

# 流水线技术(2)

# 流水线的控制实现

- ❖ **指令发出(instruction issue)**: 让一条指令从指令译码阶段(ID)移动到执行阶段(EX)的过程。
- ❖ 在 ID 段**检测**所有数据危害
  - ✧ 如果存在危害, 则在**发出**指令之前停止 (插入 “空操作” 指令)。
  - ✧ 同样, 可以确定是否需要 “向前通道”。
- ❖ 也可以在使用操作数的时钟周期之前来检测危害或向前通道
  - ✧ 如 EX 和 MEM 流水段

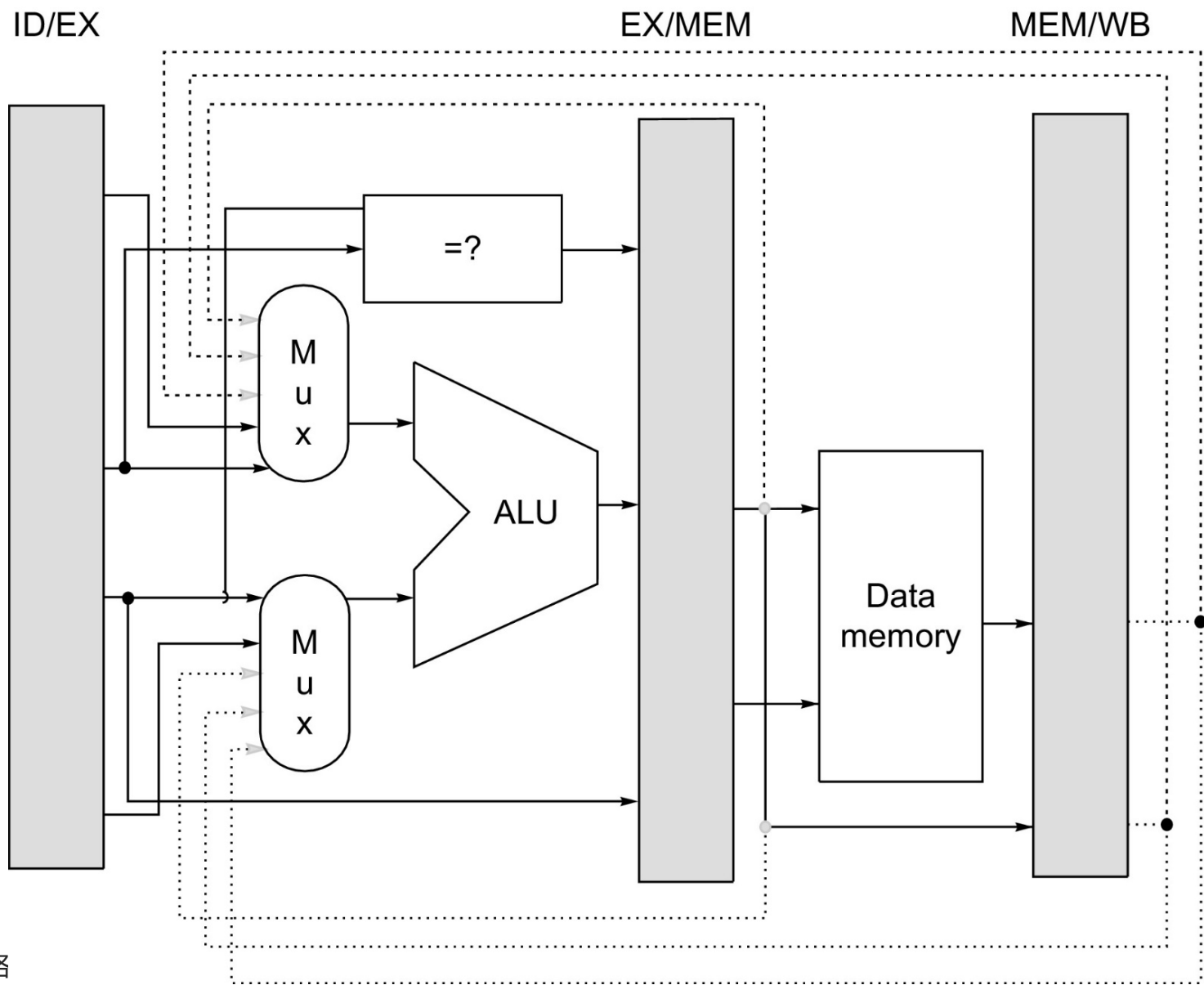
# 流水线危害检测

Situation	Example code sequence		Action
No dependence	ld	x1, 45(x2)	No hazard possible because no dependence exists on x1 in the immediately following three instructions
	add	x5, x6, x7	
	sub	x8, x6, x7	
	or	x9, x6, x7	
Dependence requiring stall	ld	x1, 45(x2)	Comparators detect the use of x1 in the add and stall the add (and sub and or) before the add begins EX
	add	x5, x1, x7	
	sub	x8, x6, x7	
	or	x9, x6, x7	
Dependence overcome by forwarding	ld	x1, 45(x2)	Comparators detect use of x1 in sub and forward result of load to ALU in time for sub to begin EX
	add	x5, x6, x7	
	sub	x8, x1, x7	
	or	x9, x6, x7	
Dependence with accesses in order	ld	x1, 45(x2)	No action required because the read of x1 by or occurs in the second half of the ID phase, while the write of the loaded data occurred in the first half
	add	x5, x6, x7	
	sub	x8, x6, x7	
	or	x9, x1, x7	

# 打开向前通道的比较逻辑

Pipeline register of source instruction	Opcode of source instruction	Pipeline register of destination instruction	Opcode of destination instruction	Destination of the forwarded result	Comparison (if equal then forward)
EX/MEM	Register-register ALU, ALU immediate	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	EX/MEM.IR[rd] == ID/EX.IR[rs1]
EX/MEM	Register-register ALU, ALU immediate	ID/EX	Register-register ALU	Bottom ALU input	EX/MEM.IR[rd] == ID/EX.IR[rs2]
MEM/WB	Register-register ALU, ALU immediate, Load	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	MEM/WB.IR[rd] == ID/EX.IR[rs1]
MEM/WB	Register-register ALU, ALU immediate, Load	ID/EX	Register-register ALU	Bottom ALU input	MEM/WB.IR[rd] == ID/EX.IR[rs2]

# 向前通道的逻辑实现



# 异常

- ❖ 异常(exception): 以意想不到的方式改变指令执行顺序
- ❖ 问题: 指令重叠执行使得难于确定哪条指令可以安全修改处理器状态
- ❖ 术语
  - ✧ 中断——interrupt
  - ✧ 故障——fault
  - ✧ 异常——exception

# 异常类型

- ❖ I/O 设备请求
- ❖ 用户程序调用操作系统服务
- ❖ 跟踪指令执行
- ❖ 断点（程序员请求中断）
- ❖ 整数算术溢出
- ❖ 浮点数算术异常
- ❖ 页面失效（没在主存中）
- ❖ 未对齐内存访问（如果需要对齐）
- ❖ 违反内存保护
- ❖ 使用未定义或未生效的指令

# 异常类型

- ❖ 硬件故障
- ❖ 电源故障





# 异常的特性

❖ 异常的特性将决定硬件需要采取的动作

❖ 同步(synchronous) vs. 异步(asynchronous)

- ✧ 如果每次使用相同的数据和内存分配执行程序时，事件都发生在同一位置，则该事件是同步的。
- ✧ 除了硬件故障，异步是由处理器和内存之外的失败引起的，一般可以在当前指令完成之后再处理。

❖ 用户请求的 vs. 强制的

- ✧ 如果用户任务直接要求，称为用户请求事件。有时是可预测的，总是可以在指令完成后处理。
- ✧ 强制事件是由硬件事件引起的。因为是不可预测的，所以难于处理。

❖ 用户可屏蔽的 vs. 用户不可屏蔽的

- ✧ 屏蔽(mask)用来控制硬件是否响应中断

# 异常的特性

## ❖ 指令内部 vs. 指令之间

- ✧ 取决于事件是否发生在指令执行过程中来阻止指令完成，或者是否发生在指令之间。
- ✧ 发生在指令内部的异常比发生在指令之间的异常更难实现，因为指令必须停止和重新启动。

## ❖ 重新开始 vs. 终止

- ✧ 如果程序总是在中断之后停止运行，则称为**终止(terminating)**事件。
- ✧ 如果程序在中断之后继续运行，则称为**重新开始(resuming)**事件。

# 异常的特性与类型

Exception type	Synchronous vs. asynchronous	User request vs. coerced	User maskable vs. nonmaskable	Within vs. between instructions	Resume vs. terminate
I/O device request	Asynchronous	Coerced	Nonmaskable	Between	Resume
Invoke operating system	Synchronous	User request	Nonmaskable	Between	Resume
Tracing instruction execution	Synchronous	User request	User maskable	Between	Resume
Breakpoint	Synchronous	User request	User maskable	Between	Resume
Integer arithmetic overflow	Synchronous	Coerced	User maskable	Within	Resume
Floating-point arithmetic overflow or underflow	Synchronous	Coerced	User maskable	Within	Resume
Page fault	Synchronous	Coerced	Nonmaskable	Within	Resume
Misaligned memory accesses	Synchronous	Coerced	User maskable	Within	Resume
Memory protection violations	Synchronous	Coerced	Nonmaskable	Within	Resume
Using undefined instructions	Synchronous	Coerced	Nonmaskable	Within	Terminate
Hardware malfunctions	Asynchronous	Coerced	Nonmaskable	Within	Terminate
Power failure	Asynchronous	Coerced	Nonmaskable	Within	Terminate

# 停止运行和重新开始

- ❖ **定义：**流水线或处理器是**可重新启动的(restartable)**
  - ✧ 流水线具备处理中断、保留状态且在不影响程序执行结果的情况下重新开始执行的能力。
- ❖ **最困难异常具有两个特性**
  - ✧ 在指令执行期间发生中断，如在 EX 或 MEM 流水段；
  - ✧ 指令可重新启动（通过保留重新启动指令的 PC 值）
    - ✧ 非分支转移指令：按正常方式执行顺序读入的后续指令
    - ✧ 分支转移指令：重新计算转移条件，然后从确定的地址取入指令运行。
- ❖ **安全保存流水线状态的步骤**
  - ✧ 在下一个 IF 周期，插入陷阱(trap)指令；
  - ✧ 在陷阱指令执行之前，关闭错误指令及其后指令的所有写操作；
  - ✧ 操作系统的中断处理程序接管控制后，马上保存错误指令的 PC 值。
- ❖ **异常处理完毕，重新加载 PC 和重新启动指令流从异常返回**

# 精确中断

## ❖ 定义：精确中断(precise exception)

- ❑ 如果能够停止流水线工作，使得在错误指令之前的指令执行完毕；而在错误指令之后的指令可以从中断位置重新开始执行，则称流水线是精确中断的。

## ❖ 精确中断不是“刚需”

- ❑ 在指令乱序完成 (complete out of order) 情况下，做到精确中断需要更多的逻辑电路。
- ❑ 具备按需分页或者 IEEE 算术陷阱处理程序的处理器必须是精确中断的。

## ❖ 近期高性能处理器引入两种操作模式

- ❑ 精确中断模式：由于只允许少量的浮点运算重叠执行，故速度较慢。
- ❑ 快速或性能模式：非精确中断

# RISC V 中断

## ❖ RISC V 流水段和每段可能发生中断的问题

✧ 由于在同一个时钟周期中有多条指令同时运行，故可能出现多个中断。

ld	IF	ID	EX	MEM	WB	
add		IF	ID	EX	MEM	WB

Pipeline stage	Problem exceptions occurring
IF	Page fault on instruction fetch; misaligned memory access; memory protection violation
ID	Undefined or illegal opcode
EX	Arithmetic exception
MEM	Page fault on data fetch; misaligned memory access; memory protection violation
WB	None

# RISC V 中断

- ❖ 因为中断可能不是按照指令顺序发生的，所以流水线不能对发生的中断进行简单地（按发生次序）处理。
  - ✧ 后续指令可能在先前指令引起中断之前引发中断
    - ✧ Id 指令在 MEM 段发生页面失效，而 add 指令在 IF 段发生页面失效。
- ❖ 中断状态矢量(exception status vector)
  - ✧ 硬件将由该指令引起的所有中断发送到与该指令相关的状态矢量中。
  - ✧ 中断状态矢量随指令一起进入流水线。
  - ✧ 中断状态矢量中的中断标志一旦置位，则关闭所有可能引发写入数据操作（包括写寄存器和写内存）的控制信号。
  - ✧ 当指令进入 WB 流水段（或离开 MEM 流水段），检查中断状态矢量。
    - ✧ 如果发生中断，则按照其在非流水线处理器中的执行顺序来处理。



# 指令集复杂性

- ❖ **定义**：当指令保证完成时，称为**提交(committed)**。
  - ✧ 指令到达 MEM 段尾部时被提交，在此之前指令不会修改处理器状态
  - ✧ 如此，精确中断就是简单的。
- ❖ 如果在指令及其前任保证完成之前的执行过程中修改处理器状态，则必须通过添加相应的硬件装置，保证在发生中断时能够恢复任何已被修改的状态。
  - ✧ IA-32架构中的自动增量寻址方式
- ❖ **条件码(condition code)**
  - ✧ 做为指令的一部分（隐式地设置条件码）
  - ✧ **优点**：将条件求值与实际分支解耦
  - ✧ **缺点**：
    - ✧ 导致在设置条件码和分支之间调度流水线延迟的困难（不能用于延迟槽）
    - ✧ 确定何时是最后一次设置条件码（推迟到所有早期指令设置条件码后）



# 指令集复杂性

## ❖ 微指令(microinstruction)

- ❑ 微指令是在序列中使用的一种简单指令，用于实现更复杂的指令集。
- ❑ IA-32 和 ARM 都使用这种方法实现更复杂的指令（克服由于不同指令需要不同数目时钟周期带来的各种困难）
- ❑ 对比来看，装入-存储处理器具有简单操作且流水线更容易实现。

## ❖ 如果架构师认识到指令集设计与流水操作之间的关系，则可以设计具备更高效流水操作的体系结构。

## ❖ 20世纪90年代，所有公司都转向了更简单的指令集，以降低实现的复杂性。

# 多周期操作

❖ 浮点数操作很难在 1 ~ 2 时钟周期内完成

## ❖ 解决办法

- ❑ 接受慢时钟 和/或 在浮点数功能单元中添加大量逻辑电路
- ❑ 浮点数流水线允许较长的操作延时

## ❖ 相比整数流水线的两点变化

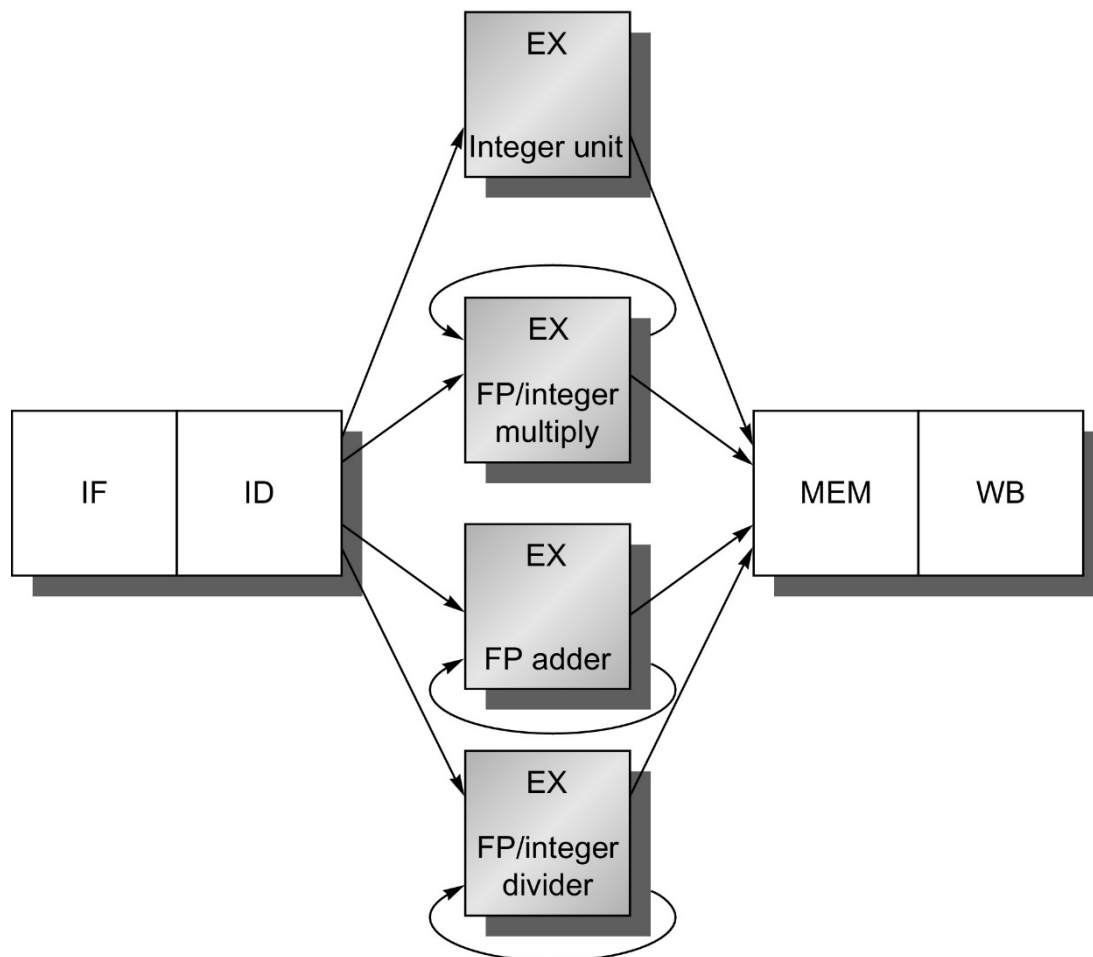
- ❑ EX 周期将根据所完成操作的需要重复多次，且重复次数是不固定的；
- ❑ 存在多个浮点数(FP)功能单元

## ❖ 存在问题

- ❑ 使用功能单元带来的结构危害
- ❑ 数据危害

# 多周期操作

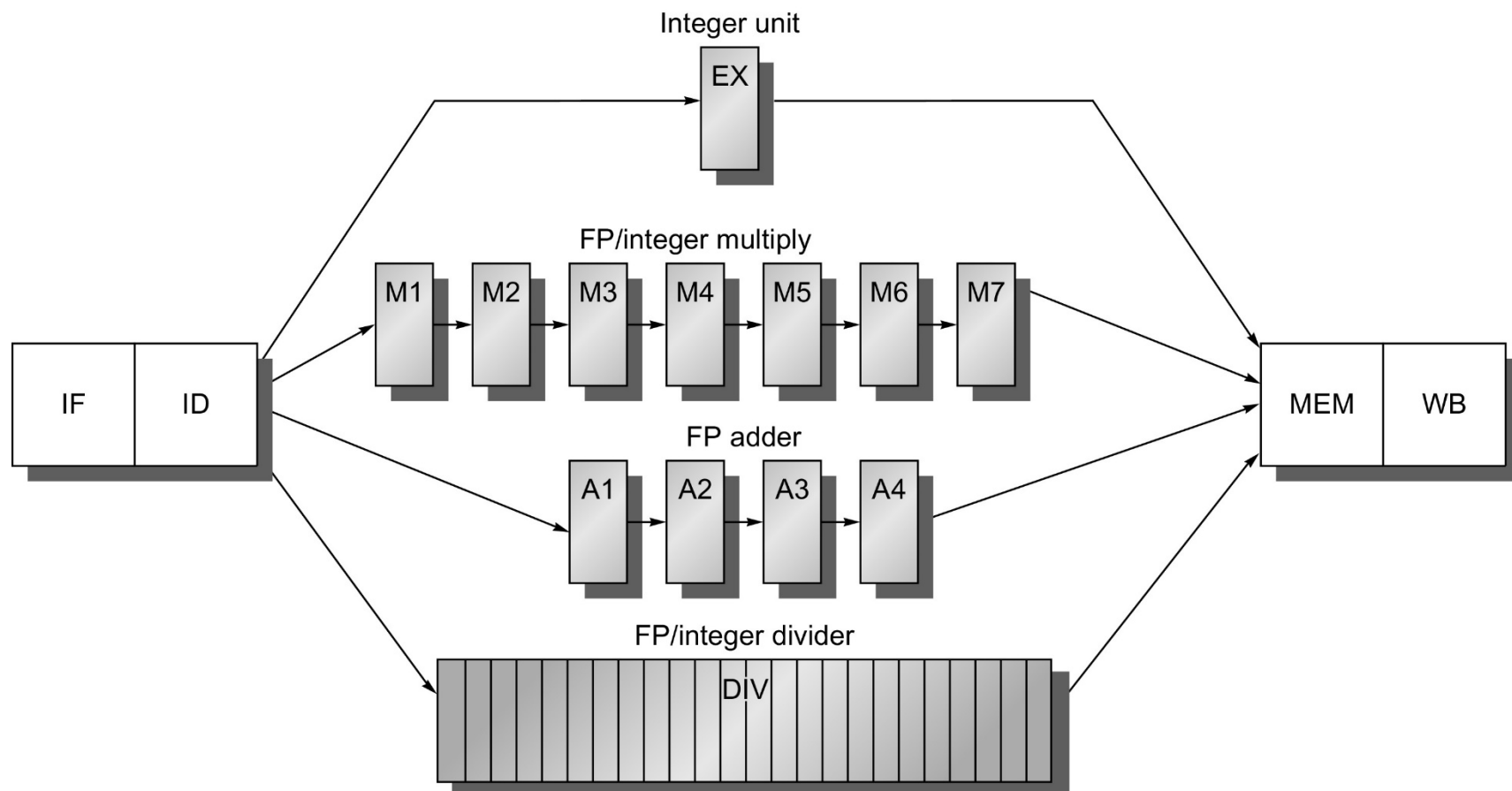
❖ RISC V 整数流水线附加三个非流水结构的浮点数功能单元



✧ “一夫当关”：由于功能单元采用非流水结构，ID 段成为性能瓶颈。

# 多周期操作

## ❖ 支持多浮点数操作的流水线



✧ 浮点数除法单元一般是非流水的。

- ❖ **延迟(latency)**: 生成结果的指令和使用结果的指令之间所需要的周期数
  - ✧ 通常是指令在执行 EX 后产生结果的流水段数
- ❖ **初始间隔(initiation interval)或重复间隔(repeat interval)**: 执行指定类型的两个操作之间必须经过的周期数
- ❖ RISC V 功能单元的**延迟**和**初始间隔**

Functional unit	Latency	Initiation interval
Integer ALU	0	1
Data memory (integer and FP loads)	1	1
FP add	3	1
FP multiply (also integer multiply)	6	1
FP divide (also integer divide)	24	25

# 多周期操作

- ❖ 对较高时钟频率的惩罚是操作的更长延迟
  - ✧ 为了获得较高的时钟频率，设计者需要在每个流水段设置较少的逻辑层，这将导致完成更加复杂操作所需的流水段数更多。

## ❖ 独立浮点数操作的流水线定时图

fmul.d	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
fadd.d		IF	ID	A1	A2	A3	A4	MEM	WB		
fld			IF	ID	EX	MEM	WB				
fsd				IF	ID	EX	MEM	WB			

- ❖ 浮点数操作的较长延迟增加了发生 RAW 危害和由此产生停顿的频度。

# 危害和向前通道

## ❖ 检测危害和向前通道

- ❑ 检测除法单元的结构危害。一旦发生，则停止发出指令。
- ❑ 由于指令的执行时间不同，一个周期内寄存器的写次数可能大于 1。
- ❑ 由于指令不再按序抵达 WB 流水段，故可能发生 WAW 危害。
- ❑ 指令完成的顺序可能与发出顺序不同，可能导致中断处理出现问题。
- ❑ 由于指令更长的延迟，RAW 危害造成的停顿将更加频繁。

## ❖ 由于 RAW 相关引起的停顿

Instruction	Clock cycle number																
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
f1d f4,0(x2)	IF	ID	EX	MEM	WB												
fmul.d f0,f4,f6		IF	ID	Stall	M1	M2	M3	M4	M5	M6	M7	MEM	WB				
fadd.d f2,f0,f8			IF	Stall	ID	Stall	Stall	Stall	Stall	Stall	Stall	A1	A2	A3	A4	MEM	WB
fsd f2,0(x2)					IF	Stall	Stall	Stall	Stall	Stall	Stall	ID	EX	Stall	Stall	Stall	MEM

# 写操作带来的结构危害

Instruction	Clock cycle number										11
	1	2	3	4	5	6	7	8	9	10	
fmul.d f0,f4,f6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
...		IF	ID	EX	MEM	WB					
...			IF	ID	EX	MEM	WB				
fadd.d f2,f4,f6				IF	ID	A1	A2	A3	A4	MEM	WB
...					IF	ID	EX	MEM	WB		
...						IF	ID	EX	MEM	WB	
fld f2,0(x2)							IF	ID	EX	MEM	WB

## ❖ 解决方法

- ❑ 增加写端口：发生“同时写”情形的频度较小——不值得
- ❑ 检测且强制访问写端口：采用互锁(interlock)技术


## ❖ 互锁(interlock)

- ❑ 方法一：使用移位寄存器在 ID 段检测写端口使用，使用停顿来解决。
- ❑ 方法二：阻止冲突指令进入 MEM 段或 WB 段。但可能造成停顿“回流”到 EX 段。



# WAW 危害

Instruction	Clock cycle number										
	1	2	3	4	5	6	7	8	9	10	11
fmul.d f0,f4,f6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
...		IF	ID	EX	MEM	WB					
...			IF	ID	EX	MEM	WB				
fadd.d f2,f4,f6				IF	ID	A1	A2	A3	A4	MEM	WB
...					IF	ID	EX	MEM	WB		
...						IF	ID	EX	MEM	WB	
fld f2,0(x2)							IF	ID	EX	MEM	WB



- ❖ 如果二者之间存在读取 `f2` 的指令，则该指令就会阻止 `fld` 的发出。
  - ✧ 方法一：推迟 `fld` 指令发出，直到 `fadd.d` 进入 MEM 段。
  - ✧ 方法二：通过检测危害和改变控制，使 `fadd.d` 不再写寄存器 `f2`。
- ❖ 简单通用方法
  - ✧ 如果在 ID 段指令要写入的寄存器与已发出指令写入的寄存器相同，则不让该指令进入到 EX 段。

# ID 段的危害检测

- ❖ **危害检测**：必须在指令发出之前完成三种检测。
- ❖ **结构危害检测**：等待资源空闲
- ❖ **RAW 数据危害检测**：等待直到源寄存器不在流水线寄存器的挂起目标寄存器列表中。
  - ✧ 挂起目标寄存器：当该指令需要结果时，目标寄存器不可用。
  - ✧ 例：假如 ID 段中的指令是一条以  $f2$  为源寄存器的 FP 操作，那么  $f2$  不能在 ID/A1、A1/A2 或 A2/A3 的目标寄存器列表中，这对应着 FP 加法指令，当 ID 段中的指令需要一个结果时，该指令还没有完成。
- ❖ **WAW 数据危害检测**：确定处于 A1, ..., A4, D, M1, ..., M7 中的任何指令是否与该指令具有相同的目标寄存器。

# 向前逻辑

- ❖ 与整数流水线具有相同的处理方式
- ❖ **检测**：是否在任何 EX/MEM、A4/MEM、M7/MEM、D/MEM 或 MEM/WB 寄存器中的目标寄存器是浮点数指令的源寄存器之一。
- ❖ **实现**：如果检测为真，则启用(enable)对应的输入多路复用器(multiplexer)，选择需要转发的数据，即向前通道的数据源。

# 维持精确中断

## ❖ 代码段中的指令间没有相关性

```
fdiv.d      f0, f2, f4  
fadd.d      f10, f10, f8  
fsub.d      f12, f12, f14
```

## ❖ 问题情形：早期发出的指令可能在稍后发出的指令之后完成

✧ 乱序完成(out-of-order completion)：如果 `fadd.d` 和 `fsub.d` 早于 `fdiv.d` 完成——具有长运行时间指令流水线的常态

## ❖ 不精确中断(imprecise exception)

✧ 假设 `fsub.d` 指令引发浮点数算术中断时，`fadd.d` 指令已经完成，而 `fdiv.d` 指令还在运行。

## ❖ 原因：指令完成的顺序与指令发出的顺序是不同的，即乱序完成。

# 维持精确中断的方法

- ❖ 忽略问题且不解决不精确中断（1960年~1970年使用）
- ❖ 缓存操作结果，直到所有已经发出指令完成（缓存器很大  $\Rightarrow$  成本高）
  - ✧ 利用历史文件(history file)存储旧的寄存器数值，当需要时回滚。
  - ✧ 利用将来文件(future file)存储新的寄存器数值，当所有早期指令完成后，再刷新寄存器文件。
- ❖ 允许存在一些不精确中断，但保存足够的信息，使得中断处理程序能够建立一个精确的序列
  - ✧ Instruction<sub>1</sub> – 长时间运行指令，发生中断
  - ✧ Instruction<sub>2</sub>, ..., Instruction<sub>n-1</sub> – 一串没有完成的指令
  - ✧ Instruction<sub>n</sub> – 已经完成指令
  - ✧ 因为 Instruction<sub>n</sub> 已经完成，所以重新开始的指令应该是 Instruction<sub>n+1</sub>。
  - ✧ 处理中断之后，软件必须仿真执行指令 Instruction<sub>1</sub> ~ Instruction<sub>n-1</sub>。

# 维持精确中断的方法

- ❖ 只有当确定发出指令之前的所有指令都将完成而不会引起中断时，才允许指令继续发出。
  - ✧ 保证当某条指令发生中断时，该指令后的指令不会完成，而该指令前的所有指令都可以完成。
  - ✧ 有时需要停顿处理器来维持精确中断

# 简单流水线的性能损失

- ❖ 取指令与发出指令会受到指令间数据相关性(data dependence)的影响
- ❖ 向前逻辑 (forwarding logic) 通过减少流水线延迟, 使得某些相关性 (dependence) 不会导致危害 (hazard)。
- ❖ 如果存在不可避免的危害 (hazard), 则流水线停顿, 直到相关性消除。
- ❖ 解决方法:
  - ✧ 静态调度(static scheduling): 编译器试图调度指令以防止危害
  - ✧ 动态调度(dynamic scheduling): 硬件重新安排指令以减少相关



# 动态调度

- ❖ **按序发出 (in-order issue)**: 如果一条指令在流水线中停顿, 则后续指令不会继续执行。
- ❖ 早期流水线在指令译码(ID)期间检测**结构危害**和**数据危害**
  - ✧ 当指令可以正常执行时, 才会从 ID 段发出;
  - ✧ 当两条指令间存在危害时, 流水线停顿, 不管后面指令是否存在相关性和停顿。
- ❖ **乱序执行(out-of-order execution)**: 指令在其操作数一旦有效时可以马上开始执行 (未考虑顺序)
  - ✧ **乱序执行**暗示着**乱序完成(out-of-order completion)**
- ❖ 为了实现乱序执行, **ID 流水段**将被 “**一分为二**” :
  - ✧ **发出(issue)**: 解码指令, 检查**结构危害**。
  - ✧ **读操作数(read operands)**: 等待直到没有**数据危害**再读操作数。



- ❖ 指令开始执行 (begin execution): 以时钟周期为单位
- ❖ 指令完成执行 (complete execution): 以时钟周期为单位
- ❖ 指令在执行(in execution): 处于上述两个时刻之间的状态

# 计分板

- ❖ 在动态调度流水线中，所有指令
  - ✧ 按序通过发出流水段(in-order issue)——按序发出
  - ✧ 在读操作数流水段指令可能停顿或相互绕行，然后进入乱序执行
- ❖ 计分板(scoreboarding)：当存在足够资源且没有数据相关性时，允许指令乱序执行的一种技术。

## ❖ 代码段示例

```
fdiv.d      f0, f2, f4  
fadd.d      f10, f0, f8  
fsub.d      f8, f8, f14
```

- ❏ 如果 `fsub.d` 先于 `fadd.d` 执行，则 `fadd.d` 和 `fsub.d` 之间可能存在 **WAR 危害**。
- ❏ 如果 `fsub.d` 的目标寄存器为 `f10`，则两条指令之间存在 **WAW 危害**。

- ❖ **目标**：当不存在结构危害时，尽可能早地执行指令，以维持每个时钟周期一条指令的执行速率。
- ❖ **任务**：完全负责指令发出和执行，包括所有危害检测。

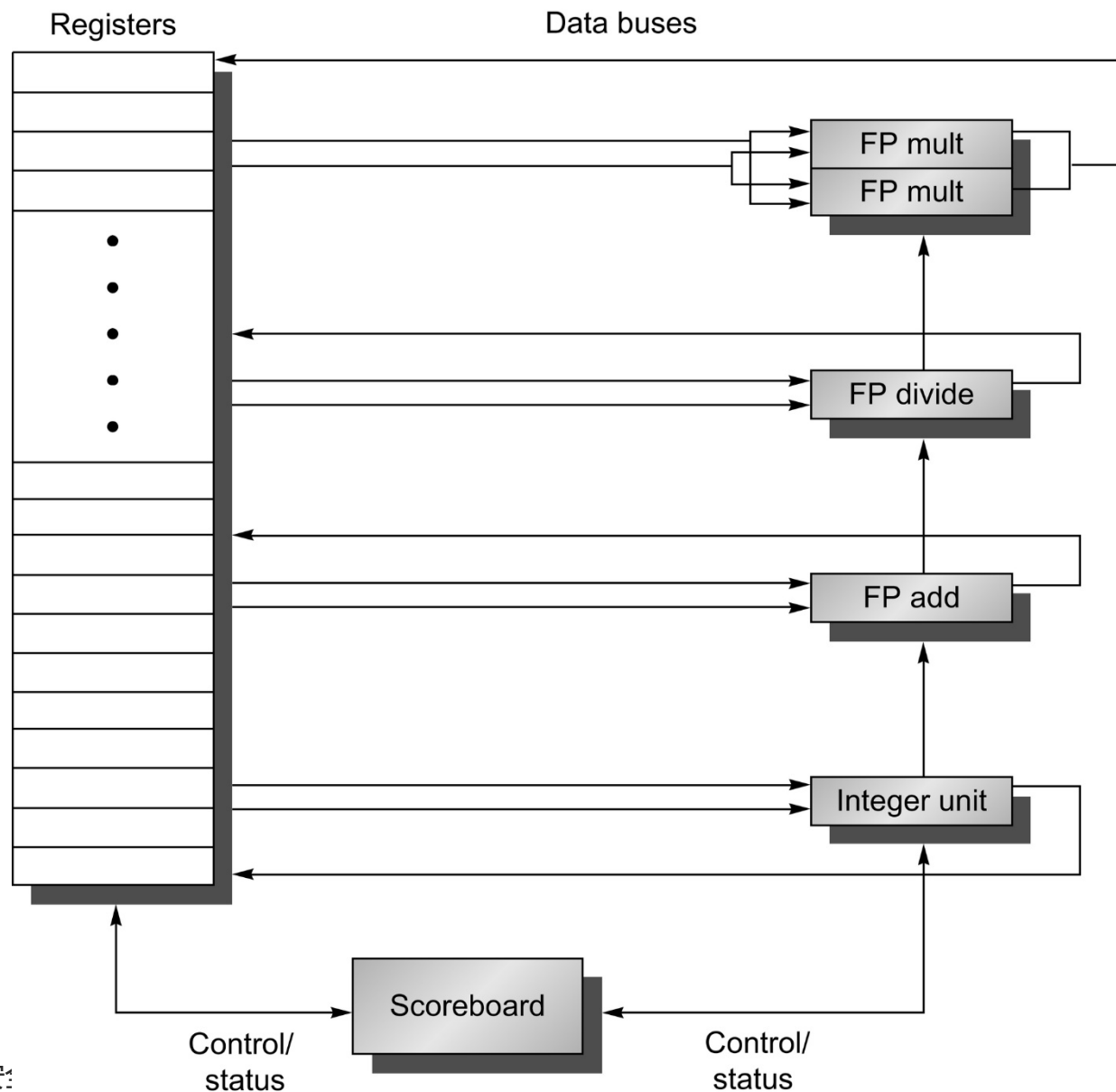
## ❖ CDC 6600 包括16个功能单元

- ❑ 4 个浮点数单元
- ❑ 5 个存储器访问单元
- ❑ 7 个整数操作单元

## ❖ 简化处理器模型

- ❑ 两个乘法器
- ❑ 一个加法器
- ❑ 一个除法器
- ❑ 一个整数单元：完成所有访存，分支转移，整数操作

# 带有记分板 RISC V 处理器的基本结构



# 计分板的主要功能

- ❖ 记录数据相关性，对应着指令发出，替换 ID 段的部分功能。
- ❖ 决定指令何时能够读取操作数并开始执行
  - ✧ 对不能立即执行的指令，监视硬件的每个变化，并决定指令何时执行。
- ❖ 控制指令何时可以将其结果写入目标寄存器
- ❖ 所有危害检测 and 解决都集中在计分板上
- ❖ 每条指令的执行经过四个步骤
  - ✧ 主要关注 FP 操作，未考虑访问内容步骤
  - ✧ 这四个步骤代替标准 RISC V 流水线的 ID、EX 和 WB 步骤

# 第一步骤：发出(issue)

## ❖ 功能：

- ❑ 如果 执行指令的功能单元空闲（无结构危害），且不与其它正在执行指令有相同的目标寄存器（无 WAW 危害），
- ❑ 则 计分板将指令发出到功能单元且更新内部数据结构；
- ❑ 否则 指令停止发出，直到清除这些危害；指令放入取指与发出之间的缓存器，如果该缓存器满，则停止取指操作。

## ❖ 作用：代替标准流水线中 ID 流水段的部分功能

## 第二步骤：读操作数(read operands)

- ❖ **功能：**当源操作数有效时，通知功能单元从寄存器中读操作数，开始执行指令。
- ❖ **作用：**
  - ✧ 动态地解决 RAW 危害
  - ✧ 乱序执行
  - ✧ 与 Issue 步骤共同完成标准流水线中 ID 流水段的功能



## 第三步骤：执行(execution)

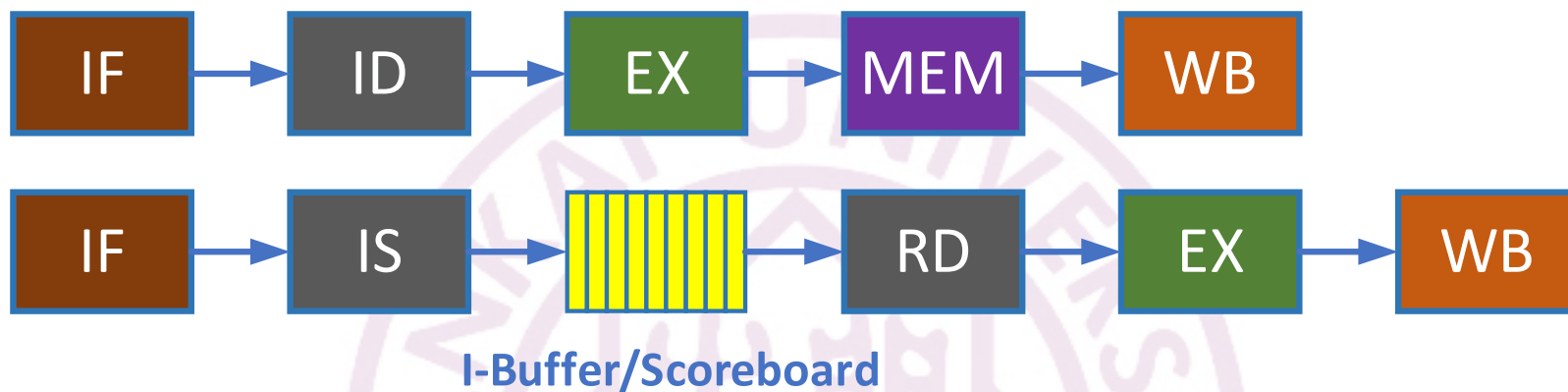
- ❖ **功能**：功能单元开始执行指令；当执行结果有效时，通知计分板指令执行完毕。
- ❖ **作用**：代替标准流水线中的 EX 流水段，且需要多个周期。

## 第四步骤：写结果(write result)

- ❖ **功能**：一旦计分板发现功能单元执行完毕，检查 WAR 危害；如果存在危害，则**停顿**正在完成的指令；否则通知功能单元将结果写入目标寄存器。
  - ✧ 正在完成的指令不允许写结果，如果
    - ✧ 先于正在完成指令的某条指令还没有读取操作数，**且**
    - ✧ 操作数之一与正在完成指令的结果使用相同的寄存器
- ❖ **作用**：代替标准流水线中的 WB 流水段。

# 计分板

## ❖ 带有计分板流水线与简单流水线的差异



# 讨论题

- ❖ 在计分板方法中，只有当指令的操作数都有效时才会从寄存器文件中读取操作数，而没有利用向前通道方法的优势。试分析这两种方法的利弊。
  - ✧ 获取操作数的时间惩罚并没有想象中那么大，因为如果不存在 WAR 危害，指令在完成执行后会尽快将结果写入寄存器文件；而带有向前通道方法的简单流水线，则需要等待可能几个周期之后静态分配的写槽（如 WB 段）
  - ✧ 计分板需要额外的缓存来解决写结果和读操作数不同重叠的问题
  - ✧ 计分板控制指令执行进程是通过与功能单元通信完成的，但与寄存器文件之间的源操作数总线和结果总线的数目是有限的，会造成结构危害。
    - ✧ CDC 6600 的解决方案是将功能单元分组，每组分配一个数据主干(data trunks)。每组中的所有功能单元分时共享该数据主干。

# 诚信 创新 实践

