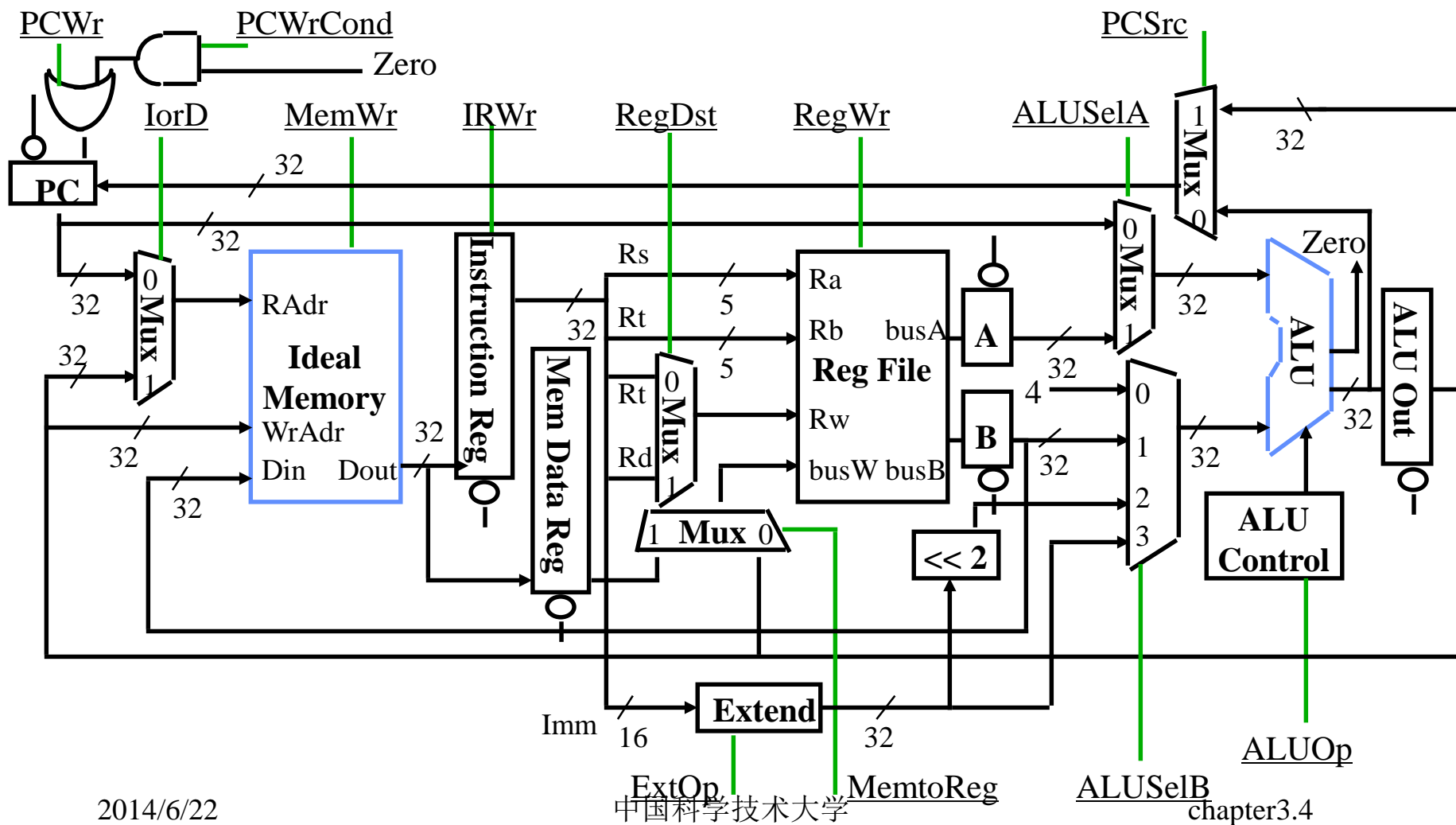
Review:性能评测

- 平均CPI?
 - 每类指令的使用频度

Type	CPI_i for type	Frequency	$CPI_i \times freq_i$
Arith/Logic	4	40%	1.6
Load	5	30%	1.5
Store	4	10%	0.4
branch	3	20%	0.6
Average CPI:4.1			

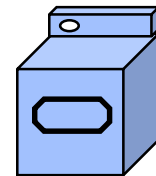
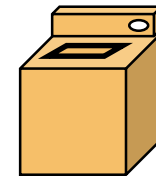
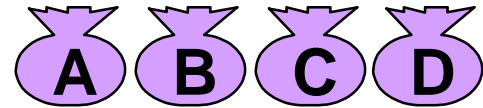
是否可以使 $CPI < 4.1$?

- 在一条指令执行过程中下图有许多空闲部件
 - 可以让指令重叠执行??

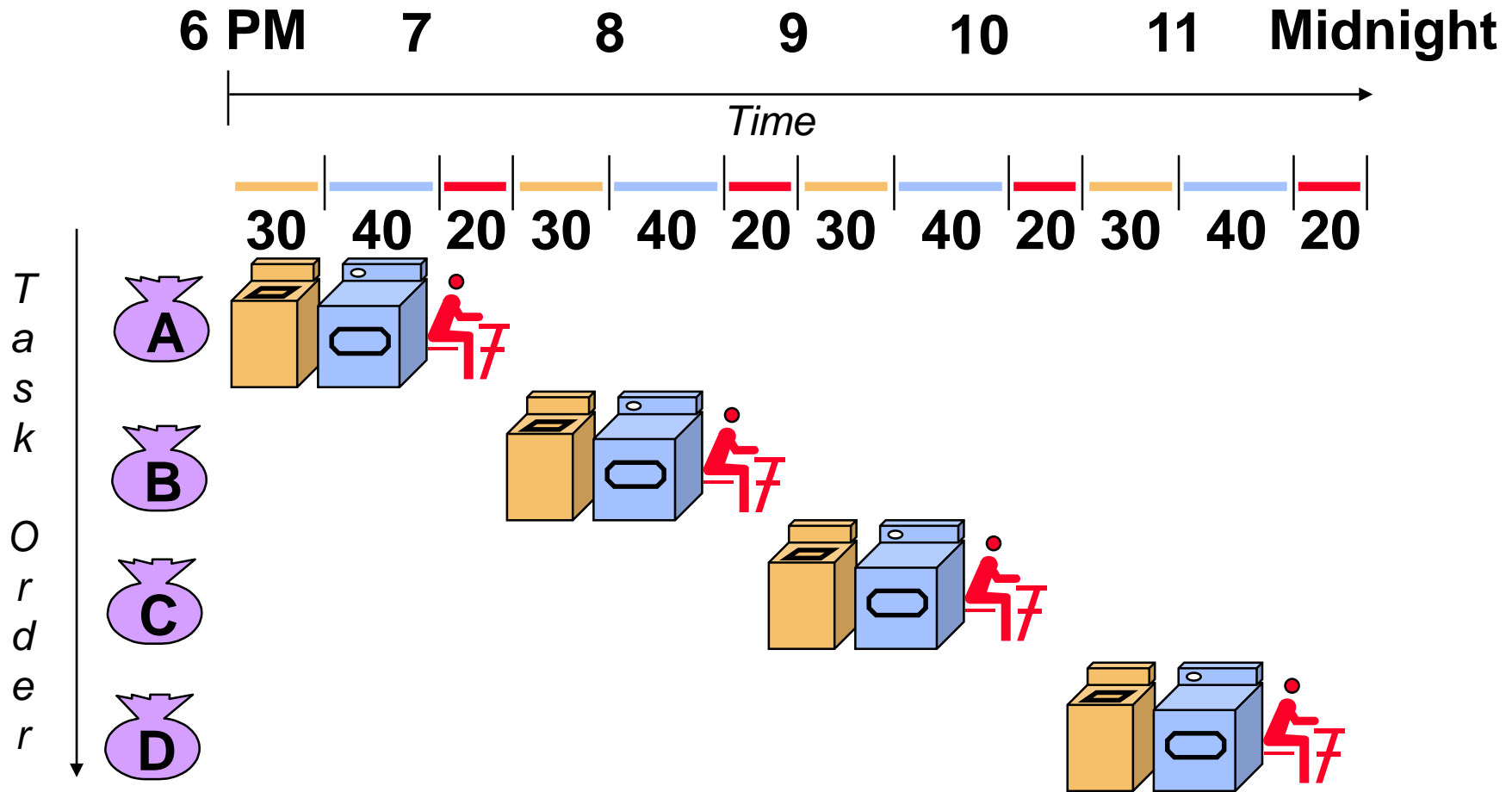


3.1 流水线的基本概念

- 洗衣为例
- **Ann, Brian, Cathy, Dave**
每人进行洗衣的动作:
wash, dry, and fold
- **washer**需要 30 minutes
- **Dryer** 需要 40 minutes
- **“Folder”** 需要 20 minutes

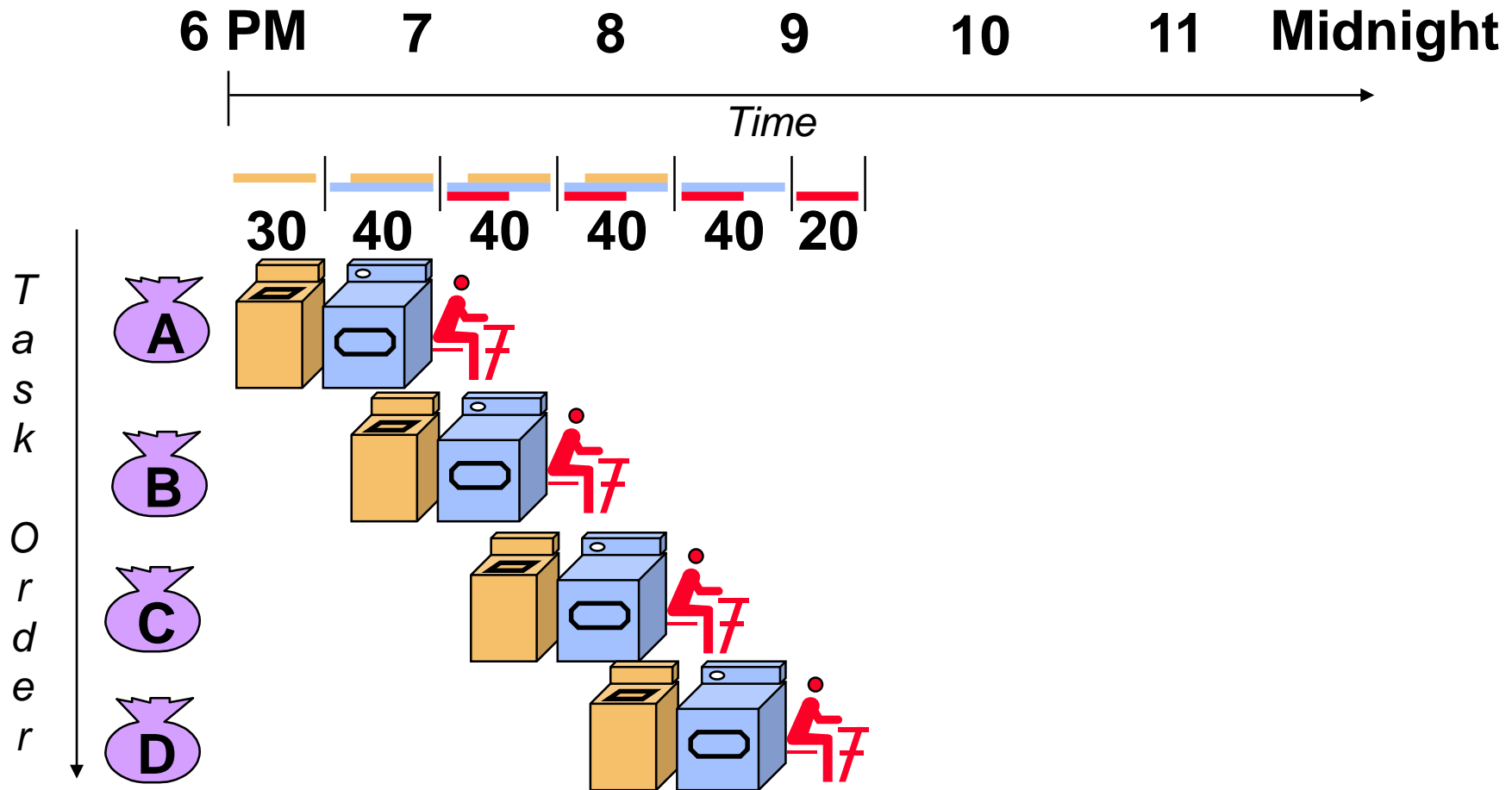


Sequential Laundry



- 顺序完成这些任务需要 6 小时
- 如果采用流水作业, 需要多长时间?

流水线作业: 尽可能让任务重叠进行



- 流水作业完成四人的洗衣任务只需要 **3.5 hours**

流水线技术要点

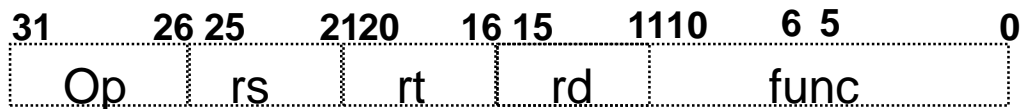
- 流水线技术并不能提高单个任务的执行效率，它可以提高整个系统的吞吐率
- 流水线中的瓶颈——最慢的那一段
- 多个任务同时执行，但使用不同的资源
- 其潜在的加速比=流水线的级数
- 流水段所需时间不均衡将降低加速比
- 流水线存在装入时间和排空时间，使得加速比降低
- 由于存在相关问题，会导致流水线停顿

3.2 DLX (MIPS)的基本流水线

- 指令流水线：CPU执行大量的指令，指令吞吐率非常重要
- DLX 的指令格式

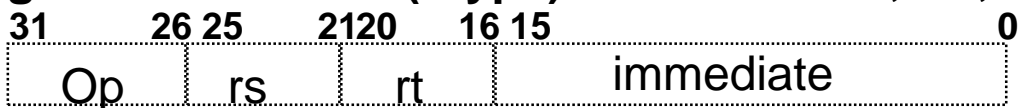
Register-Register (R-type)

ADD R1, R2, R3



Register-Immediate (I-type)

SUB R1, R2, #3



Jump / Call (J-type)

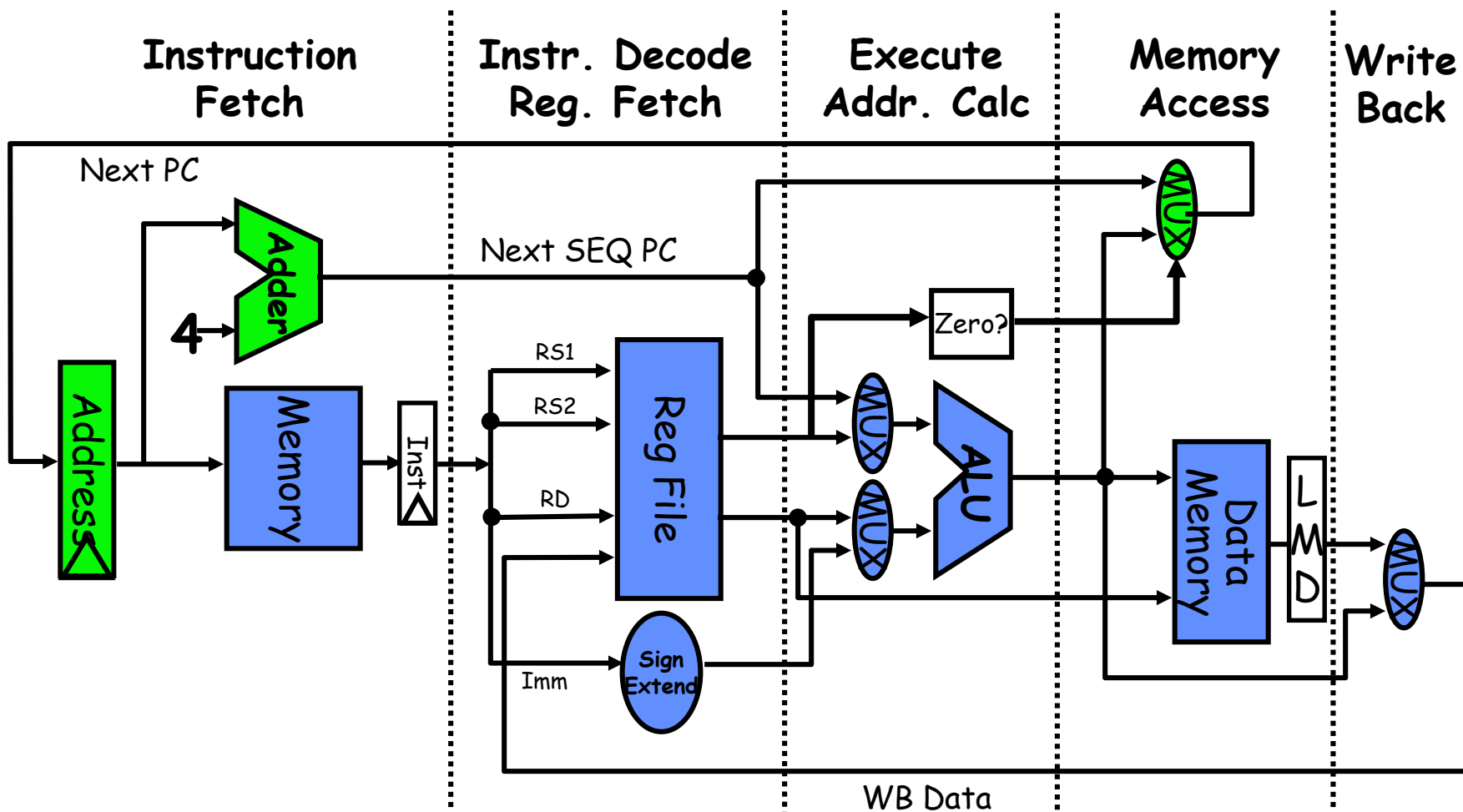
JUMP end



(jump, jump and link, trap and return from exception)

- 所有指令相同长度
- 在指令格式中寄存器位于同一位置
- 只有Loads和Stores可以对存储器操作

DLX(MIPS)数据通路一种简单实现



基本操作(Step 1 & 2)

- **Step 1 - IF**

- $IR \leftarrow Mem[PC]$ ----- fetch the next instruction from memory
- $NPC \leftarrow PC + 4$ ----- compute the new PC

- **Step 2 - ID - instruction decode and register fetch step**

- $A \leftarrow Regs[IR_{6..10}]$
- $B \leftarrow Regs[IR_{11..16}]$

» 可能读取的寄存器值没有用，但没有关系，译码后如果无用，以后操作就不用

- $Imm \leftarrow ((IR_{16})^{16} \# \# IR_{16-31})$

基本操作—Step 3, 执行阶段

根据译码的结果，有四种情况

- Memory Reference
 - $ALUOutput \leftarrow A + (IR_{16})^{16} \text{ ## } IR_{16..31}$ ----- effective address
 - $SMD \leftarrow B$ ----- data to be written if it is a STORE -- SMD (store mem data) = MDR
- Register - Register ALU instruction
 - $ALUOutput \leftarrow A \text{ op } B$
- Register - Immediate ALU instruction
 - $ALUOutput \leftarrow A \text{ op } ((IR_{16})^{16} \text{ ## } IR_{16..31})$
- Branch/Jump
 - $ALUOutput \leftarrow NPC + (IR_{16})^{16} \text{ ## } IR_{16..31}$
 - $cond \leftarrow A \text{ op } 0$ --- for conditional branches A 's value is the condition base (= for BEQZ)
 - 在简单的 **Load-Store** 机器中，不存在即需要计算存储器地址，指令地址，又要进行**ALU**运算的指令，因此可以将计算有效地址与执行合二为一，在一个流水段中进行。

Step 4 & Step5

Step 4 MEM - memory access/branch completion

- **memory reference**

- $LMD \leftarrow Mem[ALUOutput]$ ----- if it's a load; LMD (load memory data) = MDR 或
- $Mem[ALUOutput] \leftarrow SMD$

- **branch**

- if (cond) then $PC \leftarrow ALUOutput$ else $PC \leftarrow NPC$
- for Jumps the condition is always true

Step 5 WB - write back

- **Reg - Reg ALU**

- $Regs[IR_{16..20}] \leftarrow ALUOutput$

- **Reg - Immed ALU**

- $Regs[IR_{11..15}] \leftarrow ALUOutput$

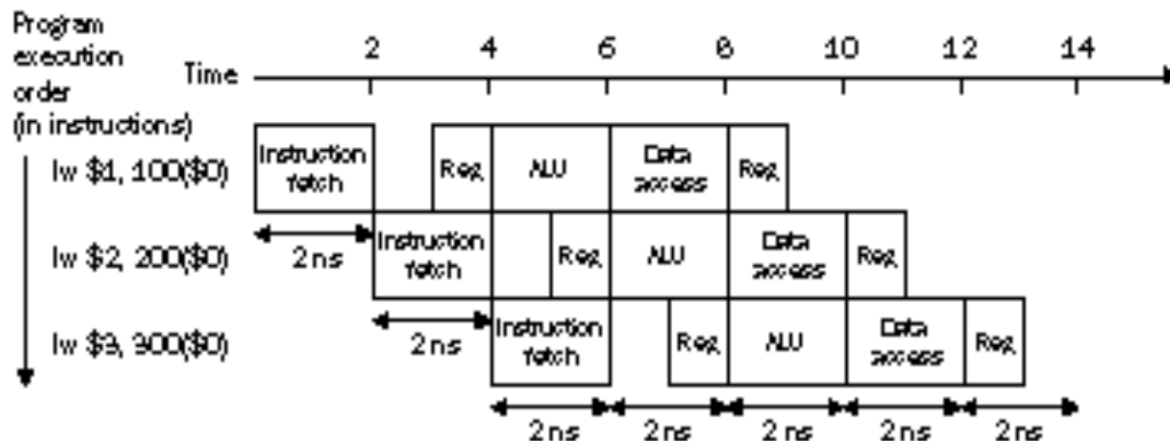
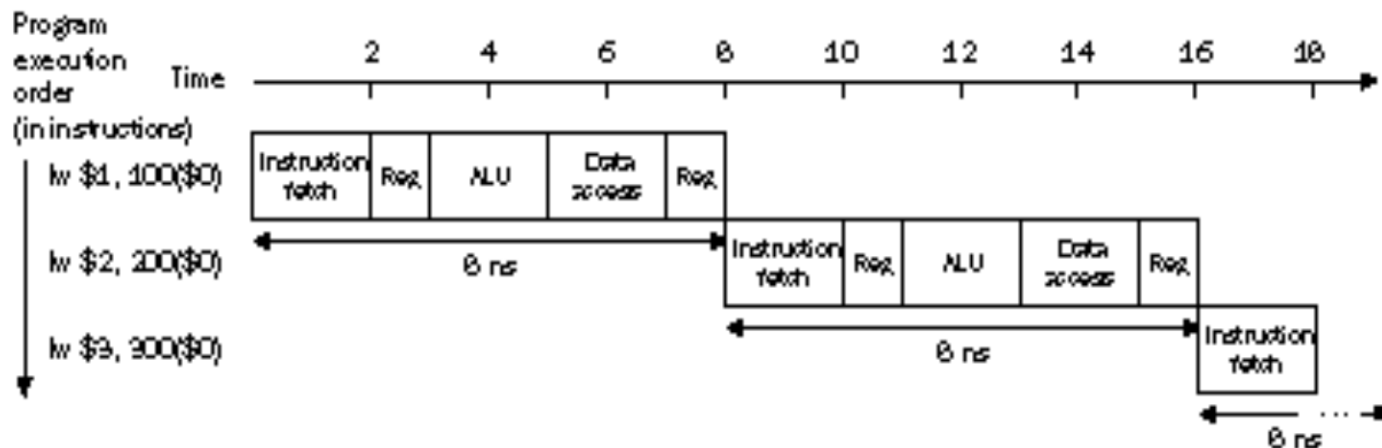
- **Load**

- $Regs[IR_{11..15}] \leftarrow LMD$

这种结构是否可行

- 模型是正确的，但没有优化
- 还有其他选择
 - 指令和数据存储器是否可以分开
 - 采用一个长周期还是5个短周期实现

单周期和多周期控制



- 多周期控制可实现指令重叠执行

DLX(MIPS)的基本流水线

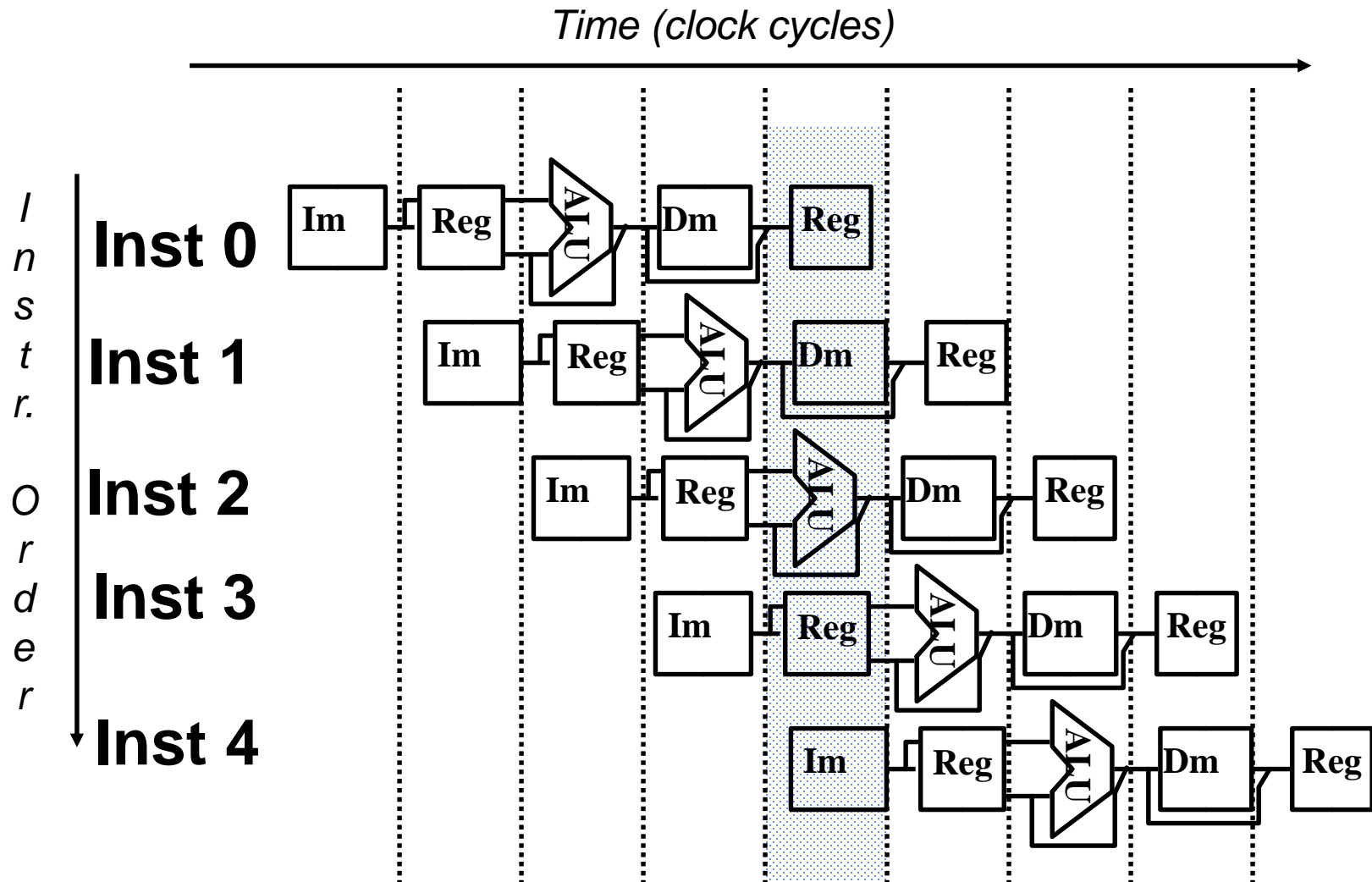
Instruction					Clock	Number			
Number	1	2	3	4	5	6	7	8	9
i	IF	ID	EX	MEM	WB				
i+1		IF	ID	EX	MEM	WB			
i+2			IF	ID	EX	MEM	WB		
i+3				IF	ID	EX	MEM	WB	
i+4					IF	ID	EX	MEM	WB

- 假设流水线周期为每步所花费的时间

为什么用流水线?

- 假设执行**100**条指令
- 单周期机器
 - $45 \text{ ns/cycle} \times 1 \text{ CPI} \times 100 \text{ inst} = 4500 \text{ ns}$
- 多周期机器
 - $10 \text{ ns/cycle} \times 4.6 \text{ CPI (due to inst mix)} \times 100 \text{ inst} = 4600 \text{ ns}$
- 理想流水线机器
 - $10 \text{ ns/cycle} \times (1 \text{ CPI} \times 100 \text{ inst} + 4 \text{ cycle drain}) = 1040 \text{ ns}$

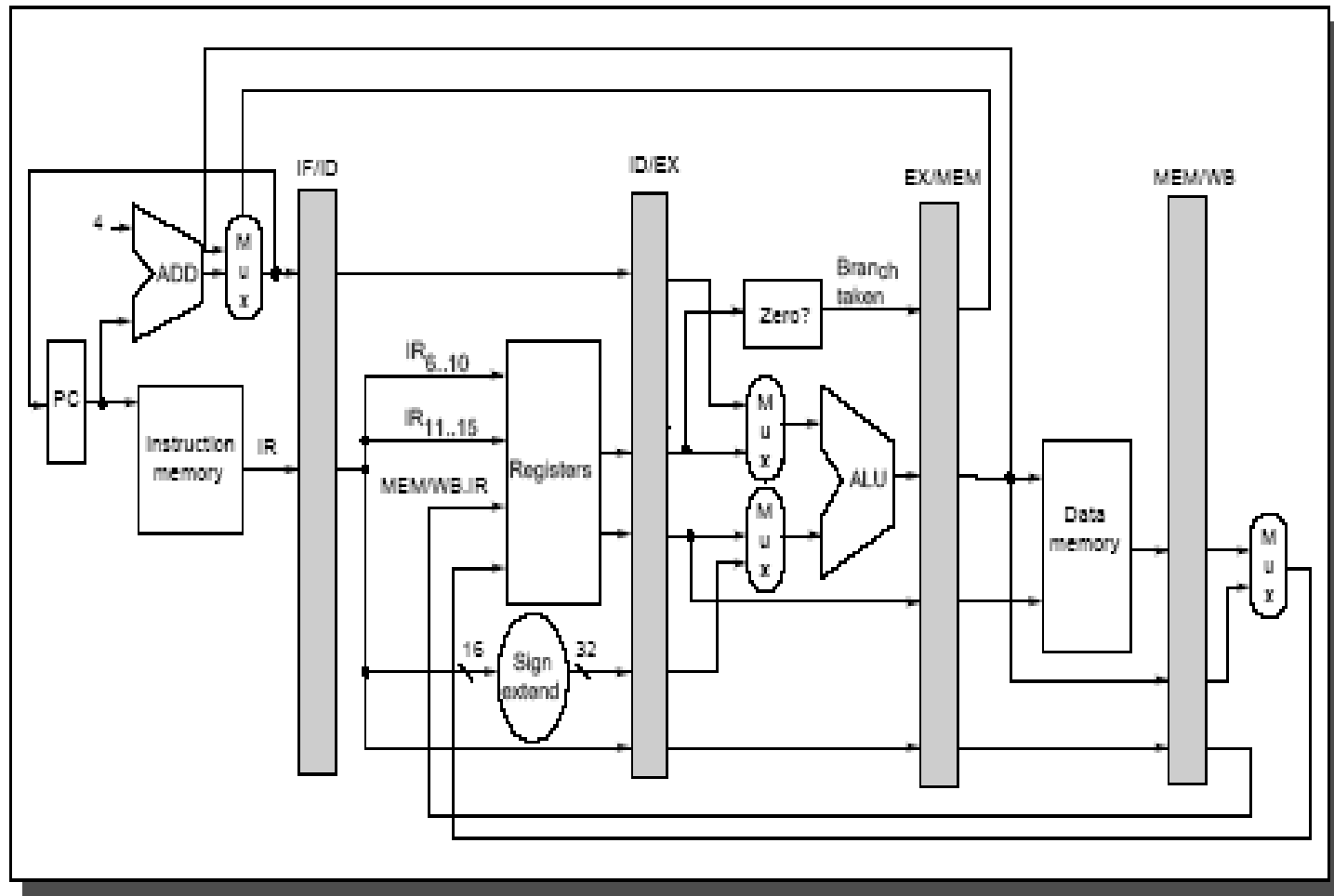
为什么用流水线(cont.)?—资源利用率高



流水线正常工作的基本条件

- 各段间需要使用寄存器文件保存当前段传送到下一段的数据和控制信息
- 存储器带宽是非流水的**5**倍

新的 DLX (MIPS)数据通路



Stage	Any instruction		
IF	$IF/ID.IR \leftarrow Mem[PC]; [rs]$ $IF/ID.NPC, PC \leftarrow (if ((EX/MEM.opcode == branch) \& EX/MEM.cond) \{EX/MEM.ALUOutput\} else \{PC+4\});$		
ID	$ID/EX.A \leftarrow Regs[IF/ID.IR[rs]]; ID/EX.B \leftarrow Regs[IF/ID.IR[rt]];$ $ID/EX.NPC \leftarrow IF/ID.NPC; ID/EX.IR \leftarrow IF/ID.IR;$ $ID/EX.Imm \leftarrow sign\ extend(IF/ID.IR[immediate\ field]);$		
	ALU instruction	Load or store instruction	Branch instruction
EX	$EX/MEM.IR \leftarrow ID/EX.IR;$ $EX/MEM.ALUOutput \leftarrow$ $ID/EX.A\ func\ ID/EX.B;$ or $EX/MEM.ALUOutput \leftarrow$ $ID/EX.A\ op\ ID/EX.Imm;$ $EX/MEM.cond \leftarrow 0;$	$EX/MEM.IR \leftarrow ID/EX.IR$ $EX/MEM.ALUOutput \leftarrow$ $ID/EX.A + ID/EX.Imm;$ $EX/MEM.cond \leftarrow 0;$ $EX/MEM.B \leftarrow ID/EX.B;$	$EX/MEM.ALUOutput \leftarrow$ $ID/EX.NPC + ID/EX.Imm;$ $EX/MEM.cond \leftarrow$ $(ID/EX.A == 0);$
MEM	$MEM/WB.IR \leftarrow EX/MEM.IR;$ $MEM/WB.ALUOutput \leftarrow$ $EX/MEM.ALUOutput;$	$MEM/WB.IR \leftarrow EX/MEM.IR;$ $MEM/WB.LMD \leftarrow$ $Mem[EX/MEM.ALUOutput];$ or $Mem[EX/MEM.ALUOutput] \leftarrow$ $EX/MEM.B;$	
WB	$Regs[MEM/WB.IR[rd]] \leftarrow$ $MEM/WB.ALUOutput;$ or $Regs[MEM/WB.IR[rt]] \leftarrow$ $MEM/WB.ALUOutput;$	For load only: $Regs[MEM/WB.IR[rt]] \leftarrow$ $MEM/WB.LMD;$	

Review lecture

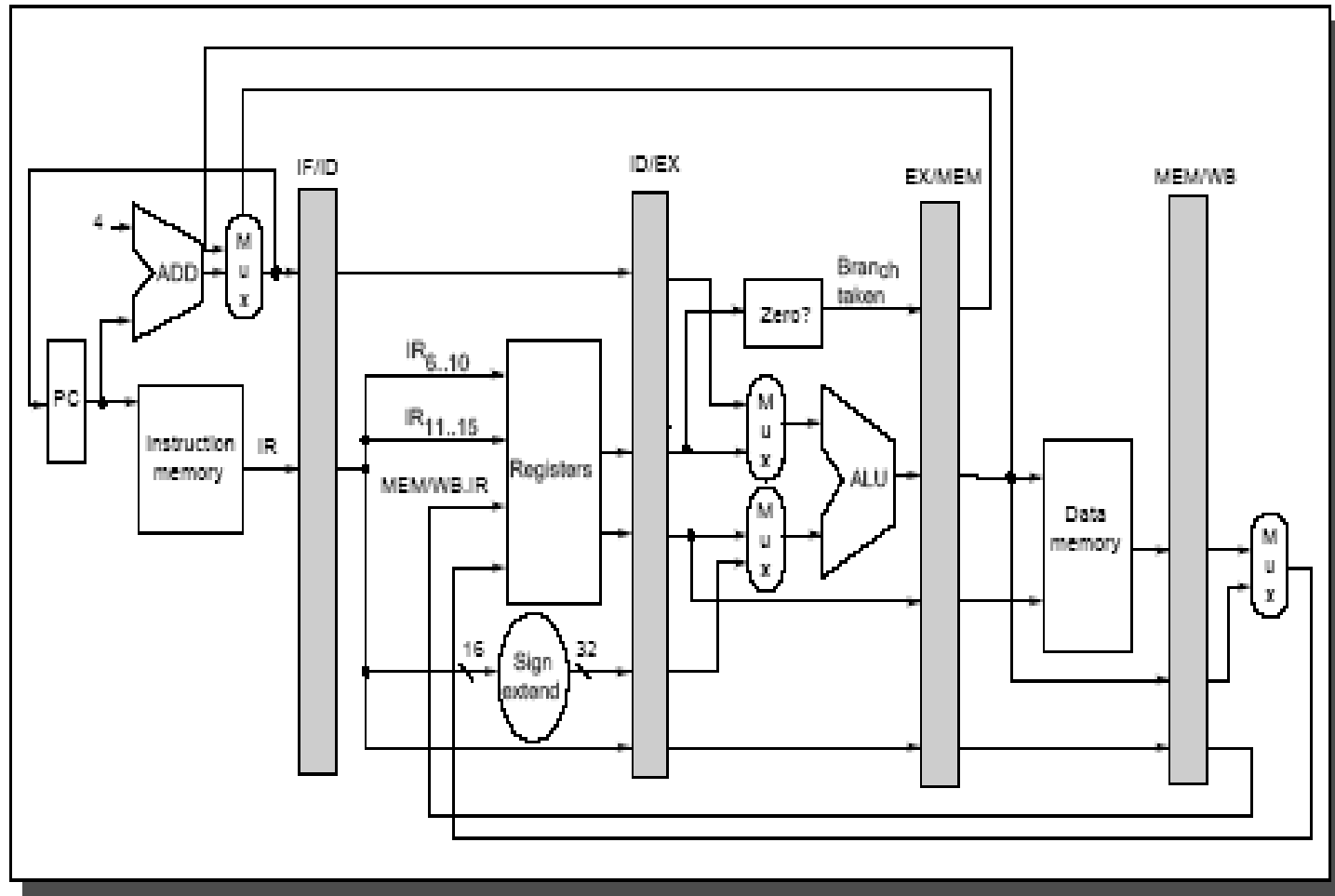
- 流水线技术要点

- 流水线技术并不能提高单个任务的执行效率，它可以提高整个系统的吞吐率
- 流水线中的瓶颈——最慢的那一段
- 多个任务同时执行，但使用不同的资源
- 其潜在的加速比=流水线的级数
- 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
- 由于存在相关问题，会导致流水线停顿

- 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
- 存储器带宽是非流水的5倍

新的DLX (MIPS)数据通路



在新的Datapath下各段的操作

- **IF**

- $IF/ID.IR \leftarrow Mem[PC];$
- $IF/ID.NPC, PC \leftarrow (if ((EX/MEM.opcode == branch) \& EX/MEM.cond) \{EX/MEM.ALUOutput\} \text{ else } \{PC+4\});$

- **ID**

- $ID/EX.A \leftarrow Regs[IF/ID.IR[rs]]; ID/EX.B \leftarrow Regs[IF/ID.IR[rt]];$
- $ID/EX.NPC \leftarrow IF/ID.NPC; ID/EX.IR \leftarrow IF/ID.IR;$
- $ID/EX.Imm \leftarrow sign\text{-}extend(IF/ID.IR[immediate\ field]);$

- **EX**

- **ALU instruction**
 - » $EX/MEM.IR \leftarrow ID/EX.IR;$
 - » $EX/MEM.ALUOutput \leftarrow ID/EX.A \text{ func } ID/EX.B; \text{ or}$
 - » $EX/MEM.ALUOutput \leftarrow ID/EX.A \text{ op } ID/EX.Imm;$

- Load or store instruction
 - » $\text{EX/MEM.IR} \leftarrow \text{ID/EX.IR}$
 - » $\text{EX/MEM.ALUOutput} \leftarrow \text{ID/EX.A} + \text{ID/EX.Imm}$
 - » $\text{EX/MEM.B} \leftarrow \text{ID/EX.B}$
- Branch instruction
 - » $\text{EX/MEM.ALUOutput} \leftarrow \text{ID/EX.NPC} + (\text{ID/EX.Imm} \ll 2)$
 - » $\text{EX/MEM.cond} \leftarrow (\text{ID/EX.A} == 0);$

• MEM

- ALU Instruction
 - » $\text{MEM/WB.IR} \leftarrow \text{EX/MEM.IR}$
 - » $\text{MEM/WB.ALUOutput} \leftarrow \text{EX/MEM.ALUOutput};$
- Load or store instruction
 - » $\text{MEM/WB.IR} \leftarrow \text{EX/MEM.IR};$
 - » $\text{MEM/WB.LMD} \leftarrow \text{Mem}[\text{EX/MEM.ALUOutput}];$ or
 $\text{Mem}[\text{EX/MEM.ALUOutput}] \leftarrow \text{EX/MEM.B};$ (**store**)

- **WB**

- **ALU instruction**

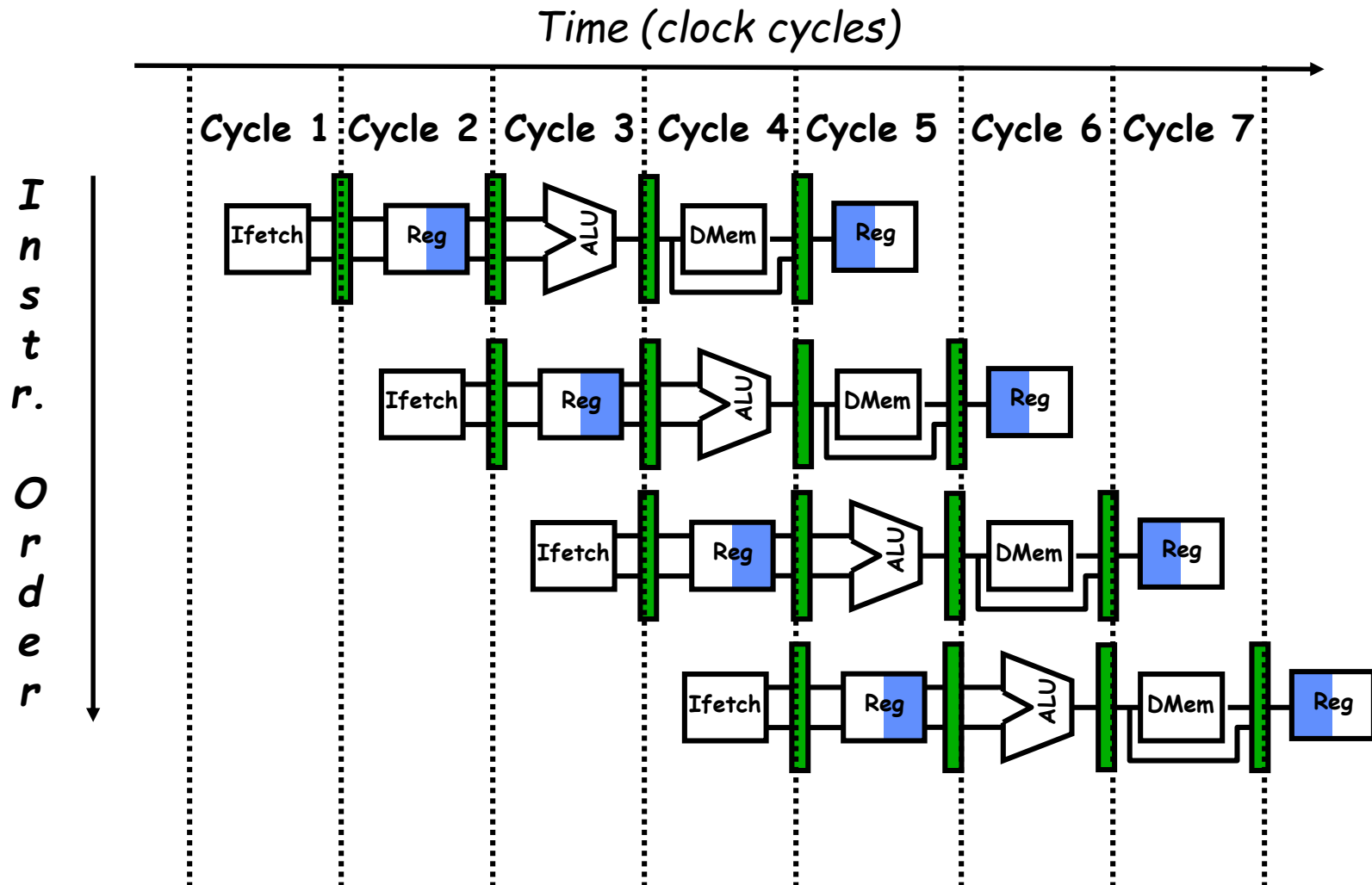
- » **Regs[MEM/WB.IR[rd]] \leftarrow MEM/WB.ALUOutput; or**

- » **Regs[MEM/WB.IR[rt]] \leftarrow MEM/WB.ALUOutput;**

- **For load only**

- » **Regs[MEM/WB.IR[rt]] \leftarrow MEM/WB.LMD**

简化的 Pipelining



流水线性能分析

- 基本度量参数：吞吐率，加速比，效率
- 吞吐率：
 - 单位时间内流水线所完成的任务数或输出结果数量
 - 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。

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			

流水线的性能指标

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。

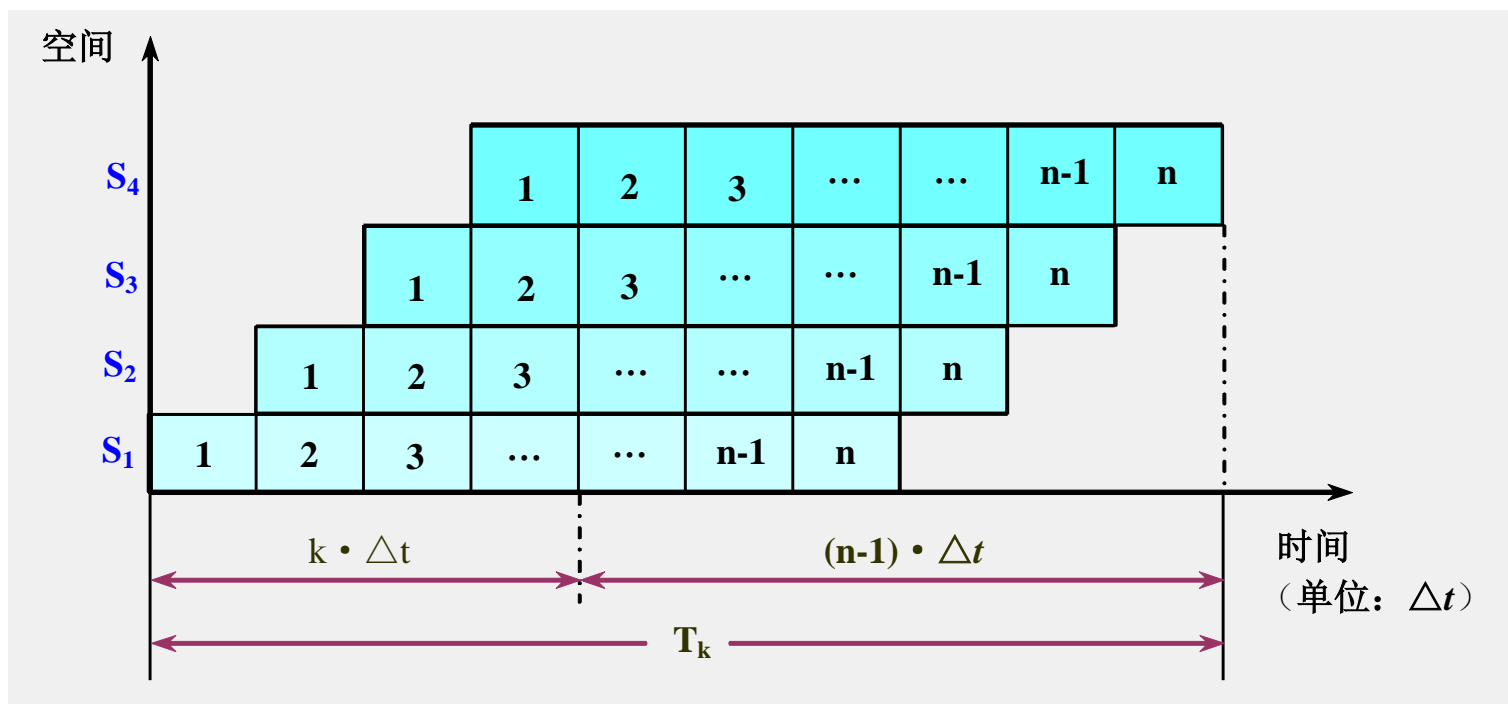
$$TP = \frac{n}{T_K}$$

n ：任务数

T_k ：处理完成 n 个任务所用的时间

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



- 流水线完成 n 个连续任务所需要的总时间为

(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

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

– 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 等有关。
- ❑ 只有当 $n \gg k$ 时，才有 $TP \approx TP_{\max}$ 。

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

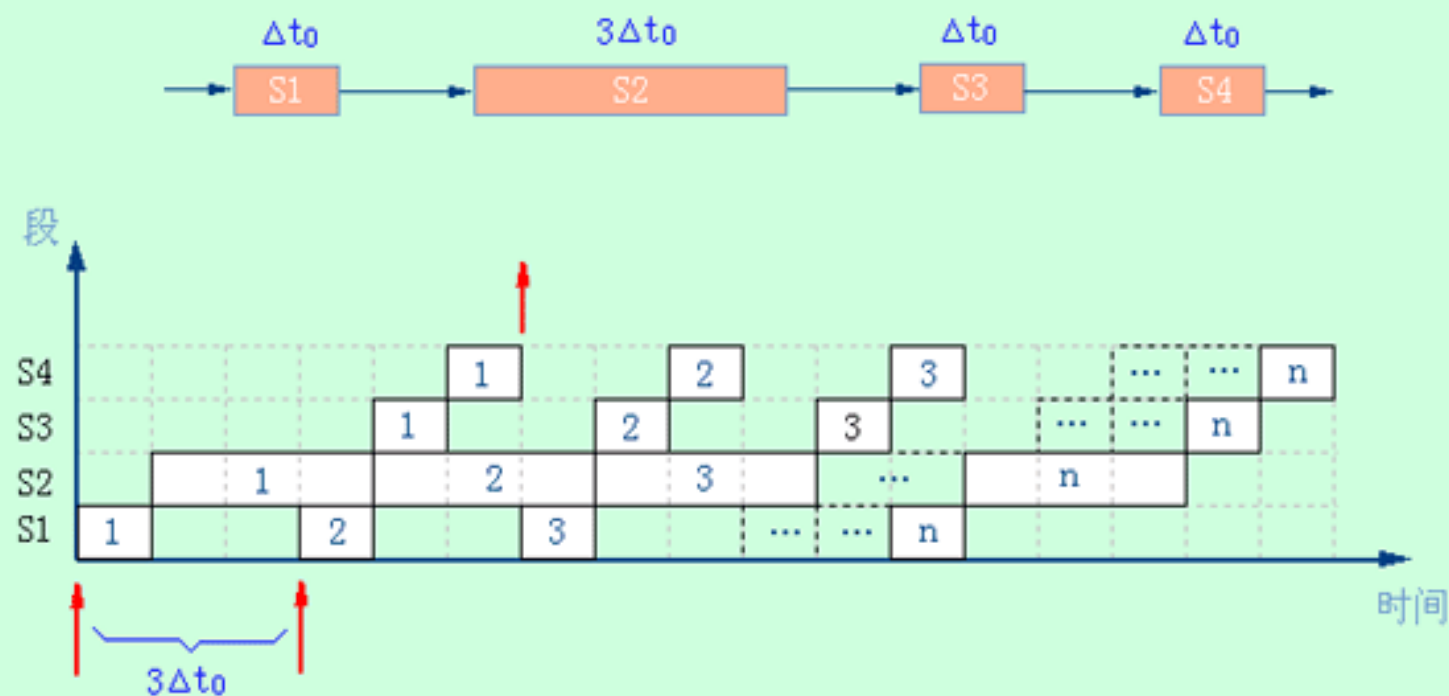
» S1, S3, S4各段的时间: Δt

» 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**。

流水线的时-空图

(各段时间不等)



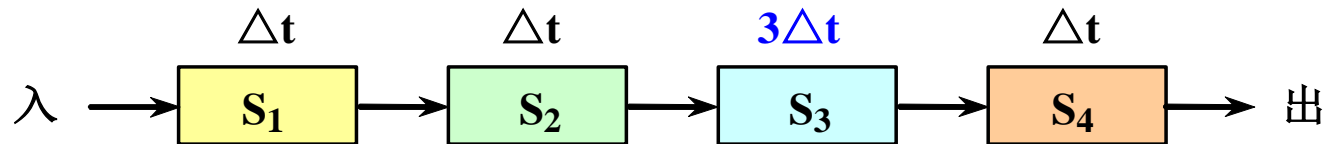
- 各段时间不等的流水线的**实际吞吐率**：
(Δt_i 为第 i 段的时间，共有 k 个段)

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

- 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， S_1 ， S_2 ， S_4 各段的时间都是 Δt ，唯有 S_3 的时间是 $3\Delta t$ 。



最大吞吐率为

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

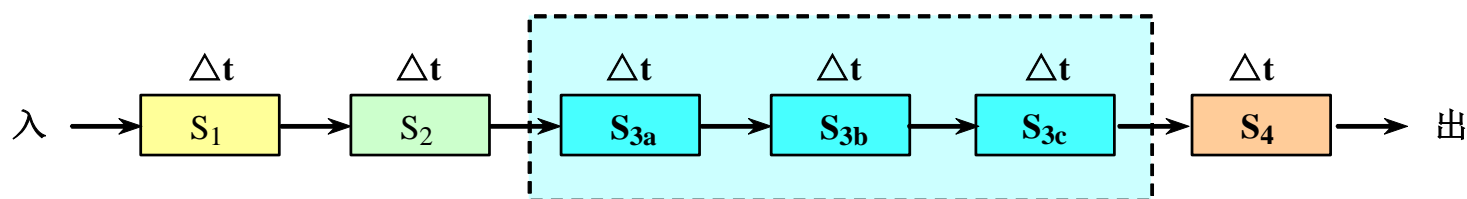
3. 解决流水线瓶颈问题的常用方法

举例

- 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：

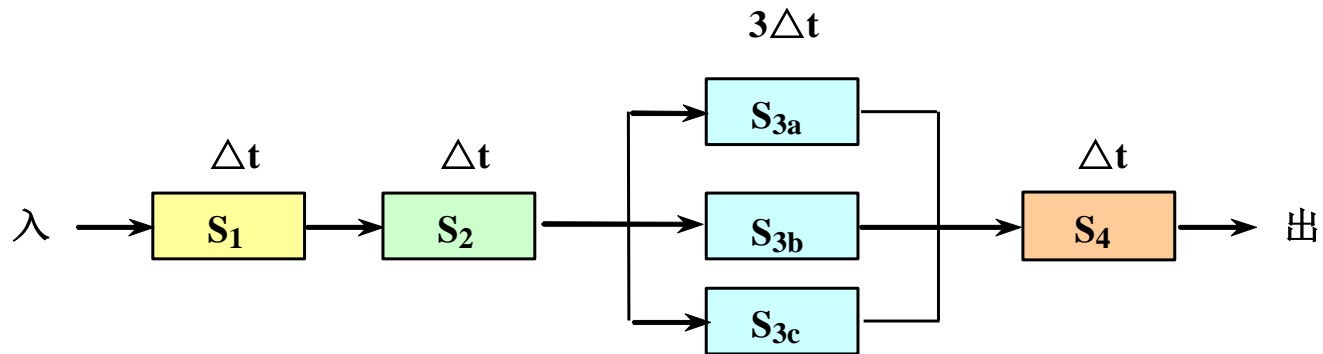
$$TP_{\max} = \frac{1}{\Delta t}$$

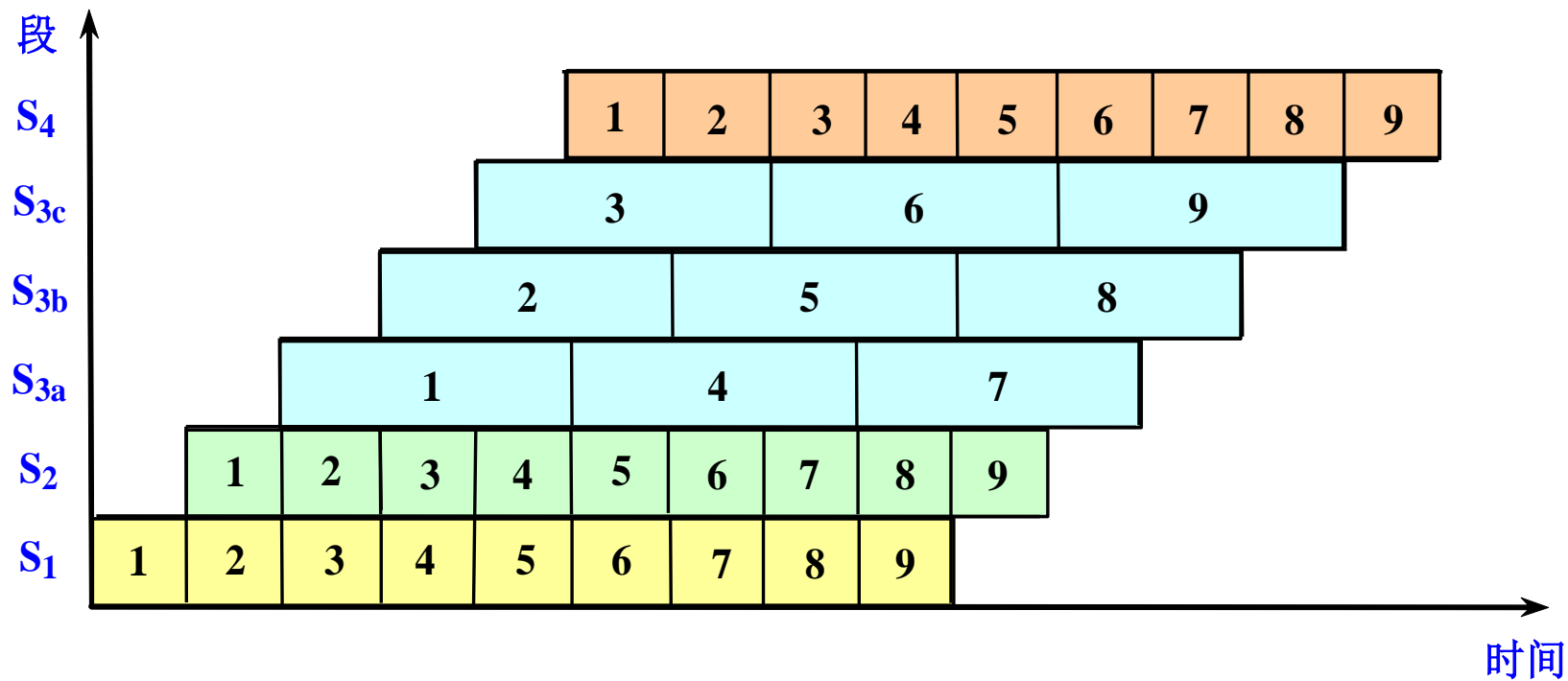
— 重复设置瓶颈段

» 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 S_3 ： S_{3a} ， S_{3b} ， S_{3c}





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 ）

- 一条 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。

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

(解释)

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

- 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。

$$E = TP \Delta t$$

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。

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

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

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

- 实际吞吐率：假设 m 段，完成 n 个任务，单位时间所实际完成的任务数。
- 加速比： m 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。

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

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

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

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

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

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

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

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

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

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

03-11-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

03-11-review: 流水线性能分析

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

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

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

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

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

吞吐率、加速比、效率
之间的关系

流水线技术应用的难度何在？
：相关问题

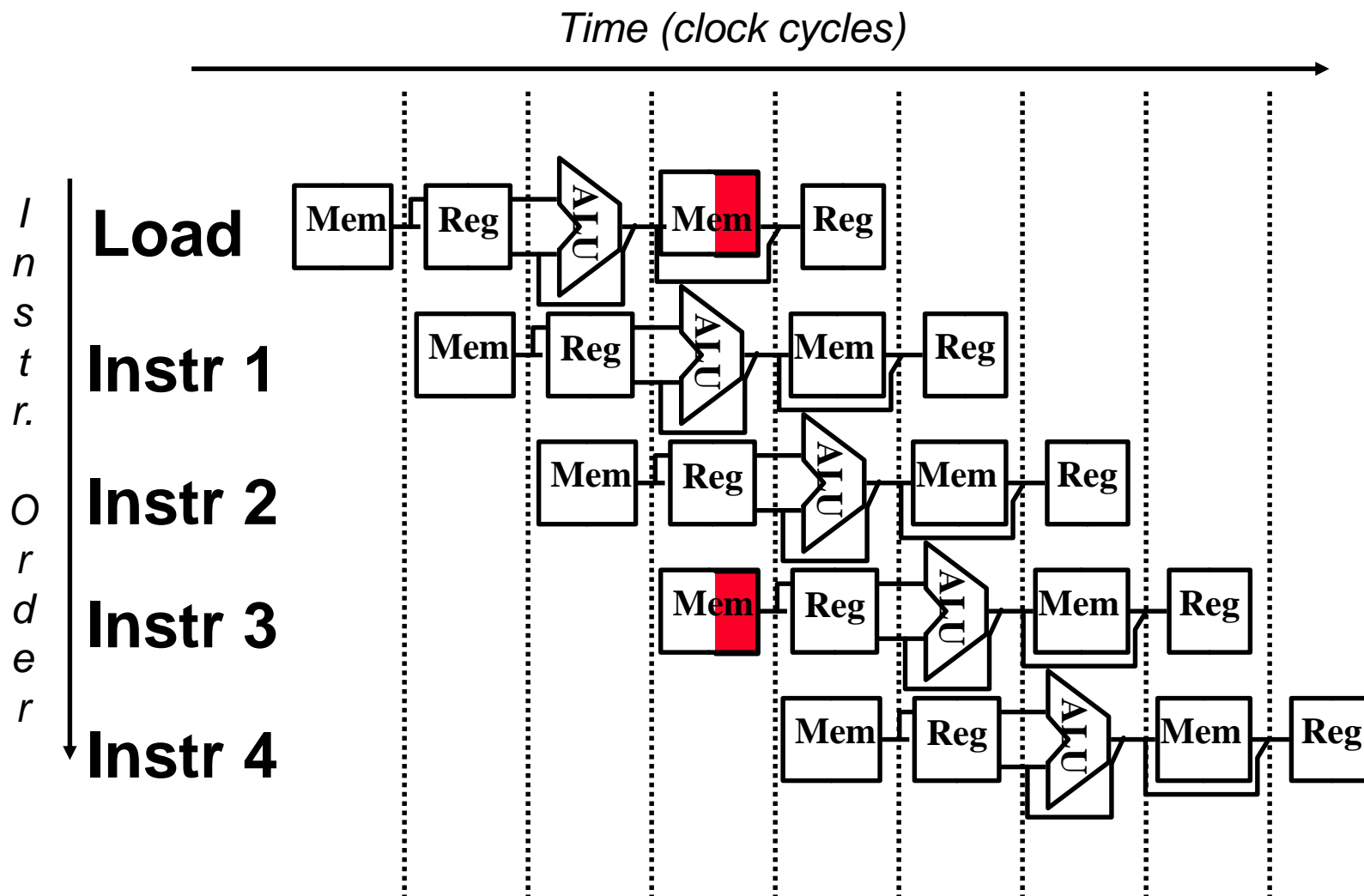
3.3 流水线的相关

- 相关的基本概念
- 结构相关
- 数据相关
- 控制相关

采用流水线技术带来的新的问题

- 流水线相关
 - 结构相关：同一时间两种方式使用同一资源
 - » 例如 **washer/dryer** 合在一起，
 - » **IM**和**ID**合在一起
 - 控制相关：试图在条件未评估之前，就做决定
 - » 例如 **branch instructions**
 - 数据相关：在数据未准备好之前，就需要使用数据
 - » 当前指令的执行需要上一条指令的结果
- 使用等待策略总是可以解决相关
 - 流水线控制必须能检测相关，否则由软件设计来避免
 - 采用相应操作解决相关 (**or** 等待)

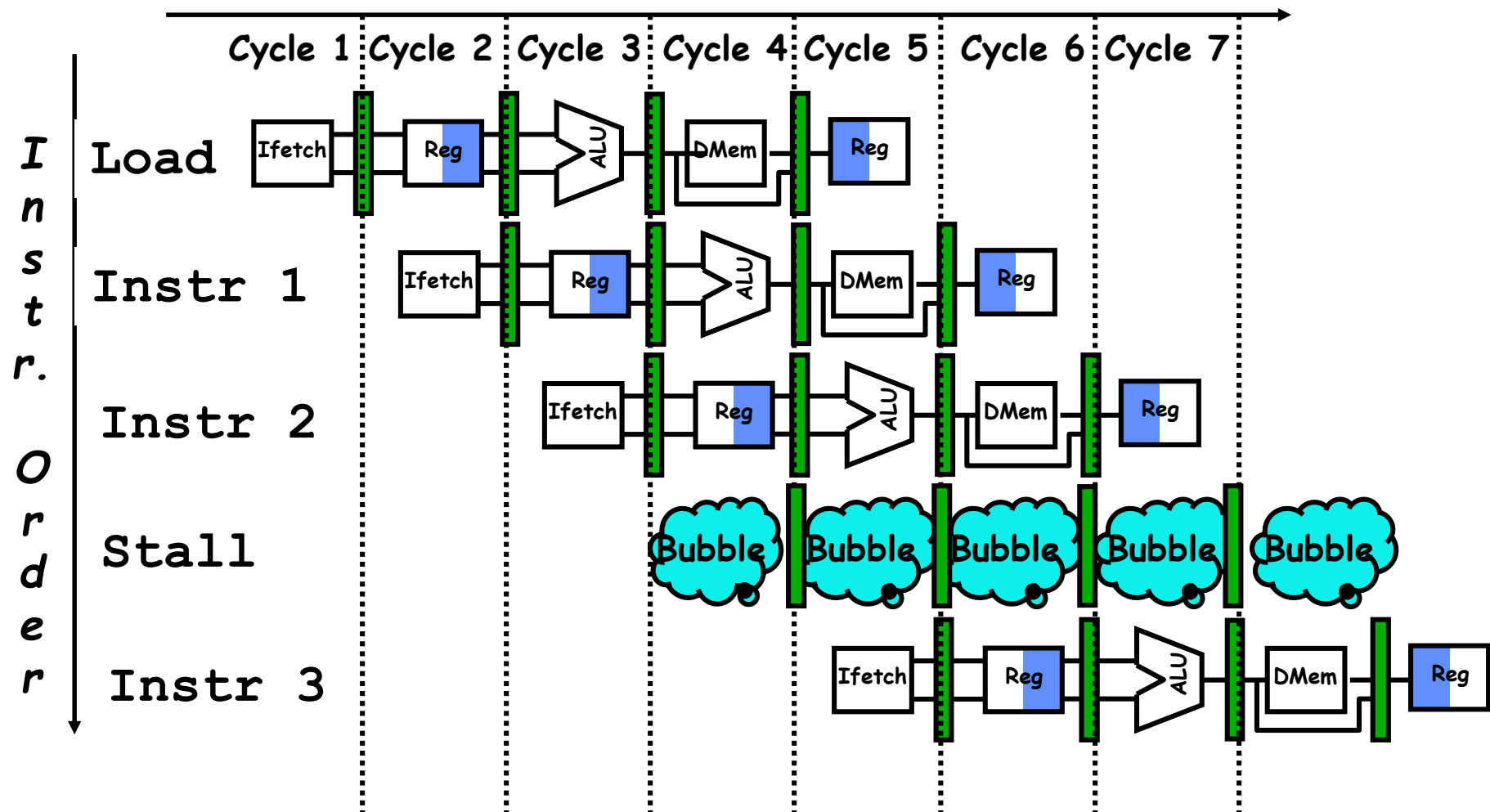
单个存储器引起的结构相关



Detection is easy in this case! (right half highlight means read, left half write)

消除结构相关

Time (clock cycles)



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**

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, $CPI = 1$:

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

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:

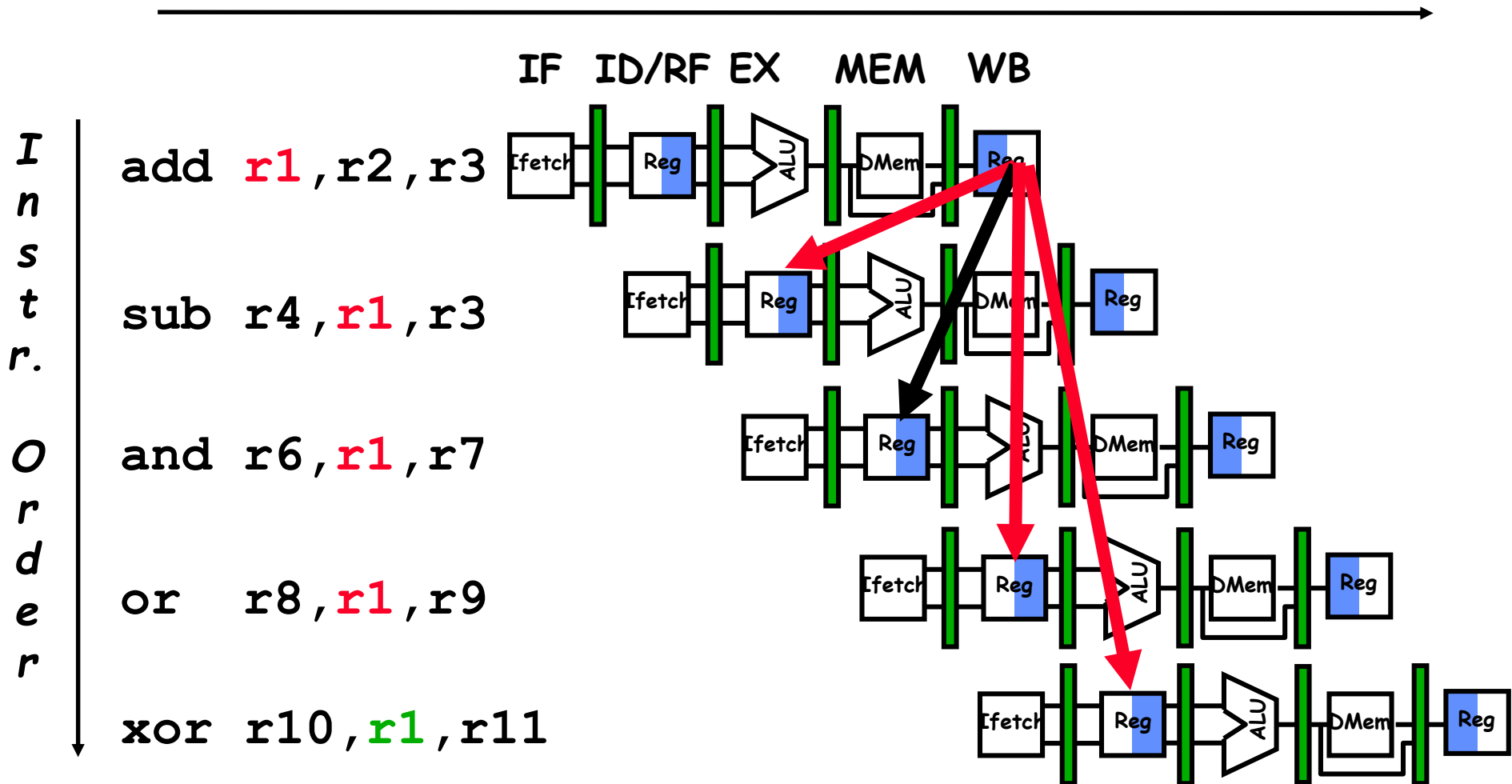
Average Instruction time = Clock cycle time

存在结构相关的机器B:

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


数据相关问题

Time (clock cycles)



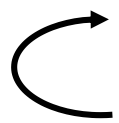
三种基本的数据相关

- 写后读相关(Read After Write (RAW))
Instr_j tries to read operand before Instr_i writes it

 I: add **r1**, r2, r3
J: sub r4, **r1**, r3

- 由于实际的数据交换需求而引起的

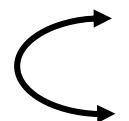
- 读后写相关Write After Read (WAR)
Instr_j writes operand before Instr_i reads it



```
I: sub r4, r1, r3
J: add r1, r2, r3
K: mul r6, r1, r7
```

- 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“r1”引起的.
- **DLX(MIPS) 5 段基本流水线**不会有此类相关因为:
 - 所有的指令都是**5段**, 并且
 - 读操作总是在第**2段**, 而
 - 写操作在第**5段**

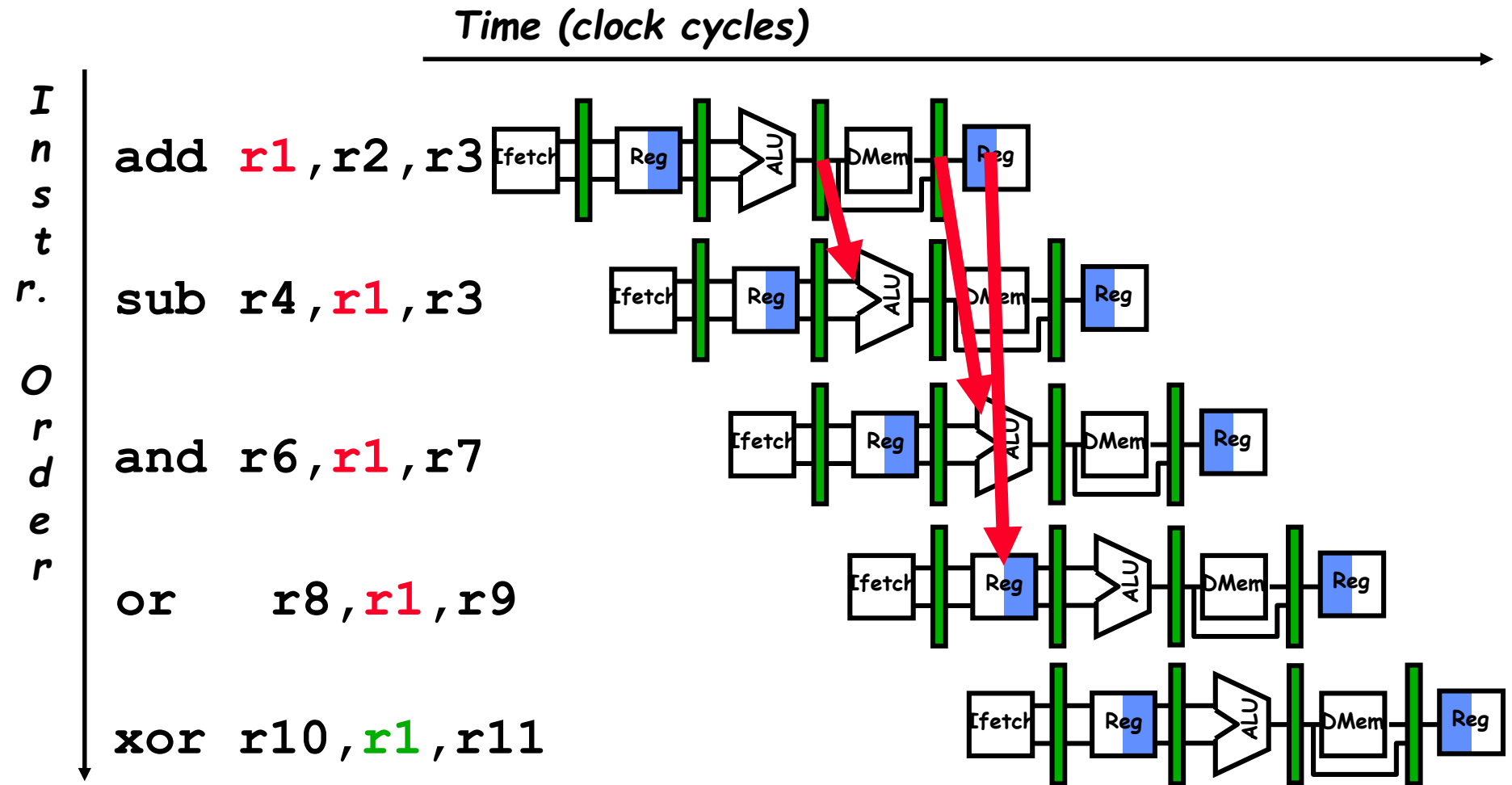
- 写后写相关（**Write After Write (WAW)**）
Instr_j writes operand before Instr_i writes it.



```
I: sub r1, r4, r3
J: add r1, r2, r3
K: mul r6, r1, r7
```

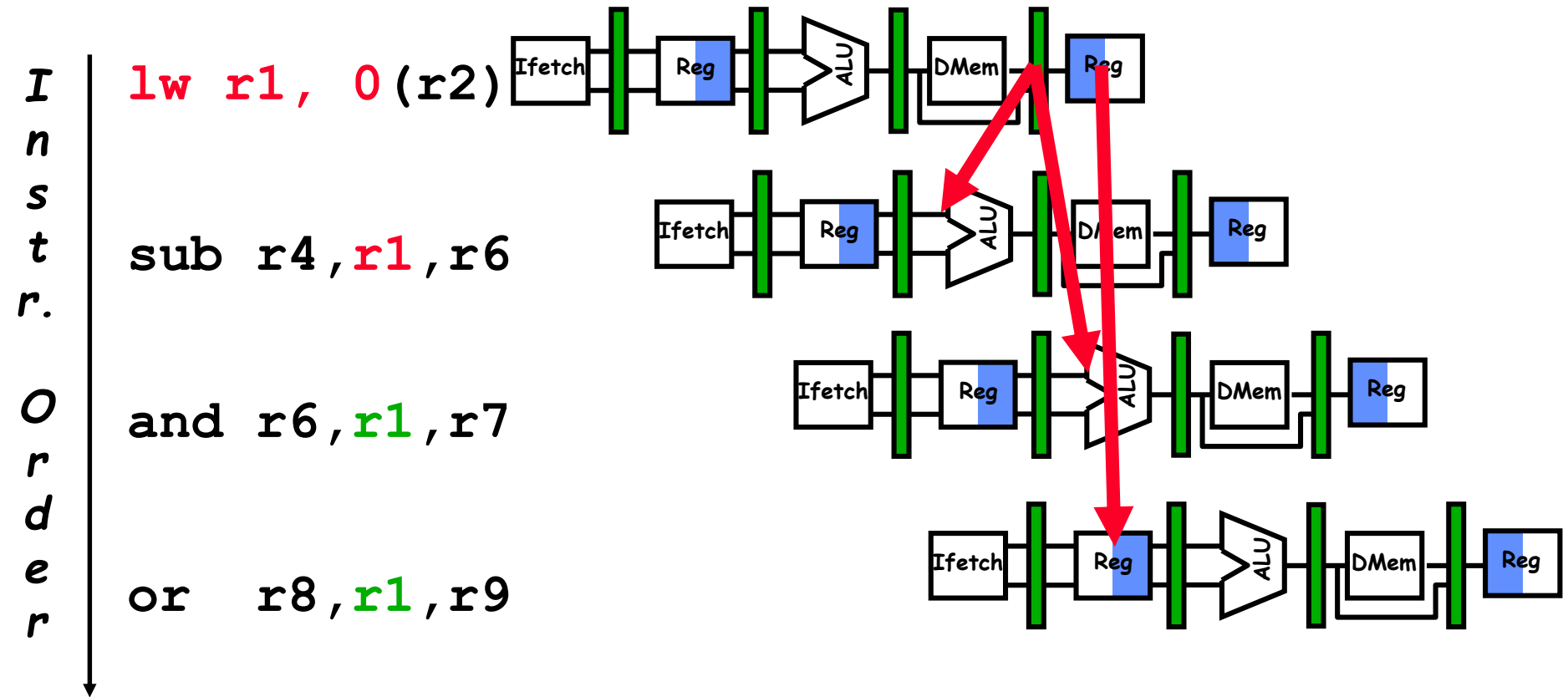
- 编译器编写者称之为“**output dependence**”，也是由于重复使用寄存器名“**r1**”引起的。
- 在**DLX(MIPS)** 5段基本流水线中，也不会发生。因为
 - 所有指令都是5段，并且写操作都在第5段
- 在后面的复杂的流水线中我们将会看到 **WAR** 和**WAW** 相关

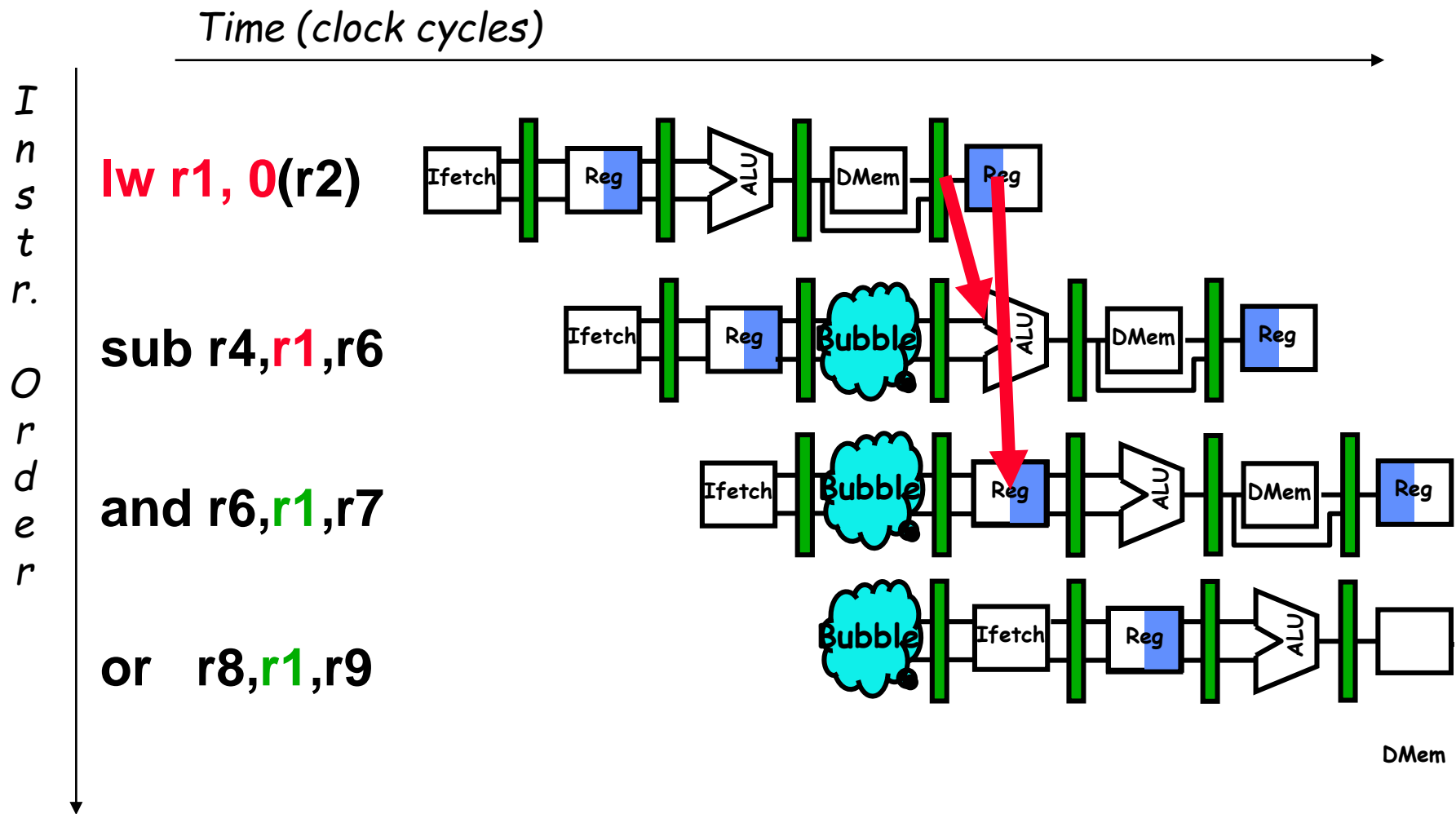
采用定向技术避免数据相关



采用定向技术仍然存在相关

Time (clock cycles)





采用软件方法避免数据相关

Try producing fast code for

$a = b + c;$

$d = e - f;$

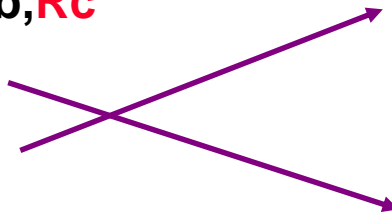
assuming $a, b, c, d, e,$ and f in memory.

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:

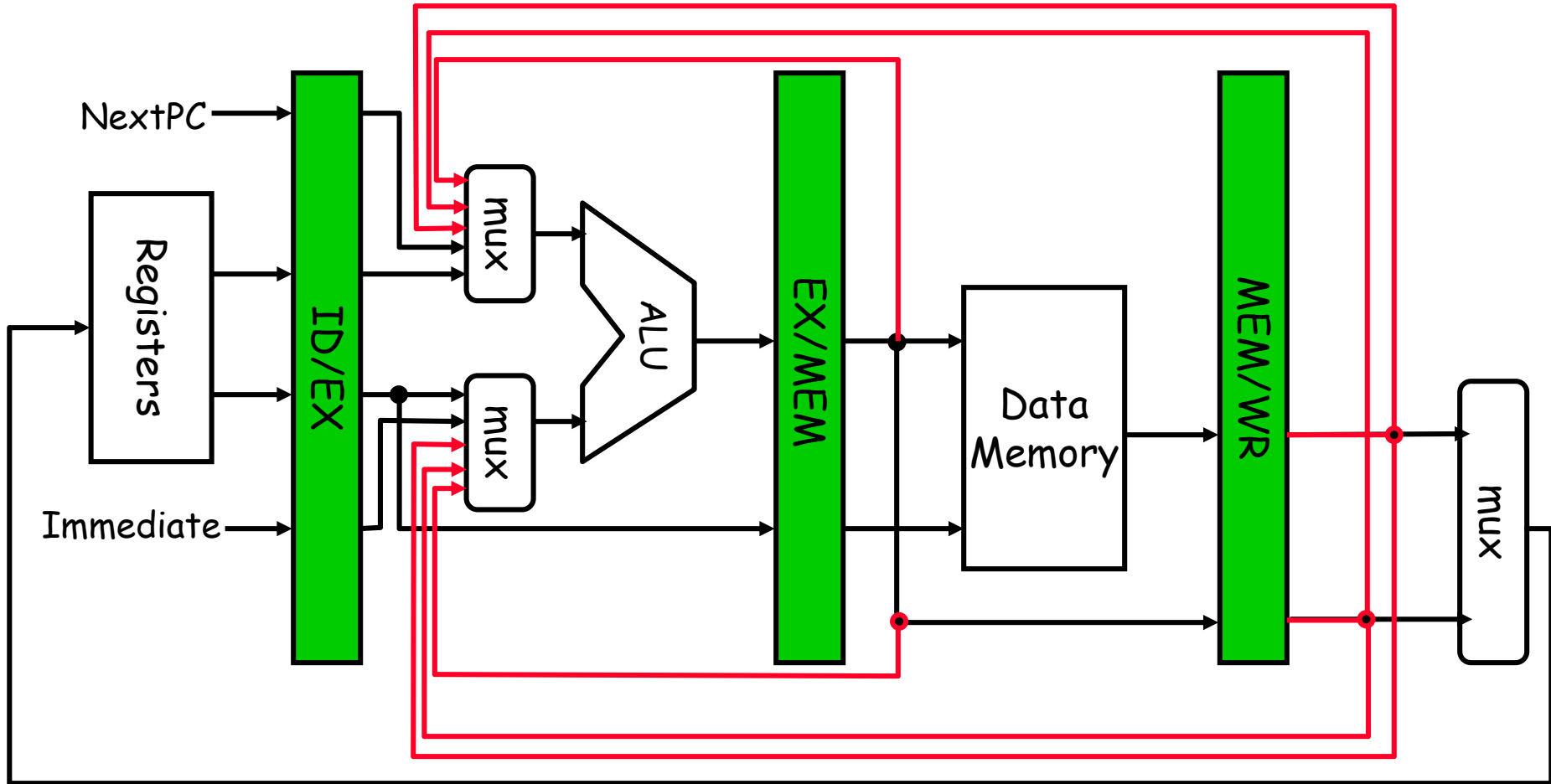
```
LW    Rb,b
LW    Rc,c
LW    Re,e
ADD   Ra,Rb,Rc
LW    Rf,f
SW    a,Ra
SUB   Rd,Re,Rf
SW    d,Rd
```



流水线相关检测部件能检测到的相关情况

Situation	Sample Code	Action Required
No dependence	LW R1,45(R2) ADD R5, R6, R7 SUB R8,R6,R7, OR R9,R6,R7	No hazard since R1 doesn't show up in the next 3 instructions
Dependence requiring stall	LW R1,45(R2) ADD R5, R1, R7 SUB R8,R6,R7, OR R9,R6,R7	comparators must detect the use of R1 in ADD and stall since LW can only produce the value after cycle 4 and ADD needs it after cycle 2 - hence no way
Dependence can be handled with forwarding	LW R1,45(R2) ADD R5, R6, R7 SUB R8,R1,R7, OR R9,R6,R7	comparators detect R1 use in SUB and setup forward of result to ALU from end of DM stage
Dependence but register file order splitting makes in a non-issue	LW R1,45(R2) ADD R5, R6, R7 SUB R8,R6,R7, OR R9,R1,R7	R1 will be placed in the first half of Rd in LW R1 will be read in the second half of RS for OR Hence no problem - be happy

采用定向技术硬件所需做的修改



定向源为R-R ALU操作的定向比较判断

Source Pipe Reg.	Opcode of Source Inst.	Dest. Pipe Reg	Opcode of Dest. Inst.	Destination of forwarded result	Compare if EQ then forward
EX/MEM	Reg-Reg ALU	ID/EX	Reg-Reg ALU ALU-imm, LD, ST, branch	Top ALU input	$Rd(EX/MEM.IR_{16..20}) = Rs1(ID/EX.IR_{6..10})$
EX/MEM	Reg-Reg ALU	ID/EX	Reg-Reg ALU	Bottom ALU input	$Rd(EX/MEM.IR_{16..20}) = Rs2(ID/EX.IR_{11..15})$
MEM/WB	Reg-Reg ALU	ID/EX	Reg-Reg ALU ALU-imm, LD, ST, branch	Top ALU input	$Rd(MEM/WB.IR_{16..20}) = Rs1(ID/EX.IR_{6..10})$
MEM/WB	Reg-Reg ALU	ID/EX	Reg-Reg ALU	Bottom ALU input	$Rd(MEM/WB.IR_{16..20}) = Rs2(ID/EX.IR_{6..10})$

定向源为ALU—imm操作的定向比较判断

Source Pipe Reg.	Opcode of Source Inst.	Dest. Pipe Reg	Opcode of Dest. Inst.	Destination of forwarded result	Compare if EQ then forward
EX/MEM	ALU Immediate	ID/EX	Reg-Reg ALU ALU-imm, LD, ST, branch	Top ALU input	$Rd(EX/MEM.IR_{11..15}) = Rs1(ID/EX.IR_{6..10})$
EX/MEM	ALU Immediate	ID/EX	Reg-Reg ALU	Bottom ALU input	$Rd(EX/MEM.IR_{11..15}) = Rs2(ID/EX.IR_{11..15})$
MEM/WB	ALU Immediate	ID/EX	Reg-Reg ALU ALU-imm, LD, ST, branch	Top ALU input	$Rd(MEM/WB.IR_{11..15}) = Rs1(ID/EX.IR_{6..10})$
MEM/WB	ALU Immediate	ID/EX	Reg-Reg ALU	Bottom ALU input	$Rd(MEM/WB.IR_{11..15}) = Rs2(ID/EX.IR_{11..15})$

定向源为Load操作的比较判定

Source Pipe Reg.	Opcode of Source Inst.	Dest. Pipe Reg	Opcode of Dest. Inst.	Destination of forwarded result	Compare if EQ then forward
MEM/WB	Load	ID/EX	Reg-Reg ALU ALU-imm, LD, ST, branch	Top ALU input	$Rd(MEM/WB.IR_{11..15}) = Rs1(ID/EX.IR_{6..10})$
MEM/WB	Load	ID/EX	Reg-Reg ALU	Bottom ALU input	$Rd(MEM/WB.IR_{11..15}) = Rs2(ID/EX.IR_{11..15})$

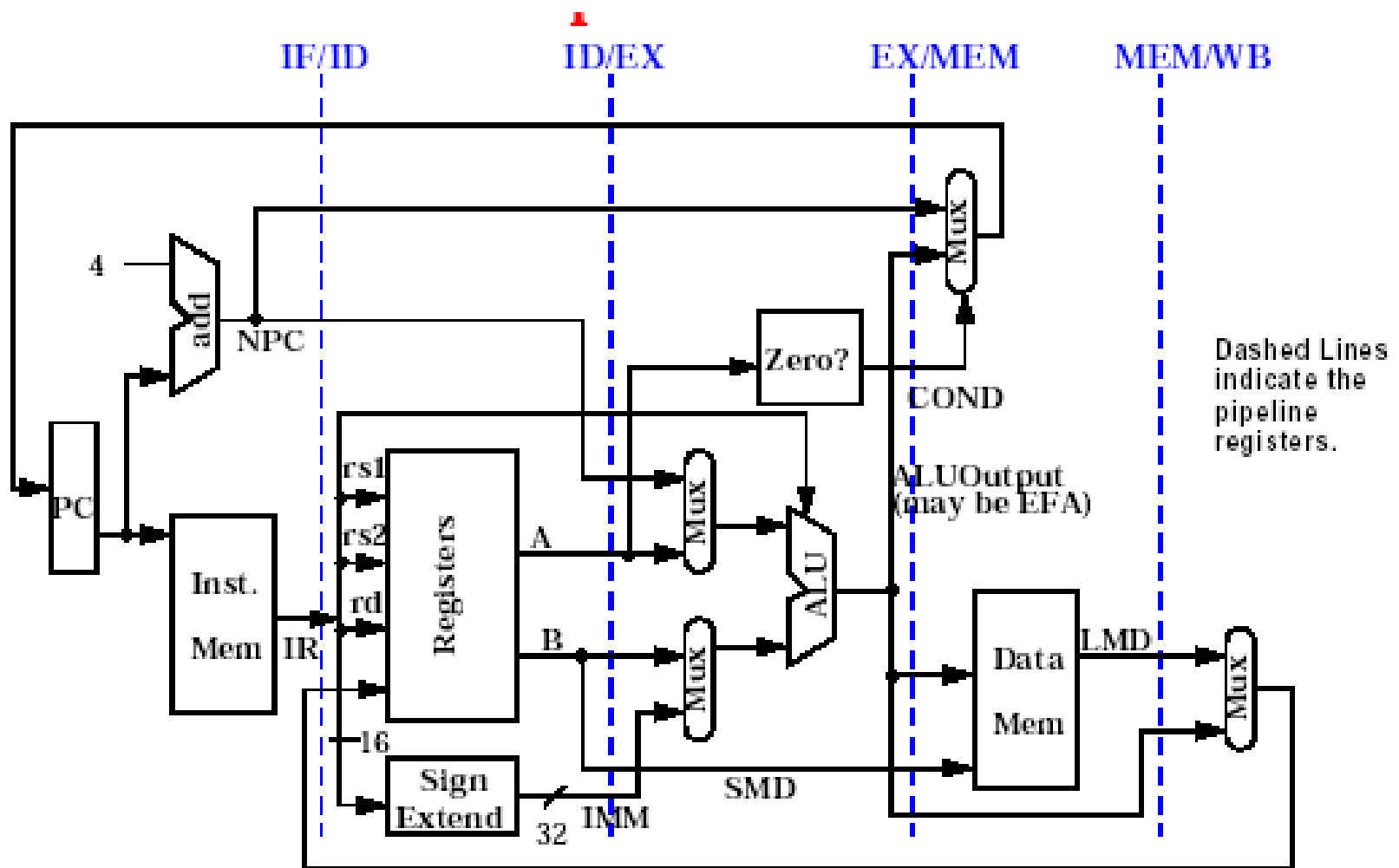
控制相关

- 分支需要解决两个问题
 - 分支目标地址（转移成功意味着PC值不是 $PC+4$ ）
 - CC是否有效，这两点在DLX(MIPS)中都在流水线的靠后段中确定
- 译码在ID段，此时取进来的指令可能是错误的指令
- 对于简单的DLX(MIPS)流水线 - 3 cycle branch penalty
 - 有效地址在EX段才能确定
 - 条件是否为真在MEM段
 - 因此有3个stall
- 流水线的时空图

控制冲突

- 执行分支指令的结果有两种
 - 分支成功：PC值改变为分支转移的目标地址。
在条件判定和转移地址计算都完成后，才改变PC值。
 - 不成功或者失败：PC的值保持正常递增，指向顺序的下一条指令。
- 分支需要解决两个问题
 - 分支目标地址（转移成功意味着PC值不是 $PC+4$ ）
 - CC是否有效，这两点在DLX(MIPS)中都在流水线的靠后段中确定
- 处理分支指令最简单的方法：
“冻结”或者“排空”流水线。
优点：简单。

回顾DLX (MIPS)数据通路



简单处理分支指令：分支成功的情况

[illegible]

简单处理分支指令：分支失败的情况

[illegible]

减少分支延时的方法

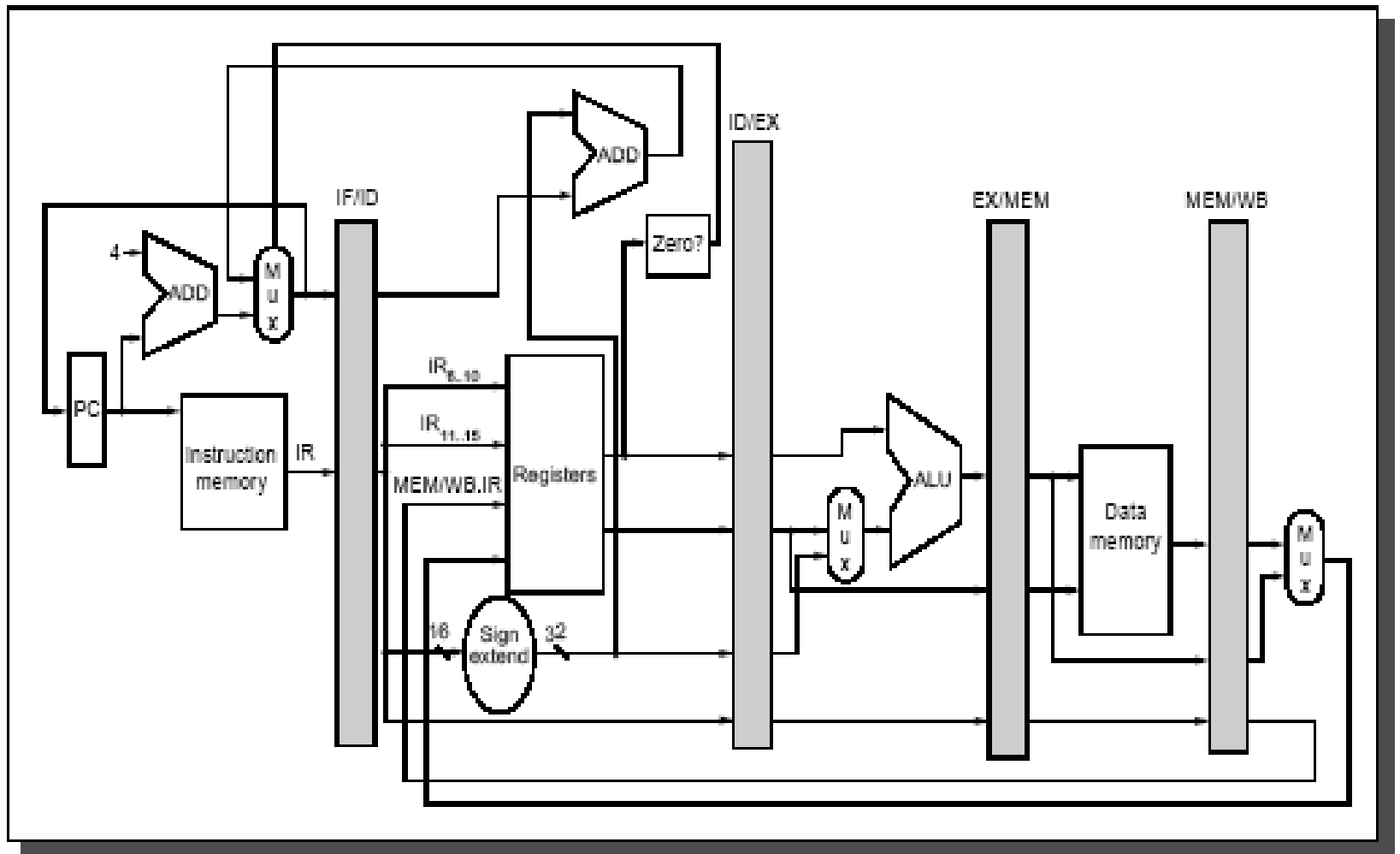
- 硬件的方法

- 修改数据通路：使得目标地址和分支条件尽早确定，其中之一尽早确定是没有用的
 - » 判断是否为0可以在ID段确定
 - » 使用另一个加法器计算
 - 可以在ID段计算BTA(分支目标地址)，即在ID段形成下一条指令地址，两种可能（BTA, PC+4），选择哪一个取决于ID段确定的CC
 - » 必要时使用互锁机制来插入Stall•
- 设计合适的ISA
 - » e.g. BNEZ, BEQZ on DLX 使得CC可以在ID段确定

- 软件（通过编译器）的方法：

- 调度一些指令放入分支的延迟槽中
- 预测的方法：统计分支成功和失败的情况，提高预测精度

新的DLX (MIPS)数据通路



改进后流水线的分支操作

Pipe stage	Branch instruction
IF	$IF/ID.IR \leftarrow Mem[PC];$ $IF/ID.NPC, PC \leftarrow (if ((IF/ID.opcode == branch) \& (Regs[IF/ID.IR_6..10]$ $op\ 0)) \{IF/ID.NPC +$ $(IF/ID.IR_{16})^{16} \# IF/ID.IR_{16..31}\} \text{ else } \{PC+4\});$
ID	$ID/EX.A \leftarrow Regs[IF/ID.IR_6..10]; ID/EX.B \leftarrow Regs[IF/ID.IR_{11..15}];$ $ID/EX.IR \leftarrow IF/ID.IR;$ $ID/EX.Imm \leftarrow (IF/ID.IR_{16})^{16} \# IF/ID.IR_{16..31}$
EX	
MEM	
WB	

四种可能的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
- **DLX (MIPS)** 分支指令平均**47%**为分支失败

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。

- 平均**53% DLX (MIPS)** 分支为分支成功
- **DLX (MIPS)** 分支目标地址在**ID段**才能计算出目标地址
 - » **DLX (MIPS)** 还是有**1个 cycle branch penalty**

延迟转移

#4: 延迟转移

主要思想:

从逻辑上“延长”分支指令的执行时间。把延迟分支看成是由原来的分支指令和若干个延迟槽构成，不管分支是否成功，都要按顺序执行延迟槽中的指令。

- 定义分支发生在一系列指令之后

`branch instruction`

`sequential successor1`

`sequential successor2`

`.....`

`sequential successorn`

`branch target if taken`



Branch delay of length n

- 5级流水只需要一个延迟槽就可以确定目标地址和确定条件
- DLX 使用这种方式

具有一个分支延迟槽的流水线的执行过程

分支失败	分支指令 i	IF	ID	EX	MEM	WB				
	延迟槽指令 $i+1$		IF	ID	EX	MEM	WB			
	指令 $i+2$			IF	ID	EX	MEM	WB		
	指令 $i+3$				IF	ID	EX	MEM	WB	
	指令 $i+4$					IF	ID	EX	MEM	WB

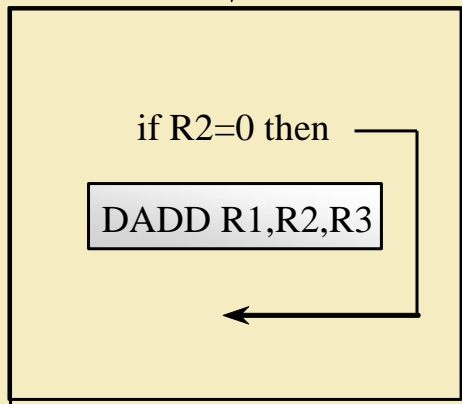
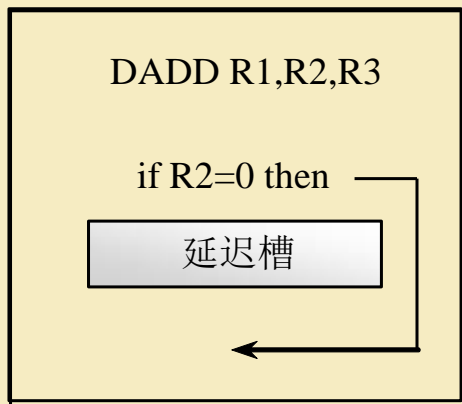
分支成功	分支指令 i	IF	ID	EX	MEM	WB				
	延迟槽指令 $i+1$		IF	ID	EX	MEM	WB			
	分支目标指令 j			IF	ID	EX	MEM	WB		
	分支目标指令 $j+1$				IF	ID	EX	MEM	WB	
	分支目标指令 $j+2$					IF	ID	EX	MEM	WB

分支延迟槽中的指令“掩盖”了流水线原来必须插入的暂停周期。

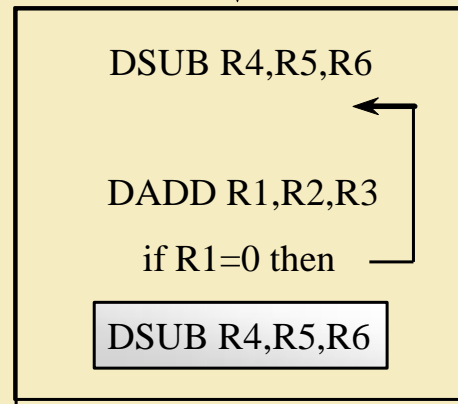
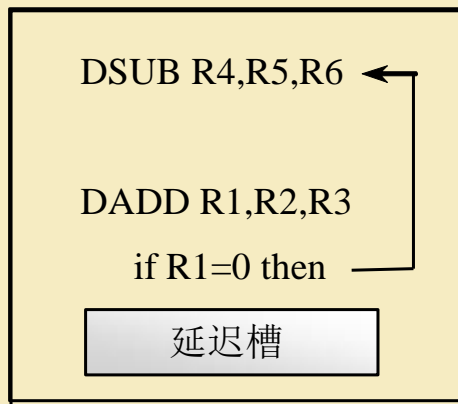
延迟转移

- 从何处选择指令来填充延迟槽？
 - 分支指令之前的指令：最好
 - 从分支目标地址处取：在分支成功可能性大时，这种策略较好
 - 从分支失败处调度：仅在分支失败时
- 编译器可以有效的调度一个延迟槽
 - 如果提供取消分支时，编译器可以调度更多的指令填入延迟槽

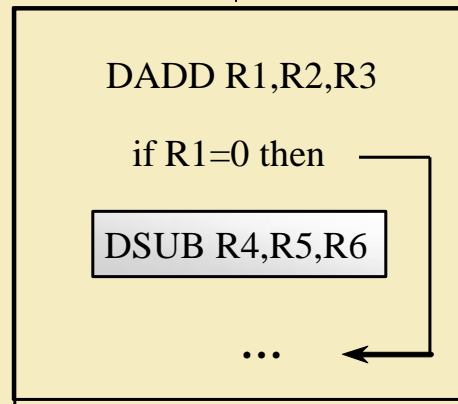
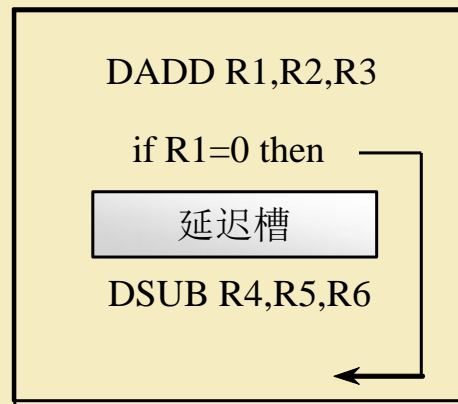
调度前和调度后的代码



(a) 从前调度



(b) 从目标处调度



(c) 从失败处调度

三种方法的要求及效果

调 度 策 略	对调度的要求	什么情况下起作用
从 前 调 度	被调度的指令必须与分支无关	任何情况
从目标处调度	必须保证在分支失败时执行被调度的指令不会导致错误。有可能需要复制指令	分支成功时 (但由于复制指令，有可能会增大程序空间)
从失败处调度	必须保证在分支成功时执行被调度的指令不会导致错误	分支失败时

分支取消机制

- 分支延迟受到两个方面的限制：
 - 可以被放入延迟槽中的指令要满足一定的条件。
 - 编译器预测分支转移方向的能力。

- 进一步改进：分支取消机制

当分支的实际执行方向和事先所预测的一样时执行分支延迟槽中的指令，否则就将分支延迟槽中的指令转化成一个空操作。

分支取消机制示意

分支失败	分支指令i	IF	ID	EX	MEM	WB				
	延迟槽指令 i+1		IF	idle	idle	idle	idle			
	指令 i+2			IF	ID	EX	MEM	WB		
	指令 i+3				IF	ID	EX	MEM	WB	
	指令 i+4					IF	ID	EX	MEM	WB

分支成功	分支指令i	IF	ID	EX	MEM	WB				
	延迟槽指令 i+1		IF	ID	EX	MEM	WB			
	分支目标指令j			IF	ID	EX	MEM	WB		
	分支目标指令j+1				IF	ID	EX	MEM	WB	
	分支目标指令j+2					IF	ID	EX	MEM	WB

预测分支成功的情况下，分支取消机制的执行情况

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

3.4 异常处理

- 流水线使得系统的吞吐率提高
- 问题:由于相关会影响系统性能的发挥
- 更严重的问题: 异常
- **Why?**
 - 多级流水—多周期指令
 - 异常可以发生在任何地方
 - 指令序与异常序可能不同
 - 必须按指令序处理异常
- 采用何种策略取决于异常的类型

异常的类型

- **I/O device request**
- **invoking an OS service from a user program**
 - e.g. via an unimplemented instruction on a Mac
- **tracing instruction execution**
- **breakpoint**
- **integer or FP arithmetic error such as overflow**
- **page fault**
- **misaligned memory access**
- **memory protection violation**
- **undefined instruction**
- **hardware malfunction - like parity or ECC error**
- **power failure**

异常响应请求的种类

- Synchronous vs. Asynchronous
 - synchronous caused by a particular instruction
 - asynchronous - external devices and HW failures
- User requested vs. Coerced
 - requested is predictable and can happen after the instruction
- User maskable vs. user non-maskable
 - e.g. arithmetic overflow on some machines is user maskable
- Within vs. Between instructions
 - within ==> synchronous, key is that completion is prevented
 - some asynchronous are also within
- Resume vs. Terminate program
 - implications for how much state must be preserved

例如

Exc. Type	Synchronous - Asynch.	Requested - Coerced	mask - non-mask	within - between	resume - terminate
I/O Device Req.	Asynch	Coerced	Non-maskable	Between	Resume
Invoke OS svc.	Synch	User Requested	Non-maskable	Between	Resume
Trace/Bkpoint	Synch	User Requested	Maskable	Between	Resume
Arith. exception	Synch.	Coerced	Maskable	Within	Resume
Page Fault	Synch.	Coerced	Non-maskable	Within	Resume
Misaligned addr.	Synch	Coerced	Maskable	Within	Resume
Mem. prot. violation	Synch	Coerced	Non-maskable	Within	Resume
Undefined Inst.	Synch.	Coerced - ???	Non-maskable	Within	Terminate
HW error	Asynch.	Coerced	Non-maskable	Within	Terminate
Power Failure	Asynch	Coerced	Non-maskable	Within	Terminate

最困难的问题

- 异常发生在指令中，并且要求恢复执行
- 要求==>流水线必须安全地 shut down
 - PC必须保存
 - 如果重新开始的是一条分支指令，它需要重新执行
 - 这意味着条件码状态必须没有改变
- 在DLX(MIPS)中的处理步骤
 - 强制trap指令在下一个IF段进入流水线
 - 封锁引起故障的指令的所有写操作和流水线中后继指令的写操作
 - 让所有前序指令执行完（如果能）
 - 保存重新执行时的地址（PC）
 - » PC 或 $PC + 1$
 - 调用OS处理异常

- 考虑延迟转移时，假设有两个延迟槽的分支

I	Branch Instr1
I+1	Delay instr1
I+2	Delay Instr2
I+3	inst
I+4	inst

- 假设**branch**指令是好的
- 第1个延迟指令引起缺页中断
- 第2条指令封锁
- 异常处理后，缺省的恢复点是第一条延迟指令
- 不会有Branch指令
- 因此需要保存的PC值不止一个，根据具体情况进行恢复

精确中断与非精确中断

- 引起异常的指令前面的指令都已执行完，故障后的指令可以重新从故障点后执行
- 理想情况，引起故障的指令没有改变机器的状态
- 要正确的处理这类异常请求，必须保证故障指令不产生副作用
- 在有些机器上，浮点数异常
 - 流水线段数多，在发现故障前，故障点后的指令就已经写了结果，在这种情况下，必须有办法处理。
- 当今很多高性能计算机，**Alpha 21164**，**MIPSR10000**等支持精确中断，但精确模式要慢**10**倍，一般用在代码调试时，很多系统要求精确中断模式，如**IEEE FP**标准处理程序，虚拟存储器等。
- 精确中断对整数流水线而言，不是太难实现

DLX (MIPS)中的异常

- IF
 - page fault, misaligned address, memory protection violation
- ID
 - undefined or illegal opcode
- EX
 - arithmetic exception
- MEM
 - page fault, misaligned address, memory protection violation
- WB
 - none

03-19-Review- 相关的处理

- 结构相关
 - 概念：由于争用资源而引起的
 - 解决办法
- 数据相关
 - 概念：由于存在实际的通信，而引起的
 - 解决办法：
 - » 硬件：定向技术（**forwarding**）
 - » 软件：指令级调度
- 控制相关
 - 概念：由于控制类指令引起的
 - 解决办法 ???

03-19-Review (续)

- 控制相关
 - 概念:
 - 减少性能损失的基本方法
 - » 冻结或排空流水线
 - » 预测分支成功
 - » 预测分支失败
 - » 延迟转移
- 异常
 - 异常的分类
 - 精确中断和非精确中断

3.5 DLX (MIPS)中多周期操作的处理

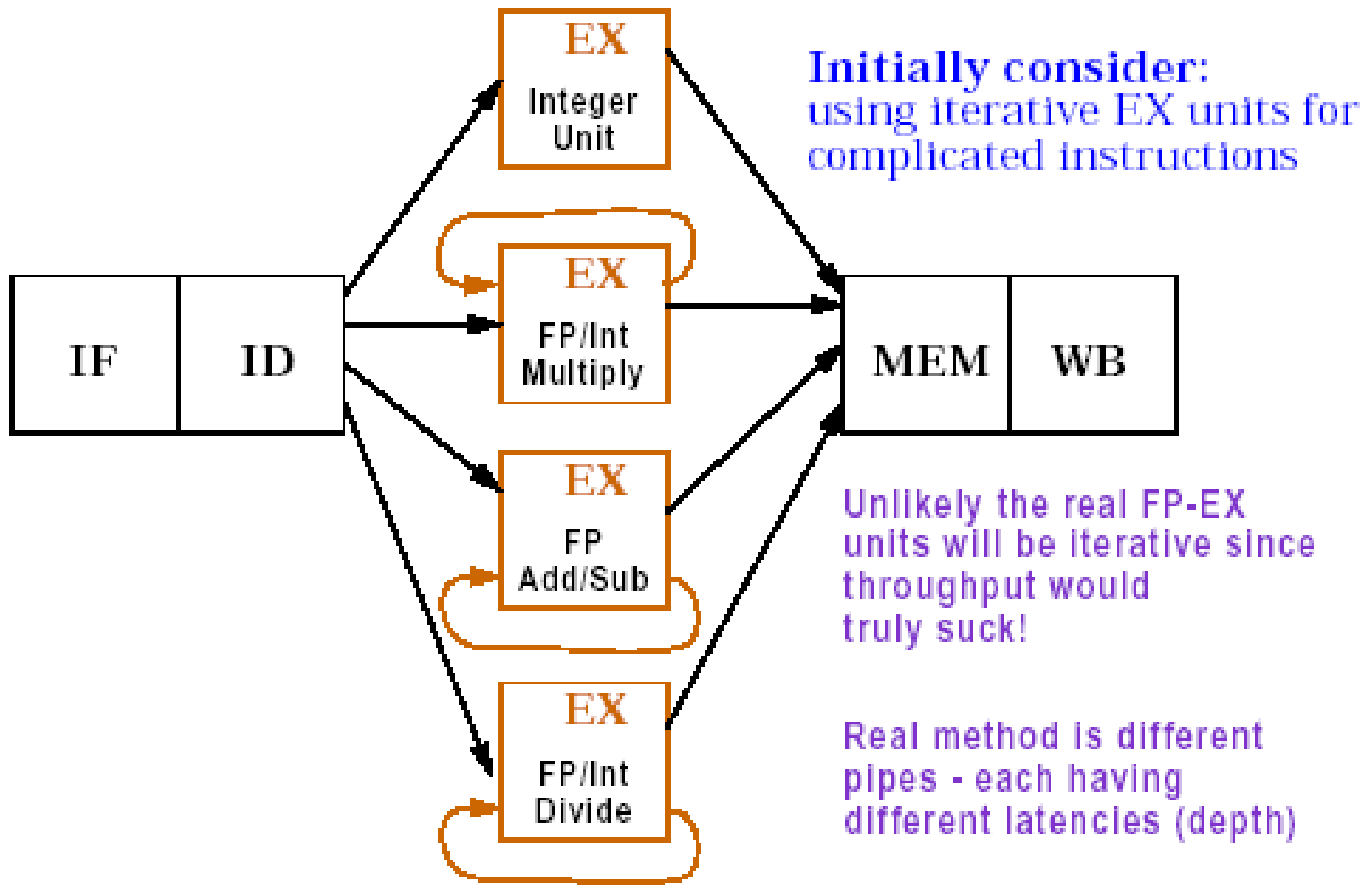
- 问题
 - 浮点操作在1~2个**cycles**完成是不现实的，一般要花费较长时间
 - 在**DLX(MIPS)**中如何处理
- 在1到2个**cycles**时间内完成的处理方法
 - 采用较慢的时钟源，或
 - 在**FP**部件中延迟其**EX**段
- 现假设**FP**指令与整数指令采用相同的流水线，那么
 - **EX** 段需要循环多次来完成**FP**操作，循环次数取决于操作类型
 - 有多个**FP**功能部件，如果发射出的指令导致结构或数据相关，需暂停

对DLX(MIPS)的扩充

四个功能部件

- **Integer** 部件处理: **Loads, Store, Integer ALU**操作和 **Branch**
- **FP/Integer** 乘法部件: 处理浮点数和整数乘法
- **FP**加法器: 处理**FP**加, 减和类型转换
- **FP/Integer**除法部件: 处理浮点数和整数除法
- 这些功能部件未流水化

扩展的DLX(MIPS)流水线



Latency & Repeat Interval

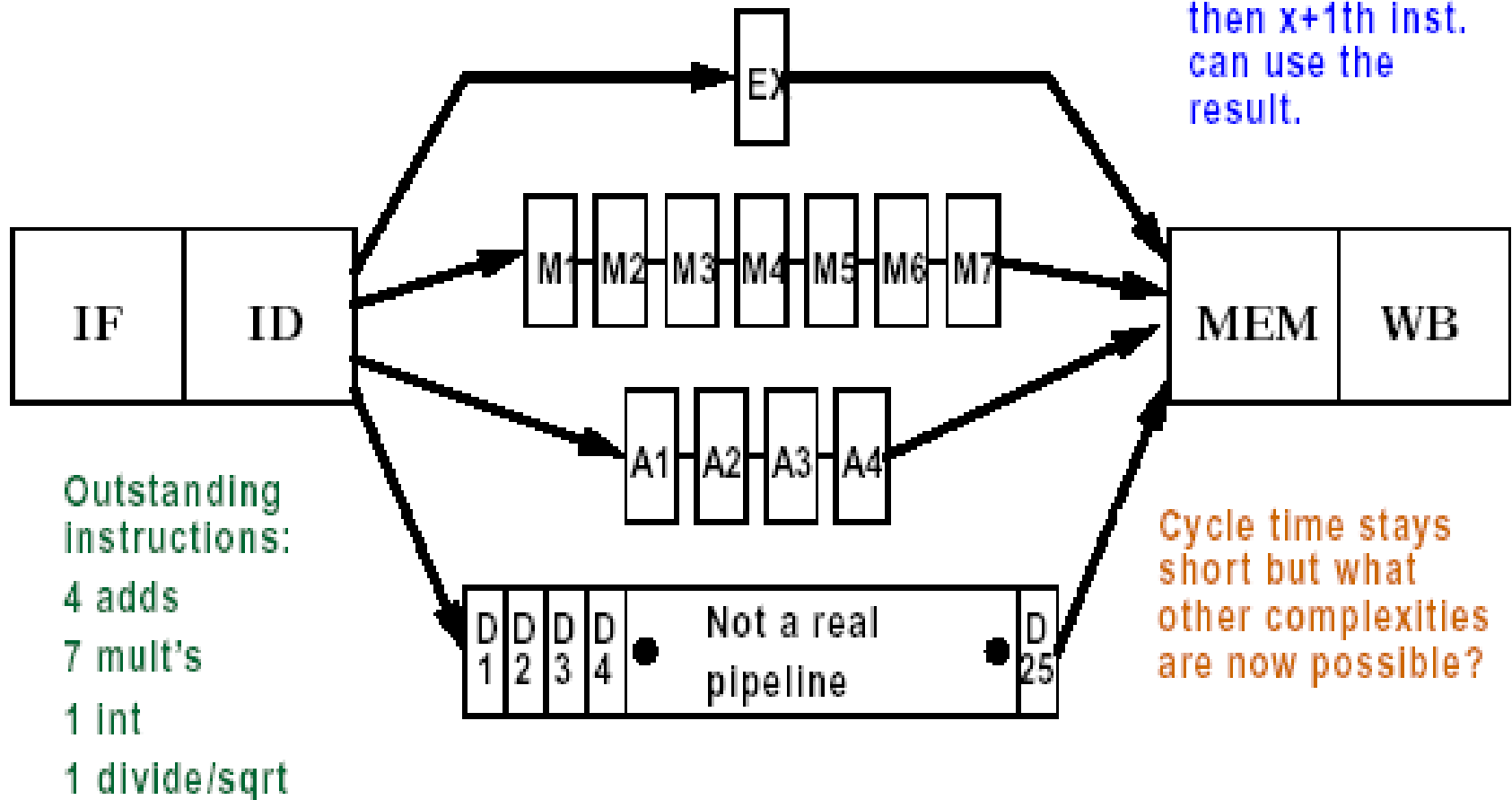
- 延时(Latency)
 - 定义为完成某一操作所需的cycle数
 - 定义为使用当前指令所产生结果的指令与当前指令间的最小间隔周期数
- 循环间隔（Repeat/Initiation interval）
 - 发射相同类型的操作所需的间隔周期数
- 对于**EX**部件流水化的新的**DLX(MIPS)**

Function Unit	Latency	Repeat Interval
Integer ALU	0	1
Data Memory (Integer and FP loads(1 less for store latency))	1	1
FP Add	3	1
FP multiply	6	1
FP Divide (also integer divide and FP sqrt)	24	25

将部分执行部件流水化后的DLX(MIPS)流水线

Note # of stages are $1 + \text{latency}_{EX}$

If Latency = x ,
then $x+1$ th Inst.
can use the
result.



新的相关和定向问题

- 结构冲突增多
 - 非流水的Divide部件，使得EX段增长24个cycles
 - 在一个周期内可能有多个寄存器写操作
- 可能指令乱序完成（乱序到达WB段）有可能存在WAW
- 由于在ID段读，还不会有 WAR 相关
- 乱序完成导致异常处理复杂
- 由于指令的延迟加大导致RAW 相关的stall数增多
- 需要付出更多的代价来增加定向路径

新的结构相关

Instruction	1	2	3	4	5	6	7	8	9	10	11
MULTD F0, F4, F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
...		IF	ID	EX	MEM	WB					
...			IF	ID	EX	MEM	WB				
ADDD F2, F4, F6				IF	ID	A1	A2	A3	A4	MEM	WB
...					IF	ID	EX	MEM	WB		
...						IF	ID	EX	MEM	WB	
LD F8, 0(R2)							IF	ID	EX	MEM	WB

- 纵向检查指令所使用的资源
 - 第10个cycle，三条指令同时进入MEM，但由于MULTD和ADDD在MEM段没有实际动作，这种情况没有关系
 - 第11个cycle，三条指令同时进入WB段，存在结构相关

解决方法

- **Option 1**

- 在ID段跟踪写端口的使用情况，以便能暂停该指令的发射
- 一旦发现冲突，暂停当前指令的发射

- **Option 2**

- 在进入MEM或WB段时，暂停冲突的指令，让有较长延时的指令先做，因为较长延时的指令，会更容易引起其他RAW相关，从而导致更多的stalls

关于数据相关

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
LD F4,0(R2)	IF	ID	EX	M	WB										
MULTD F0, F4, F6		IF	ID	St.	M1 F4	M2	M3	M4	M5	M6	M7 F0	M	WB		
ADDD F2, F0, F8			IF	St.	ID	St.	St.	St.	St.	St.	St.	A1	A2	A3	A4
SD F2, 0(R2)					IF	St.	St.	St.	St.	St.	St.	ID	EX	St.	St.

Note: Figure 3.46 in text is wrong

*finally on cycle 16 SD gets to enter Mem
EX doesn't need to stall since it's the EFA
calculation which uses R2*

新的冲突源

- **GPR与FPR间的数据传送造成的数据相关**
 - **MOVI2FP and MOVFP2I instructions**
- 如果在**ID**段进行相关检测，指令发射前须做如下检测：
 - 结构相关
 - » 循环间隔检测
 - » 确定寄存器写端口是否可用
 - **RAW**相关
 - » 列表所有待写的目的寄存器
 - » 不发射以待写寄存器做为源寄存器的指令，直到该寄存器值可用
 - **WAW**相关
 - » 仍然使用上述待写寄存器列表
 - » 不发射那些目的寄存器在待写寄存器列表中的指令，直到对应的待写寄存器值可用(完成**WB**)。

精确中断与长流水线

- 例如

DIVF F0, F2, F4

ADDF F10, F10, F8

SUBF F12, F12, F14

- ADDF 和SUBF都在DIVF前完成
- 如果DIVF导致异常，会如何？
 - 非精确中断
- Ideas???

处理中断4种可能的办法

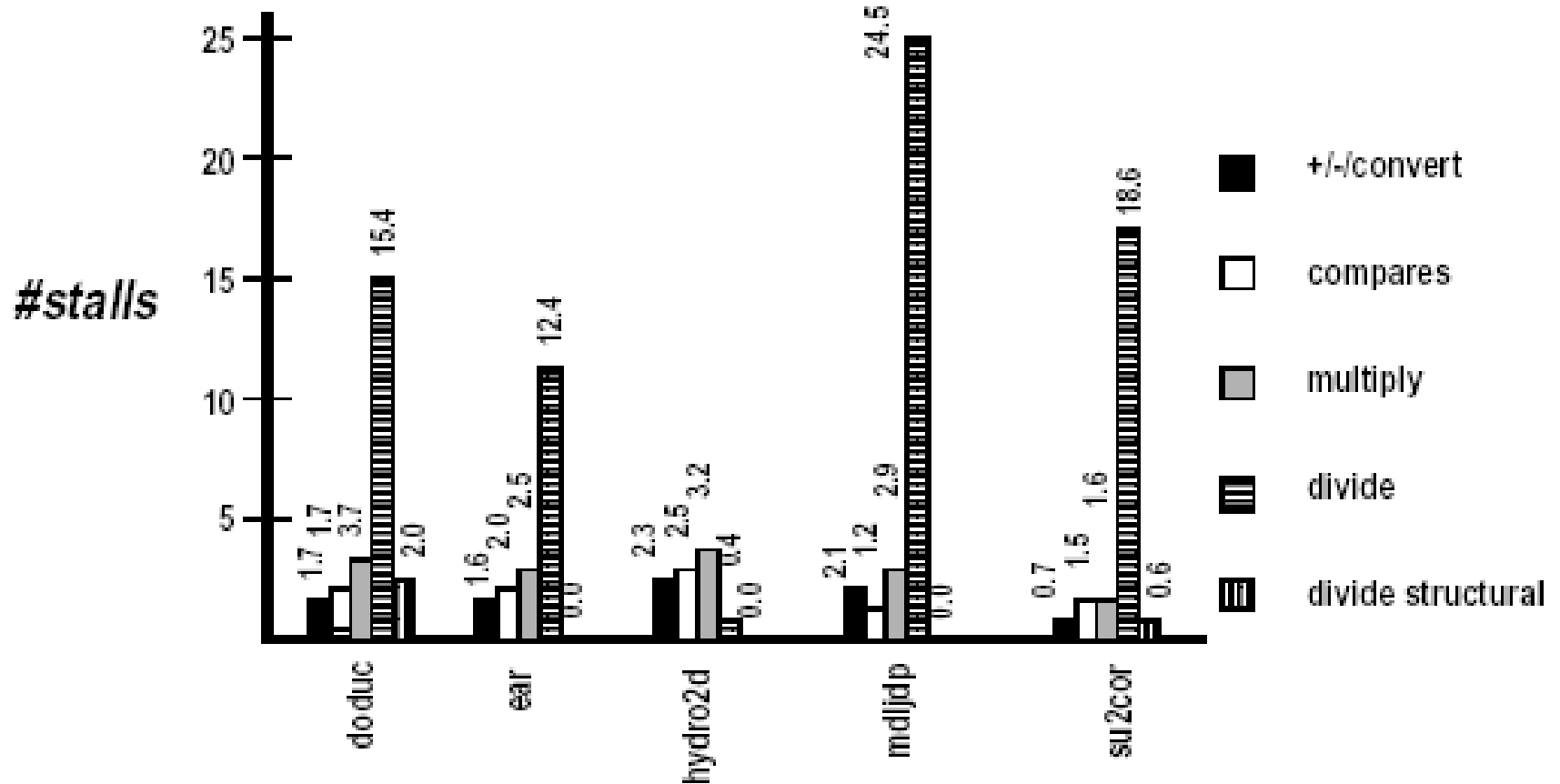
- 方法1：忽略这种问题，当非精确处理
 - 原来的supercomputer的方法
 - 但现代计算机对IEEE 浮点标准的异常处理，虚拟存储的异常处理要求必须是精确中断。
- 方法2：缓存操作结果，直到早期发射的指令执行完。
 - 当指令运行时间较长时，Buffer区较大
 - **Future file (Power PC620 MIPS R10000)**
 - » 缓存执行结果，按指令序确认
 - **history file (CYBER 180/990)**
 - » 尽快确认
 - » 缓存区存放原来的操作数，如果异常发生，回卷到合适的状态

第3 & 4种方法

- 以非精确方式处理，用软件来修正
 - 为软件修正保存足够的状态
 - 让软件仿真尚未执行完的指令的执行
 - 例如
 - » **Instruction 1 – A** 执行时间较长，引起中断的指令
 - » **Instruction 2, instruction 3,instruction n-1** 未执行完的指令
 - » **Instruction n** 已执行完的指令
 - » 由于第**n**条指令已执行完，中断返回地址为第**n+1**条指令，如果我们保存所有的流水线的PC值，那么软件可以仿真**Instruction1** 到 **Instruction n-1** 的执行
- 暂停发射，直到确定先前的指令都可无异常的完成，再发射下面的指令。
 - 在EX段的前期确认（MIPS流水线在前三个周期中）
 - MIPS R2K to R4K 以及Pentium使用这种方法

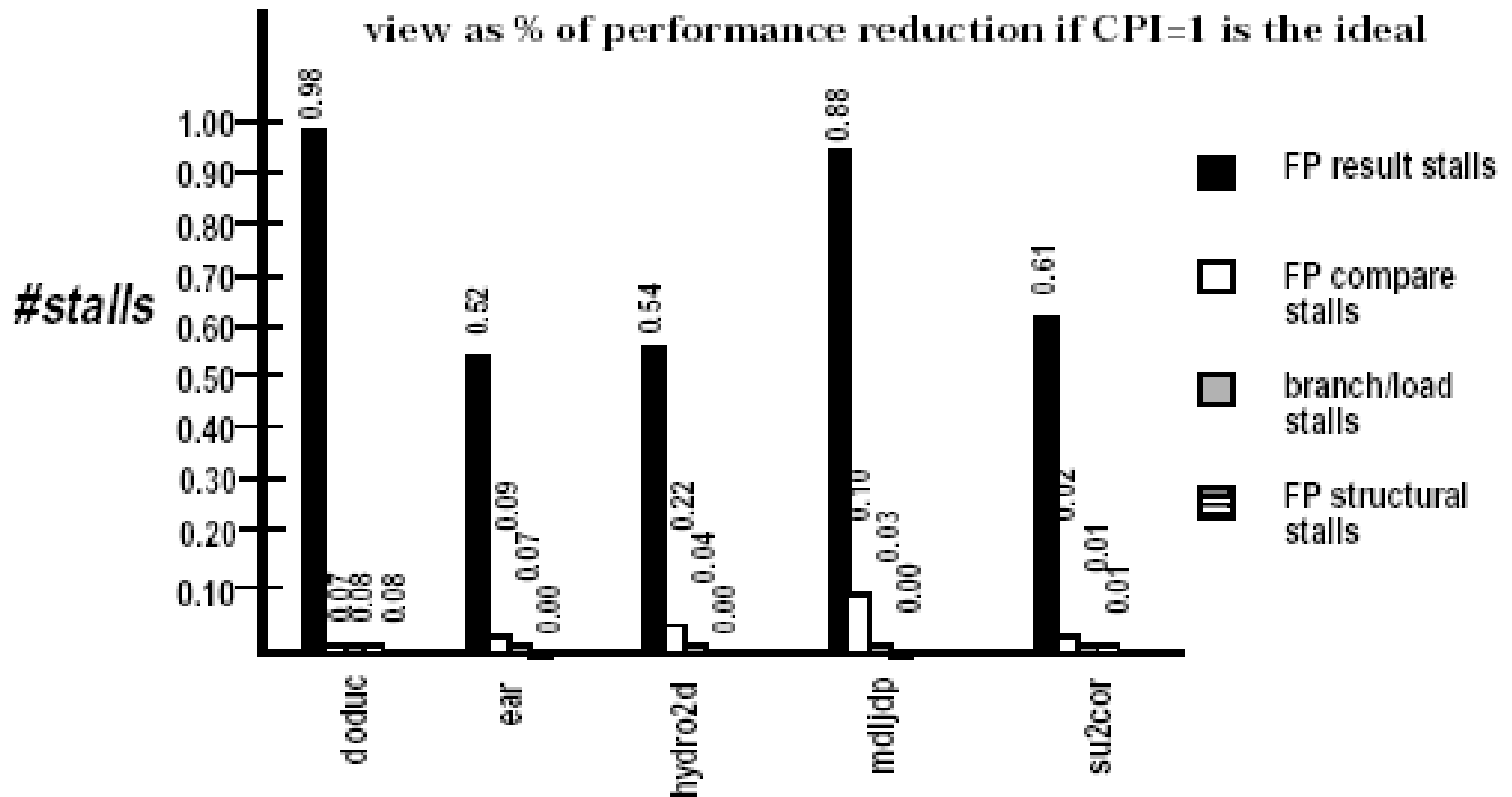
DLX(MIPS)流水线的性能

stalls per FP operation



Stalls per FP operation for each major type of FP operation for the SPEC89 FP benchmarks

平均每条指令的stall数

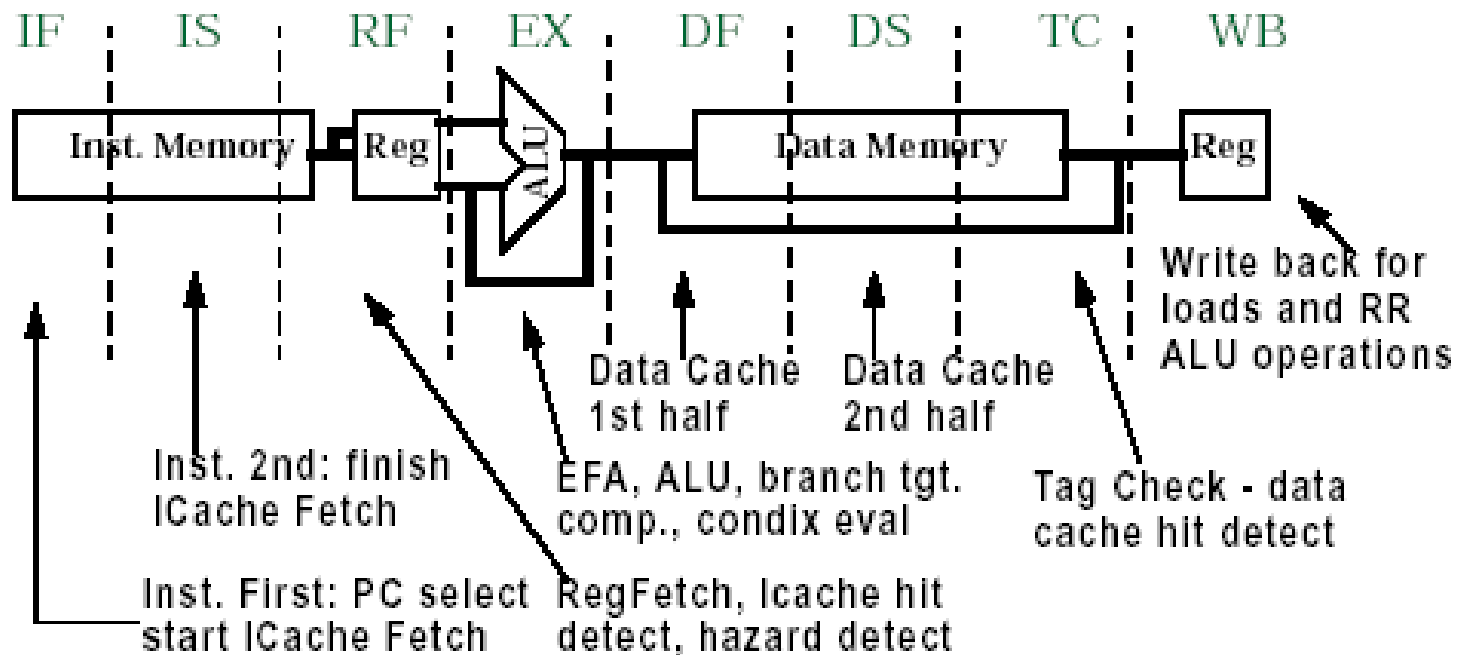


The stalls occurring for the MIPS FP pipeline for five for the SPEC89 FP benchmarks.

MIPS R4000

实际的 64-bit 机器

- 主频 100MHz ~ 200MHz
- 较深的流水线（级数较多）（有时也称为 *superpipelining*）
- 指令集与 DLX 非常类似



MIPS R4000的8 级整数流水线

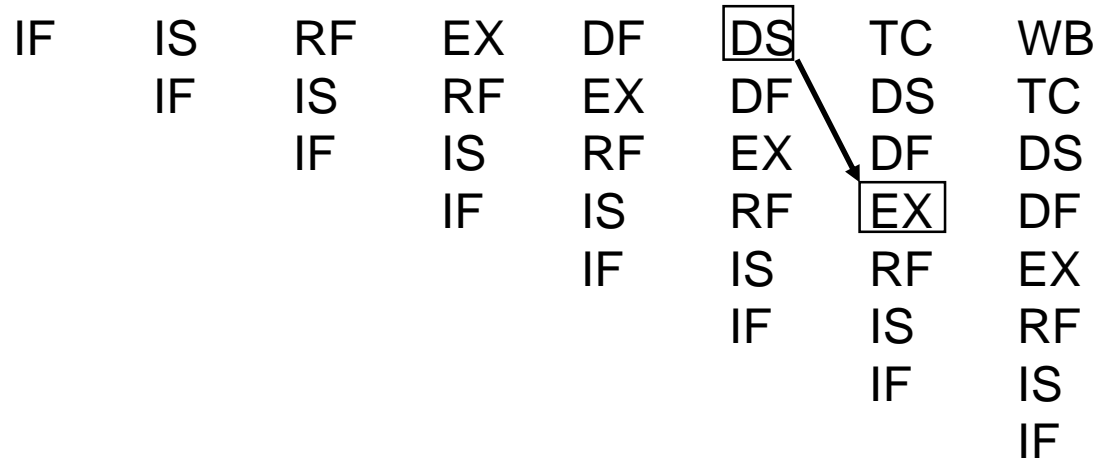
- **IF**–取指阶段的前半部分；选择**PC**值，初始化指令**cache**的访问
- **IS**–取指阶段的后半部分，主要完成访问指令**cache**的操作
- **RF**–指令译码，寄存器读取，相关检测以及指令**cache**命中检测
- **EX**–执行，包括：计算有效地址，进行**ALU**操作，计算分支目标地址和检测分支条件
- **DF**–取数据，访问数据**cache**的前半部分
- **DS**–访问数据**cache**的后半部分
- **TC**–tag 检测，确定数据**cache**是否命中
- **WB**–Load操作和R-R操作的结果写回

需注意的问题

- 在使用定向技术的情况下，**Load** 延迟为**2个cycles**
 - **Load**和与其相关的指令间必须有**2条指令**或**两个bubbles**
 - 原因：**load**的结果在**DS**结束时可用
- 分支延迟**3个cycles**
 - 分支与目标指令间需要**3条指令**或**3个bubbles**
 - 原因：目标地址在**EX**段后才能知道
- **R4000**的流水线中，到**ALU**输入端有四个定向源
 - **EX/DF, DF/DS, DS/ TC, TC/WB**

图示

TWO Cycle Load Latency

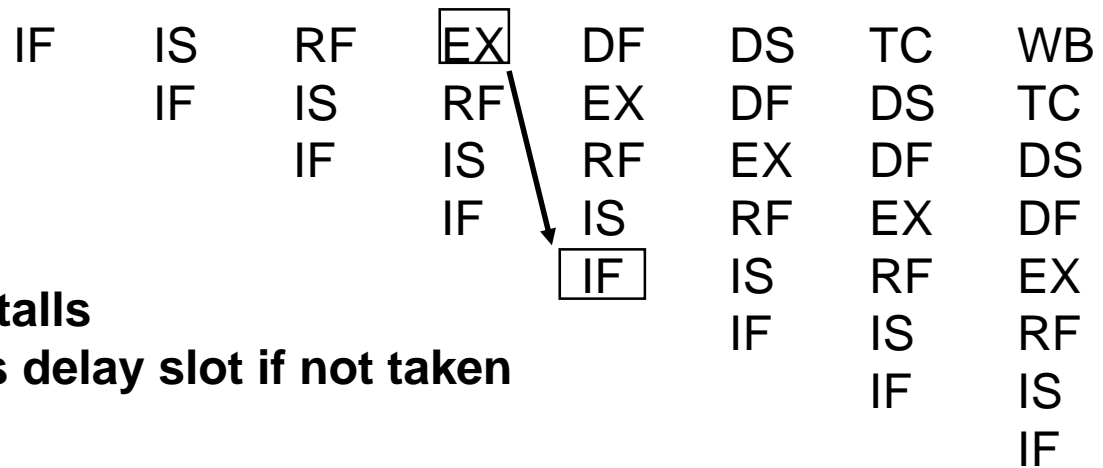


THREE Cycle Branch Latency

(conditions evaluated
during EX phase)

Delay slot plus two stalls

Branch likely cancels delay slot if not taken



MIPS R4000 浮点数操作

- 3个功能部件组成: **FP Adder, FP Multiplier, FP Divider**
- 在乘/除操作的最后一步要 使用**FP Adder**
- **FP操作需要2（negate）-112个（square root）cycles**
- **8 kinds of stages in FP units:**

<i>Stage</i>	<i>Functional unit</i>	<i>Description</i>
A	FP adder	Mantissa ADD stage
D	FP divider	Divide pipeline stage
E	FP multiplier	Exception test stage
M	FP multiplier	First stage of multiplier
N	FP multiplier	Second stage of multiplier
R	FP adder	Rounding stage
S	FP adder	Operand shift stage
U		Unpack FP numbers

MIPS FP 流水段

<i>FP Instr</i>	<i>1</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>	<i>6</i>	<i>7</i>	<i>8</i>	<i>...</i>
Add, Subtract	U	S+A	A+R	R+S					
Multiply	U	E+M	M	M	M	N	N+A	R	
Divide	U	A	R	D ²⁸	...	D+A	D+R, D+R, D+A, D+R, A, R		
Square root	U	E	(A+R) ¹⁰⁸	...	A	R			
Negate	U	S							
Absolute value	U	S							
FP compare	U	A	R						

Stages:

M First stage of multiplier

N Second stage of multiplier

R Rounding stage

S Operand shift stage

U Unpack FP numbers

A Mantissa ADD stage

D Divide pipeline stage

E Exception test stage

FP instruction	Latency	Initiation interval	Pipe stages
Add, subtract	4	3	U,S+A,A+R,R+S
Multiply	8	4	U,E+M,M,M,M,N,N+A,R
Divide	36	35	U,A,R,D ²⁷ ,D+A,D+R,D+A,D+R,A,R
Square root	112	111	U,E,(A+R) ¹⁰⁸ ,A,R
Negate	2	1	U,S
Absolute value	2	1	U,S
FP compare	3	2	U,A,R

双精度浮点操作指令延迟、启动间隔和流水段的使用情况

		Clock cycle												
Operation	Issue/stall	0	1	2	3	4	5	6	7	8	9	10	11	12
Multiply	Issue	U	M	M	M	M	N	N+A	R					
Add	Issue		U	S+A	A+R	R+S								
	Issue			U	S+A	A+R	R+S							
	Issue				U	S+A	A+R	R+S						
	Stall					U	S+A	A+R	R+S					
	Stall						U	S+A	A+R	R+S				
	Issue							U	S+A	A+R	R+S			
	Issue								U	S+A	A+R	R+S		

注： Multiply Issue **U M M -> U E+M M**

		Clock cycle												
Operation	Issue/stall	0	1	2	3	4	5	6	7	8	9	10	11	12
Add	Issue	U	S+A	A+R	R+S									
Multiply	Issue		U	M	M	M	M	N	N+A	R				
	Issue			U	M	M	M	M	N	N+A	R			

注: **Multiply** 的 第 2 拍的M -> E+M

Operation	Issue/stall	Clock cycle											
		25	26	27	28	29	30	31	32	33	34	35	36
Divide	issued in cycle 0...	D	D	D	D	D	D+A	D+R	D+A	D+R	A	R	
Add	Issue		U	S+A	A+R	R+S							
	Issue			U	S+A	A+R	R+S						
	Stall				U	S+A	A+R	R+S					
	Stall					U	S+A	A+R	R+S				
	Stall						U	S+A	A+R	R+S			
	Stall							U	S+A	A+R	R+S		
	Stall								U	S+A	A+R	R+S	
	Stall									U	S+A	A+R	R+S
	Issue										U	S+A	A+R
	Issue											U	S+A
	Issue												U

		Clock cycle												
Operation	Issue/stall	0	1	2	3	4	5	6	7	8	9	10	11	12
Add	Issue	U	S+A	A+R	R+S									
Divide	Stall		U	A	R	D	D	D	D	D	D	D	D	D
	Issue			U	A	R	D	D	D	D	D	D	D	D
	Issue				U	A	R	D	D	D	D	D	D	D

R4000性能（1）

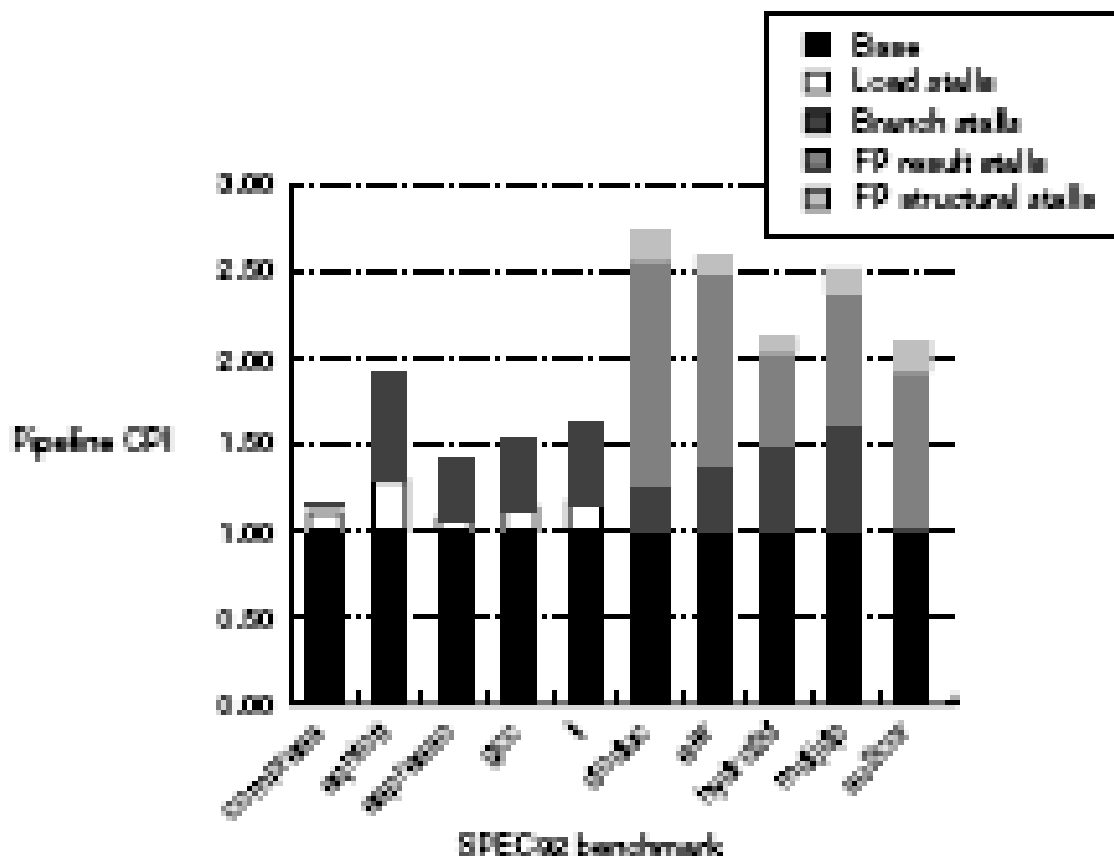


Figure A.48 The pipeline CPI for 10 of the SPEC92 benchmarks, assuming a perfect cache. The pipeline CPI varies from 1.2 to 2.8. The leftmost five programs are integer programs, and branch delays are the major CPI contributor for these. The rightmost five programs are FP, and FP result stalls are the major contributor for these. Figure A.49 shows the numbers used to construct this plot.

R4000 性能 (2)

Benchmark	Pipe CPI	Load Stalls	Branch Stalls	FP Res. Stalls	FP Struc. Stalls
compress	1.20	0.14	0.06	0.00	0.00
eqntott	1.88	0.27	0.61	0.00	0.00
espresso	1.42	0.07	0.35	0.00	0.00
gcc	1.56	0.13	0.43	0.00	0.00
li	1.64	0.18	0.46	0.00	0.00
INTEGER AVERAGE	1.54	0.16	0.39	0.00	0.00
doduc	2.84	0.01	0.22	1.39	0.22
mkldp2	2.66	0.01	0.31	1.20	0.15
ear	2.17	0.00	0.46	0.59	0.12
hydro2d	2.53	0.00	0.62	0.75	0.17
su2cor	2.18	0.02	0.07	0.84	0.26
FP AVERAGE	2.48	0.01	0.33	0.95	0.18
OVERALL AVERAGE	2.00	0.10	0.36	0.46	0.09

基本流水线小结

- 流水线提高的是指令带宽（吞吐率），而不是单条指令的执行速度
- 相关限制了流水线性能的发挥
 - 结构相关：需要更多的硬件资源
 - 数据相关：需要定向，编译器调度
 - 控制相关：尽早检测条件，计算目标地址，延迟转移，预测
- 增加流水线的级数会增加相关产生的可能性
- 异常，浮点运算使得流水线控制更加复杂
- 编译器可降低数据相关和控制相关的开销
 - **Load** 延迟槽
 - **Branch** 延迟槽
 - **Branch**预测

Review

- 流水线技术并不能提高单个任务的执行效率，它可以提高整个系统的吞吐率
 - 多个任务同时执行，但使用不同的资源
- 流水线性能分析：吞吐率、加速比、效率
 - 流水线中的瓶颈——最慢的那一段
 - 其潜在的加速比=流水线的级数
 - 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
- 由于存在相关问题，会导致流水线停顿
 - 结构相关、数据相关和控制相关

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, $CPI = 1$:

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

review 流水线性能分析

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

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

吞吐率、加速比、效率
之间的关系

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

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

流水线技术应用的难度何在？
：相关问题

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

小结: Pipelining

- 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
- 难度何在? 相关问题

Quiz

流水线的成本（**cost**）可以用 **$c+k*h$** 估算，其中 **c** 为所有功能段本身的总成本，**h**为段间锁存器成本，**k**为段数。流水线的性价比可以定义为

$$\text{PCR} = \text{Throughput}/(c+k*h),$$

其中 **Throughput** = $1/t$,

t为 $t_{\text{latch}} + T/k$,

t_{latch} 为锁存器的延迟时间，

T为在非流水线的机器上采用顺序执行方式完成一个任务所花费的总时间。

试推导出使得**PCR**最大化的最优段数 **k_{opt}** 的表达式。

Review- 相关的处理

- 结构相关
 - 概念：由于争用资源而引起的
 - 解决办法
- 数据相关
 - 概念：由于存在实际的通信，而引起的
 - 解决办法：
 - » 硬件：定向技术（**forwarding**）
 - » 软件：指令级调度

Review（续）

- 控制相关
 - 概念：
 - 减少性能损失的基本方法-转移地址，条件码静态处理：
 - » 冻结或排空流水线
 - » 预测分支成功
 - » 预测分支失败
 - » 延迟转移
- 异常
 - 异常的分类
 - 精确中断和非精确中断