计算机科学与技术专业

计算机组成

流水线及其冒险

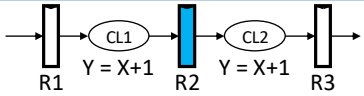
高小鹏

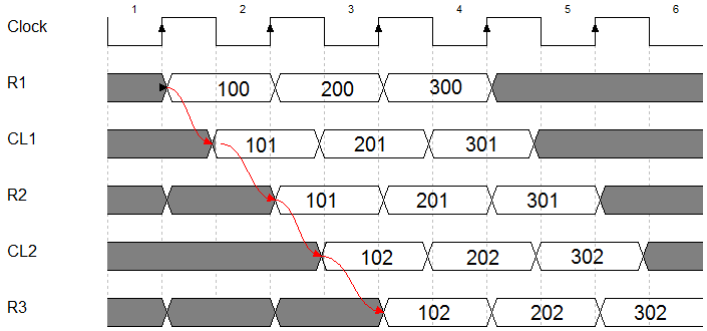
北京航空航天大学计算机学院

目录

示例：简单的流水线电路

- 以下是R1在3个连续时钟周期分别输入100、200、300的电路时序图

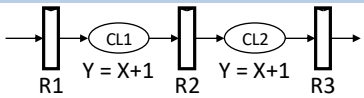




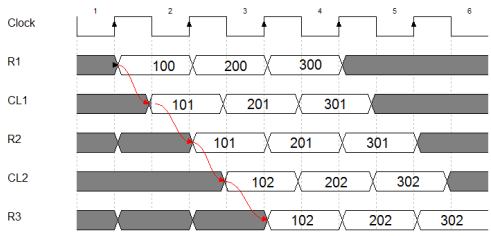
3

流水线工作过程的形式表示

- 为清晰的表示流水线的工作过程，采用表格化的描述方式



- 水平方向：按流水线执行顺序布局
  - 由于寄存器值的保存与传递是分析要点，以及简化分析过程，不单独记录组合逻辑计算结果
- 垂直方向：寄存器在相应时钟上升沿后的值



	CL1	CL2	
CLK	R1	R2	R3
↑ 1	XXX→100		
↑ 2	100→200	101	
↑ 3	100→200	201	102
↑ 4		301	202
↑ 5			302

4

回顾：MIPS数据通路的5个阶段

- 1)取指令：IF，Instruction Fetch
  - 从IM中读取指令，并更新PC
- 2)译码：ID，Instruction Decode
  - 指令驱动RF读取寄存器值(包括立即数扩展)
  - 控制器对指令的op和funct进行译码
- 3)执行：EX，EXecution
  - ALU计算（load/store计算地址；其他指令则为计算）
- 4)访存：MEM，MEMory
  - 读(或写)存储器
- 5)回写：WB，Write Back
  - 将ALU结果或存储器读出的数据写回寄存器堆

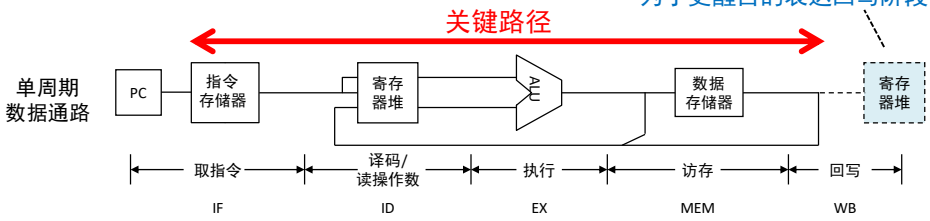
单周期及多周期的特点

- 单周期
  - 所有指令的执行周期数都为1，即CPI=1　CPI：Cycle Per Instruction
  - 存在一条非常明显的关键路径，CPU的时钟频率很低
  - 结果导致每条指令的执行时间都等最慢的那条指令的执行时间
- 多周期
  - 通过插入寄存器，解决了单周期时钟频率低的的不足
  - 数据通路每次只执行一条指令，功能部件利用率低
    - 示例：add进行到执行阶段时，IM和RF都处于空闲状态

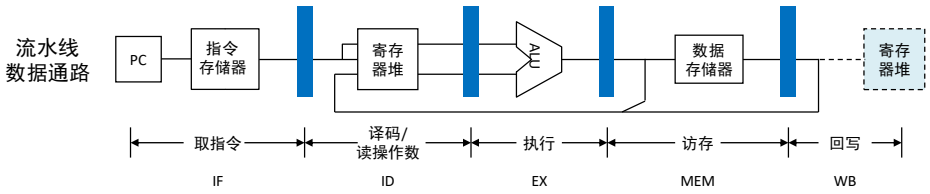
流水线架构模型

- 单周期数据通路架构模型

※RF再次出现的目的在于
为了更醒目的表达回写阶段



- 流水线架构模型类似于多周期，通过插入多个寄存器将单周期的关键路径物理上切割开



流水线执行特点

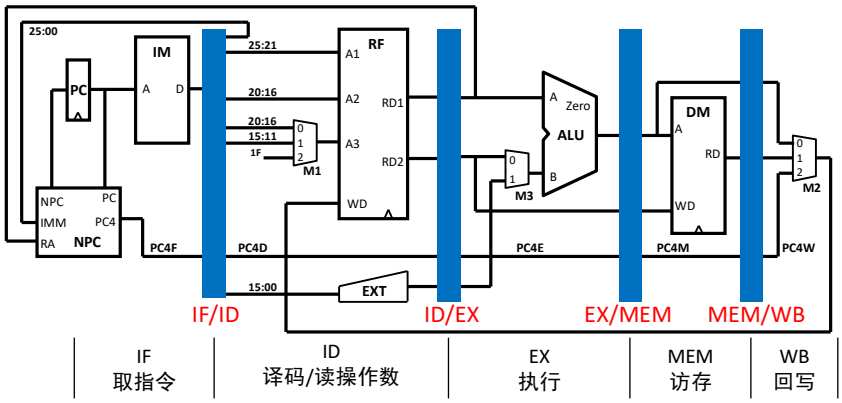
- 理论上，流水线每个cycle都可以从IM中读取一条新指令，同时将已在流水线里的指令向前推进一个阶段
  - 多条指令在数据通路中，但同一时刻占用不同的资源（即处于不同阶段）

- 对于N级流水线，单条指令执行周期数均为N
  - 少数控制流指令除外（如分支类指令、跳转类指令）
- 理论上，当流水线充满后，每个cycle可以执行完一条指令

目录

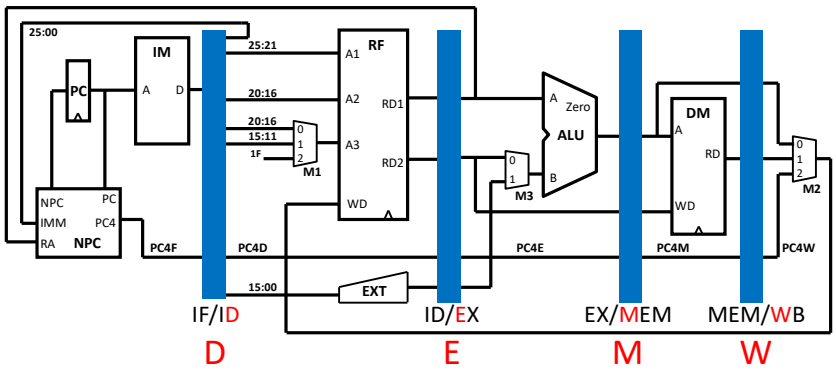
细化流水线数据通路

- 流水线寄存器命名方法1：采用相邻两级名字组合的方式
- 示例：第一级寄存器被命名为IF/ID
  - 类似的，其他各级寄存器为ID/EX、EX/MEM、MEM/WB



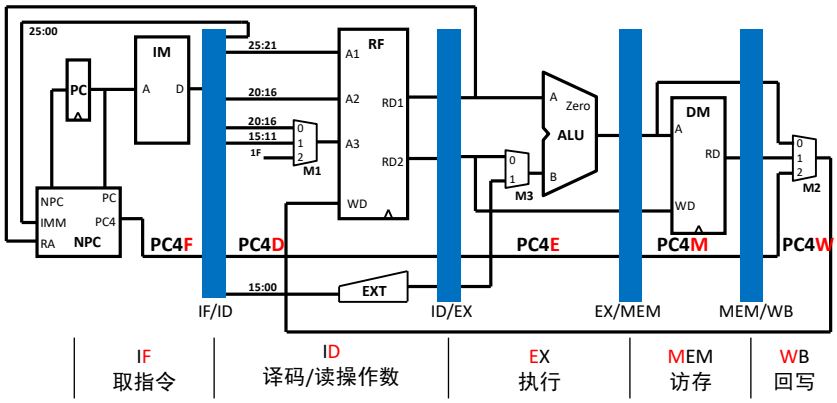
细化流水线数据通路

- 流水线寄存器命名方法2：用各级功能的某个字母命名
  - 方法2更加简洁些
- 示例：第一级流水线寄存器被命名为D（ID）



细化流水线数据通路

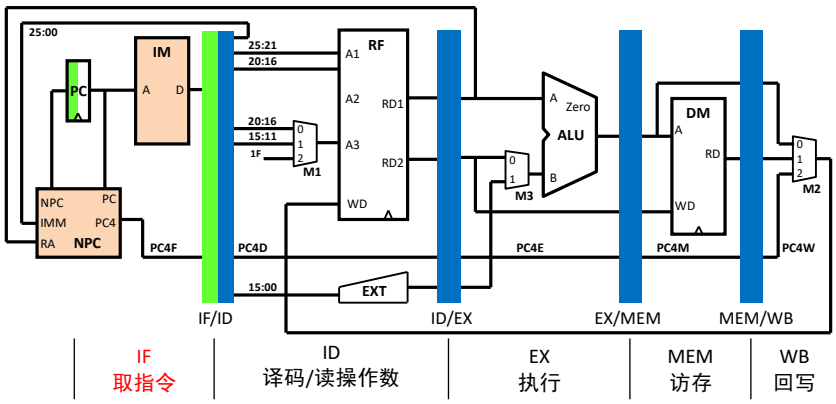
- 信号命名：末尾添加特定字母以区分
- 示例：NPC输出的PC+4
  - IF阶段命名为PC4F
  - 类似的，ID阶段、EX阶段、MEM阶段和WB阶段，分别为PC4D、PC4E、PC4M和PC4W



示例：lw执行过程1/5

- 第1个时钟周期
  - PC驱动IM读取指令，指令写入IF/ID
  - PC驱动NPC计算，将PC+4写入PC

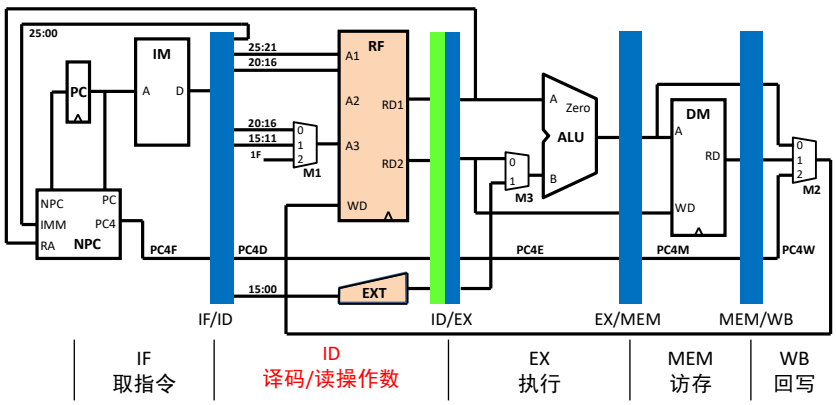
寄存器被写入
组合逻辑正在工作



示例：lw执行过程2/5

- 第2个时钟周期
  - IF/ID驱动RF读取操作数，结果写入ID/EX
  - IF/ID驱动EXT计算扩展的立即数，结果写入写入ID/EX

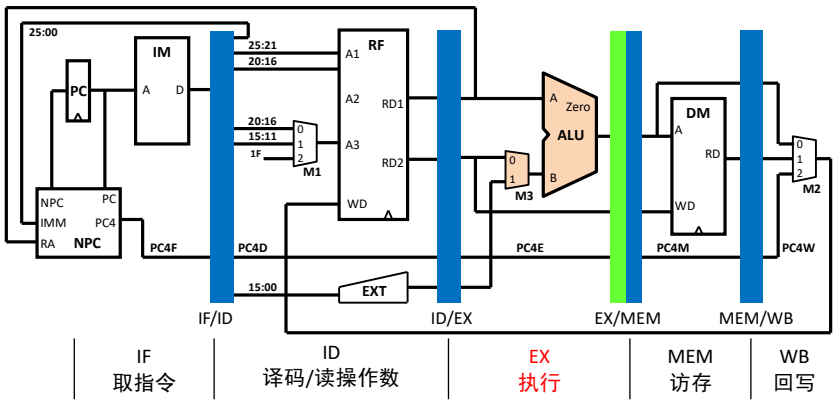
寄存器被写入
组合逻辑正在工作



示例：lw执行过程3/5

- 第3个时钟周期
  - ID/EX驱动ALU计算，结果写入EX/MEM

寄存器被写入
组合逻辑正在工作

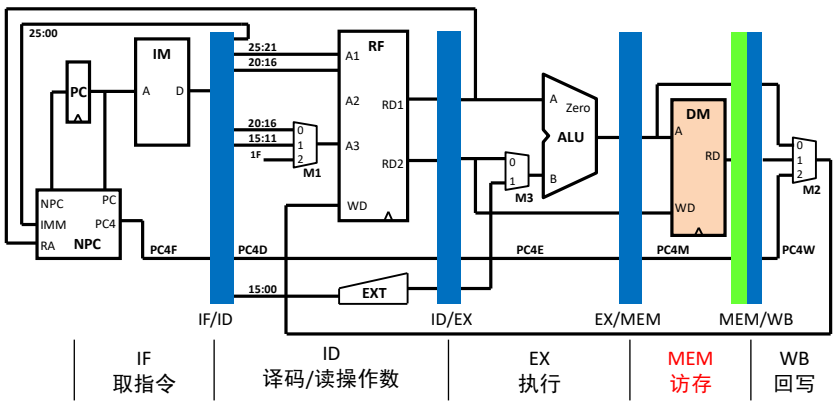


IF 取指令	ID 译码/读操作数	EX 执行	MEM 访存	WB 回写

示例：lw执行过程4/5

- 第4个时钟周期
  - EX/MEM驱动DM读取数据，结果写入MEM/WB

寄存器被写入
组合逻辑正在工作

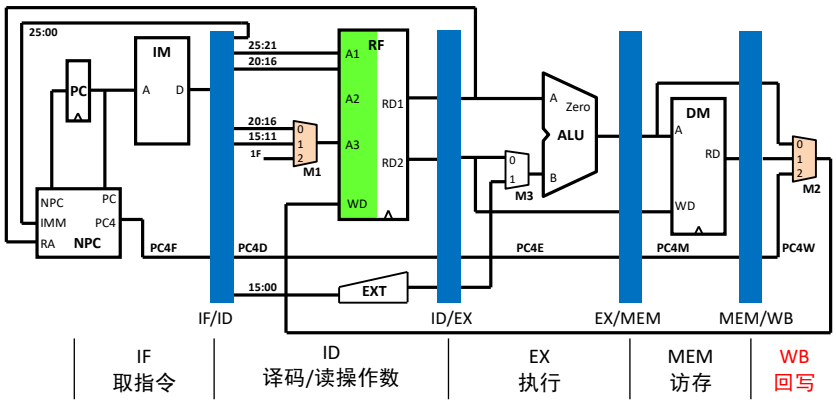


IF 取指令	ID 译码/读操作数	EX 执行	MEM 访存	WB 回写

示例：lw执行过程5/5

- 第5个时钟周期
  - MEM/WB将结果写入RF

寄存器被写入
组合逻辑正在工作

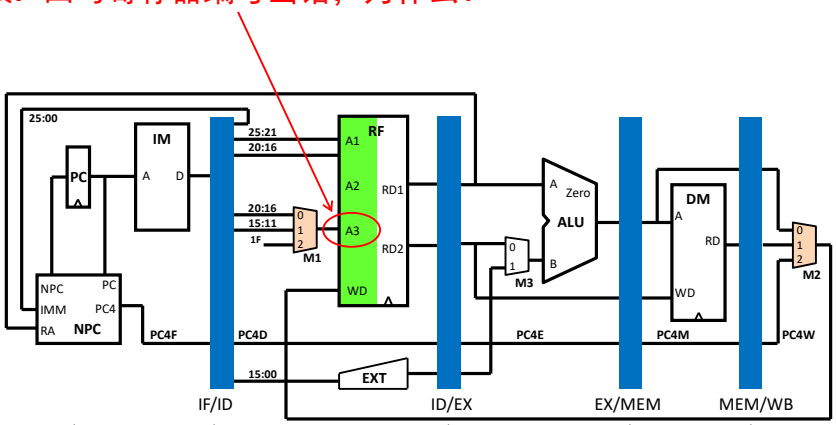


IF 取指令	ID 译码/读操作数	EX 执行	MEM 访存	WB 回写

示例：lw执行过程5/5

- 第5个时钟周期
  - MEM/WB将结果写入RF
- Q：回写寄存器编号出错，为什么？

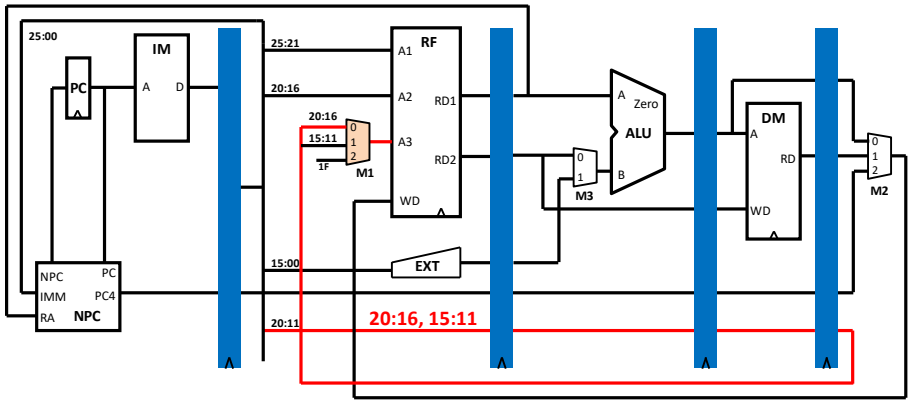
寄存器被写入
组合逻辑正在工作



IF 取指令	ID 译码/读操作数	EX 执行	MEM 访存	WB 回写

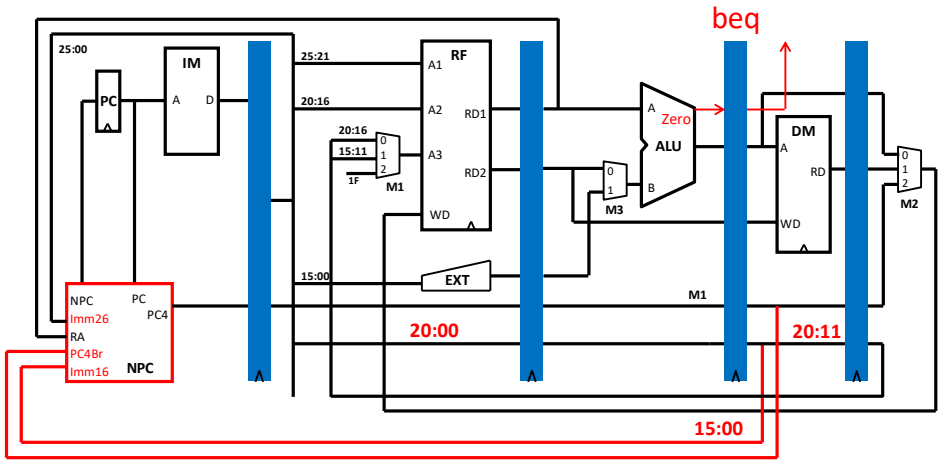
修正后的数据通路：RF的A3

- 出错原因在于指令的部分信息没有随指令执行同步传递
- 为了确保回写正确，寄存器回写编号必须同步传递



修正后的数据通路：beq

- 当beq进入EX/MEM后，EX/MEM才能输出正确的Zero，为此：
  - 必须将EX/MEM存储的PC+4和16位立即数传递给NPC
  - NPC也需要对输入端口进行相应调整



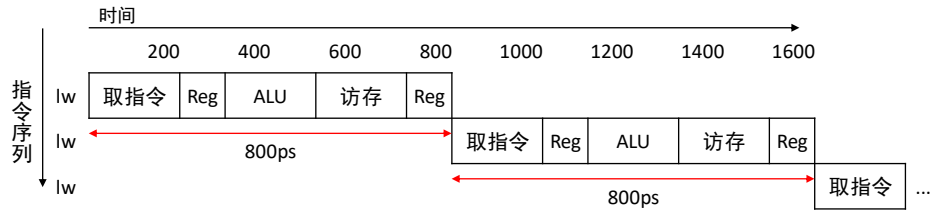
流水线性能

- 假设采用之前的性能参数，流水线的时钟频率是多少？
  - 单周期：lw是关键路径，延迟为800ps，因此时钟频率上限为1.25GHz
  - 流水线：各段最大延迟为200ps，即因此时钟频率上限为5GHz

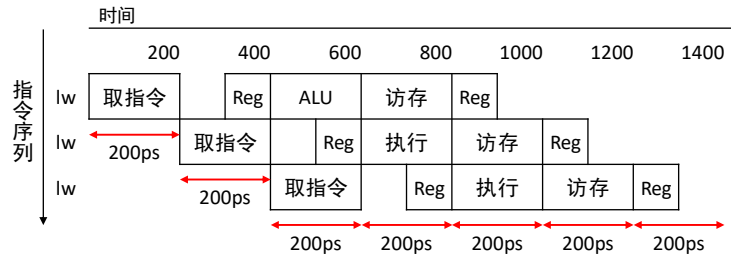
指令	读取指令	读寄存器	ALU	数据存取	写寄存器
addu	200	100	200		100
subu	200	100	200		100
ori	200	100	200		100
lw	200	100	200	200	100
sw	200	100	200	200	
beq	200	100	200		
jal	200				100
jr	200	100			

流水线性能

- 单周期：每个时钟周期800ps，主频1.25GHz



- 流水线：每个时钟周期200ps，主频5GHz



指令级并行

- 流水线使得CPU可以在同一时刻执行多条指令
  - 多条指令同时运行
  - 各功能部件利用率高
  - 大幅度提高了CPU的吞吐率
- 这种技术被称为指令级并行
  - ILP：Instruction Level Parallelism

23

更细致的流水线时空图

- 目的：尽可能的标识出每个功能部件在每个时钟周期执行的功能

相对PC的地址偏移		1	2	3	4	5	6	7	8	9
0	load	IM	RF(r)	ALU	DM(r)	RF(w)				
4	add		IM	RF(r)	ALU	--	RF(w)			
8	sub			IM	RF(r)	ALU	--	RF(w)		
12	store				IM	RF(r)	ALU	DM(w)	--	
16	or					IM	RF(r)	ALU	--	RF(w)

  - IM：读指令存储器
  - RF(r)：读寄存器堆； RF(w)：写寄存器堆
  - ALU：ALU计算
  - DM(r)：读数据存储器；DM(w)：写数据存储器

24

流水线加速比

$$\text{流水线加速比} = \frac{\text{单周期指令执行时间}}{\text{流水线的级数}}$$

- 潜在加速比 = 流水线级数
  - 假设单周期数据通路切分为N段，且各段延迟均相等，则流水线性能是单周期的N倍！
- 流水段执行时间不平衡，则加速比下降
- 填充流水线和排放流水线，导致加速比下降
- 流水线不改善单条指令执行周期数
- 流水线改善的是吞吐率
  - 理论上，当流水线充满后，每个时钟周期可以执行完一条指令
- 流水线时钟频率受限于最慢的流水段

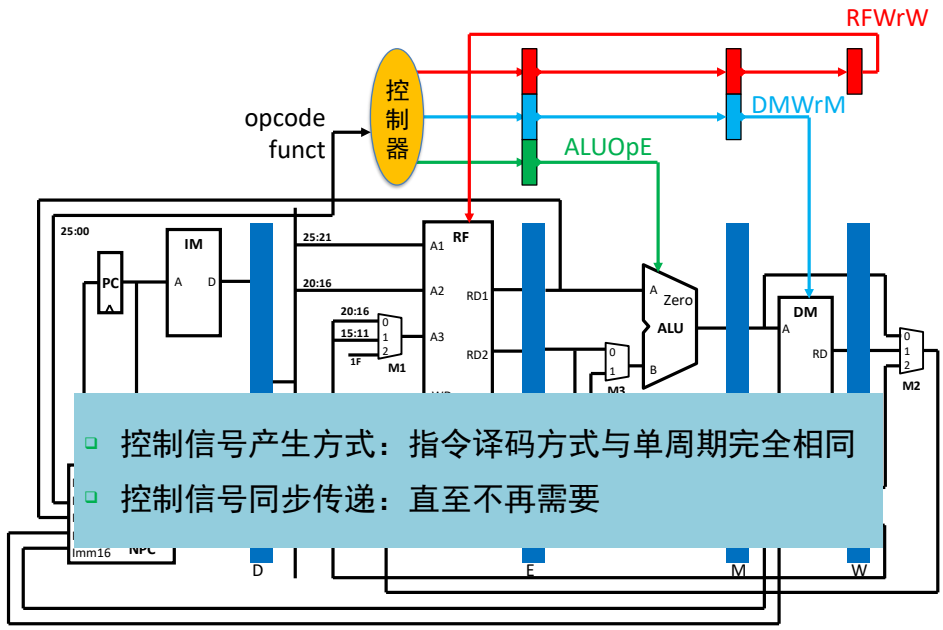
指令集对流水线设计的影响

- MIPS指令集是面向流水线架构设计的
- 所有的指令都是32位
  - 取指和译码都能在一个周期内完成
- 指令格式种类少且规整，2个源寄存器的位置保持不变
  - 这使得读取操作数和译码可以同时执行
- 存储器操作只有load和store
  - 第3拍计算地址，第4拍访存
- 存储器操作是地址对齐的
  - 有利于与主存的协同设计，且访存周期数固定

目录

- 流水线概述
- 流水线数据通路
- 流水线控制
- 流水线冒险
- 流水线性能分析
- 3种CPU模型对比

流水的控制信号



- 控制信号产生方式：指令译码方式与单周期完全相同
- 控制信号同步传递：直至不再需要

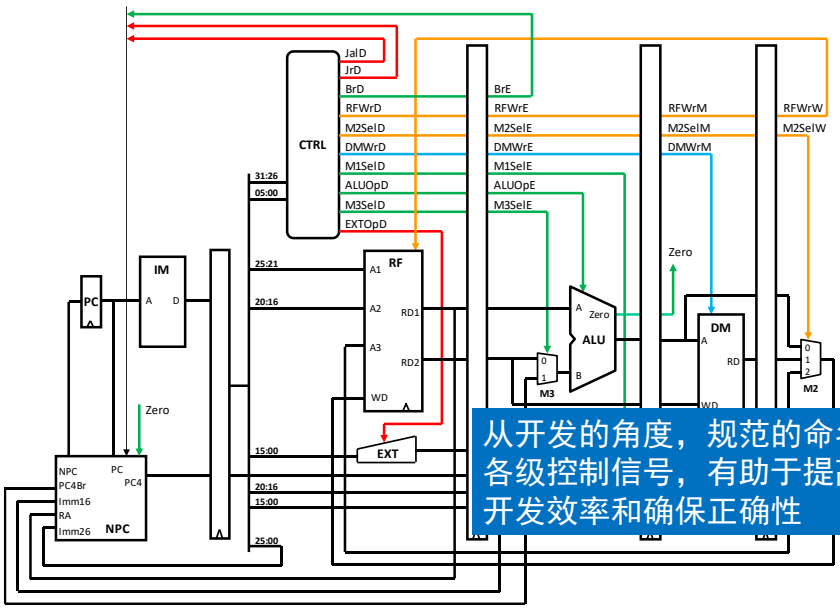
控制信号的额外含义

- 单周期
  - 控制信号只是用于控制功能部件执行正确的功能
  - 示例：RFWr有效意味着可以写入RF
- 流水线
  - 除了具有单周期的相同意义外，控制信号在各级寄存器中流水传递，还代表了指令执行到哪个阶段
  - 示例：RFWrW有效，表明load类、R型计算类、I型计算类等已经进入MEM/WB了，即指令执行到最后一个流水段了

Q：为什么需要了解指令进入到哪个阶段了？

已经在流水线中的多条指令，可能会发生冲突。为此，需要了解各条指令在流水线中的所处位置

流水线的主控制器



目录

- 流水线概述
- 流水线数据通路
- 流水线控制
- 流水线冒险
- 流水线性能分析
- 3种CPU模型对比

概述

- 结构冒险
  - 需要的资源被占用
- 数据冒险
  - 指令之间存在数据依赖
  - 后继指令需要等待前序指令执行结束
- 控制冒险
  - 指令流的方向选择依赖于前序指令执行结果

目录

- 流水线概述
- 流水线数据通路
- 流水线控制
- 流水线冒险
  - 结构冒险
- 流水线性能分析
- 3种CPU模型对比

结构冒险

- 后继指令需要的资源被前序指令占用，导致资源冲突
- 假设只有一个存储器，就必然会导致资源冲突

地址偏移		1 ↓	2 ↓	3 ↓	4 ↓	5 ↓	6 ↓	7 ↓	8 ↓	9 ↓
0	lw	IM	RF(r)	ALU	DM(r)	RF(w)				
4	add		IM	RF(r)	ALU	--	RF(w)			
8	sub			IM	RF(r)	ALU	--	RF(w)		
12	or				IM(r)	RF(r)	ALU	--	RF(w)	
16	and					IM	RF(r)	ALU	--	RF(w)

※第4个时钟周期时，lw和or均需要占用存储器（lw读数据，or取指令）

  - 当Load/Store读写存储器时，取指令由于存储器被占用就必须暂停
    - 导致流水线产生“空泡”
- 为此，流水线要求数据和指令必须分离存储
  - 这种结构实际上对应了分离的一级指令cache和一级数据cache

结构冒险

- 寄存器堆也可能会导致结构冒险

地址偏移		1	2	3	4	5	6	7	8	9
0	lw	IM	RF(r)	ALU	DM(r)	RF(w)				
4	add		IM	RF(r)	ALU	--	RF(w)			
8	sub			IM	RF(r)	ALU	--	RF(w)		
12	or				IM	RF(r)	ALU	--	RF(w)	
16	and					IM	RF(r)	ALU	--	RF(w)

※第5个时钟周期时，lw和or均需要占用寄存器（lw写寄存器，and读寄存器）

- 两种解决思路
  - 读写共用端口：要求必须双倍速率（读写各占半个周期）
  - 读写端口分离：读端口与写端口分离（这是目前的常见思路）
- 结论：寄存器堆不会成为瓶颈

目录

基于寄存器的数据相关

- 当2条指令都读写同一个寄存器时，这就产生了数据相关
  - 数据相关是计算机程序的基本特征
  - 指令流通过读写寄存器完成信息交换与传递
    - 主存单元也是指令流交换信息的重要途径之一
- 指令对寄存器的访问模式
  - 读(Read)：不会改变寄存器的值
  - 写(Write)：会改变寄存器的值（破坏性操作）
- 对于同一个寄存器，2条指令的操作序列有4种组合
  - R-R，W-W，R-W，W-R
    - 字母先后代表操作的先后顺序
  - 示例：
    - lw-sub：在$t0上是W-R
    - add-sub：在$t2上是W-W

```
lw  $t0, 0($t1)
sub $t3, $t0, $t2
and $t5, $t0, $t4
or  $t7, $t0, $t6
add $t1, $t2, $t3
```

```
lw  $t0, 0($t1)      W:写$t0
add $t2, 0($s0)      W:写$t2
sub $t2, $t0, $t4    R:读$t0
                     W:写$t2
```

数据相关与程序正确性

- 对于数据相关与程序正确性，程序员与CPU的视角是不同的
- 程序员的视角
  - 数据相关是程序员的“要求”：没有数据相关，程序员就无法写程序
  - 从程序员角度，只要前序指令完成后再执行后继指令即可
    - 这就是程序员逻辑
- CPU的视角：提高性能同时确保指令执行结果符合程序员逻辑
  - 单周期、多周期：CPU每次只执行一条指令，因此指令执行逻辑与程序员逻辑完全一致
  - 流水线：CPU能够同时执行多条指令，这就存在一个挑战性问题，即后执行的指令是否一定能获取先执行的指令的结果？

流水线并行执行指令与程序正确性

- 案例分析

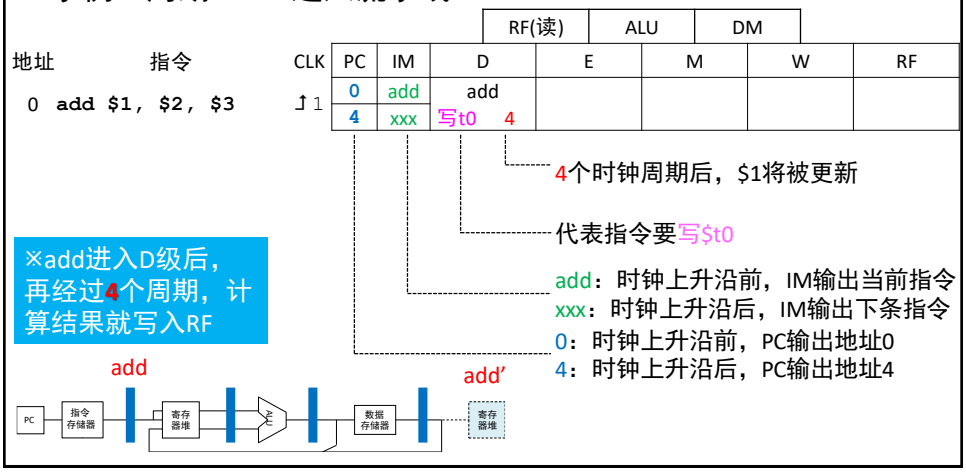
偏移	指令
0	add $1, $2, $3
4	sub $4, $1, $2
8	or $5, $1, $2
12	and $6, $1, $2
16	xor $7, $1, $2

	1	2	3	4	5	6	7	8	9
0	IM	RF(r)	ALU	DM(r)	RF(w)				
4		IM	RF(r)	ALU	--	RF(w)			
8			IM	RF(r)	ALU	--	RF(w)		
12				IM	RF(r)	ALU	--	RF(w)	
16					IM	RF(r)	ALU	--	RF(w)

- ✗ add-sub：sub第3周期时要读取$1，但add在第5周期才回写结果。因此，如果不加任何处理，则sub指令执行是错误的
- ✗ add-or：同上，执行是错误的
- ? add-and：两条指令在同一周期访问$1，and能否读出正确结果？
  - 这个问题后续再讨论
- ✓ add-xor：add第5周期回写结果，xor第6周期读取结果，因此执行正确

从时间角度观察流水线执行

- 换个视角：从同一个时钟周期角度，观察各条指令分别位于什么阶段以及在执行什么功能
  - 有一个重要细节，就是刻画时钟周期与时钟上升沿
- 示例：周期1~add进入流水线



数据相关的时间分析1/5

- 示例：周期2~sub进入流水线
  - sub进入D级后，马上就驱动RF读取$1

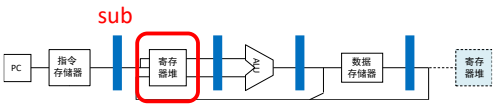
每经过1个时钟周期，结果产生的时间就减少1个周期

地址	指令
0	add $1, $2, $3
4	sub $4, $1, $2

CLK	PC	IM	D	E	M	W	RF
			RF(读)	ALU	DM		
↑1	0	add	add 写$1 4				
	4	sub					
↑2	4	sub	sub 读$1 0	add 写$1 3			
	8	or					

0个时钟周期后，sub就必须读取$1的值

代表指令要读$1



数据相关的时间分析2/5

- 示例：周期2~sub进入流水线
  - add-sub：由于结果无法及时回写会导致出错
    - sub马上要读取$1，而add还需要3个周期才能将最新值写入$1
    - 这意味着时间上无论如何也来不及了

地址	指令
0	add $1, $2, $3
4	sub $4, $1, $2

CLK	PC	IM	D	E	M	W	RF
			RF(读)	ALU	DM		
↑1	0	add	add 写1 4				
	4	sub					
↑2	4	sub	sub 读$1 0	add 写$1 3			
	8	or					

数据相关的时间分析3/5

- 示例：周期3~or进入流水线
  - add-or：由于结果无法及时回写会导致出错
    - 分析同上

地址	指令	CLK	PC	IM	D (RF(读))	E (ALU)	M (DM)	W	RF
0	add $1, $2, $3	↑1	0	add	add 写$1 4				
			4	sub					
4	sub $4, $1, $2	↑2	4	sub	sub 读$1 0	add 写$1 3			
			8	or					
8	or $5, $1, $2	↑3	8	or	or 读$1 0	sub	add 写$1 2		
			12	and					

数据相关的时间分析4/5

- 示例：周期4~and进入流水线
  - add-and：由于结果无法及时回写会导致出错
    - 分析同上

地址	指令	CLK	PC	IM	D (RF(读))	E (ALU)	M (DM)	W	RF
0	add $1, $2, $3	↑1	0	add	add 写$1 4				
			4	sub					
4	sub $4, $1, $2	↑2	4	sub	sub 读$1 1	add 写$1 3			
			8	or					
8	or $5, $1, $2	↑3	8	or	or 读$1 1	sub	add 写$1 2		
			12	and					
12	and $6, $1, $2	↑4	12	and	and 读$1 0	or	sub	add 写$1 1	
			16	xor					

数据相关的时间分析5/5

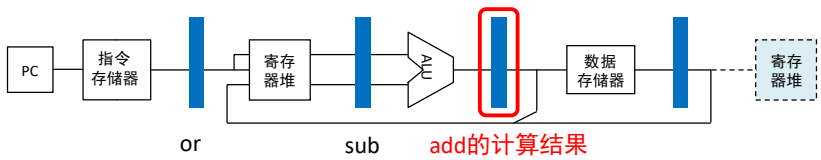
- 示例：周期5~xor进入流水线
  - add-xor：由于结果已经写回，因此可以正确执行
    - 在第5个↑后，$1保存了最新的计算结果。这意味着add的执行完全结束了
    - 同时，xor存入D，开始读取$1。这意味着xor能够获得最新的计算结果

地址	指令	CLK	PC	IM	D (RF(读))	E (ALU)	M (DM)	W	RF
0	add $1, $2, $3	↑1	0	add	add				
			4	sub	写$1 4				
4	sub $4, $1, $2	↑2	4	sub	sub	add			
			8	or	读$1 0	写$1 3			
8	or $5, $1, $2	↑3	8	or	or	sub	add		
			12	and	读$1 0	读$1 0	写$1 2		
12	and $6, $1, $2	↑4	12	and	and	or	sub	add	
			16	xor	读$1 0	读$1 0		写$1 1	
16	xor $7, $1, $2	↑5	16	xor	or	and	or	sub	add
			20		读$1 0				写$1 0

事实：计算结果在回写前已经产生了

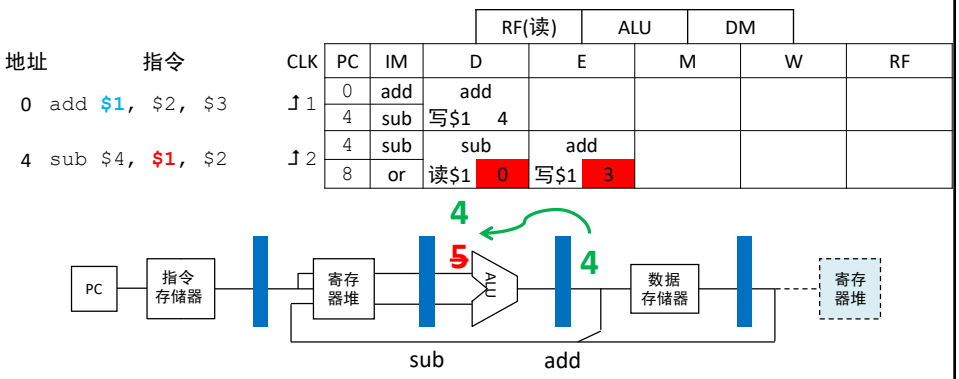
- 新的计算结果：先保存在流水线寄存器，后保存到寄存器堆
- 示例：add
  - 在第3个周期后，计算结果已经保存在M级了
  - 但还需要2个周期，计算结果才能保存到寄存器堆里

地址	指令	CLK	PC	IM	D (RF(读))	E (ALU)	M (DM)	W	RF
0	add $1, $2, $3	↑1	0	add	add				
			4	sub	写$1 4				
4	sub $4, $1, $2	↑2	4	sub	sub	add			
			8	or	读$1 0	写$1 3			
8	or $5, $1, $2	↑3	8	or	or	sub	add		
			12	and	读$1 0	读$1 0	写$1 2		



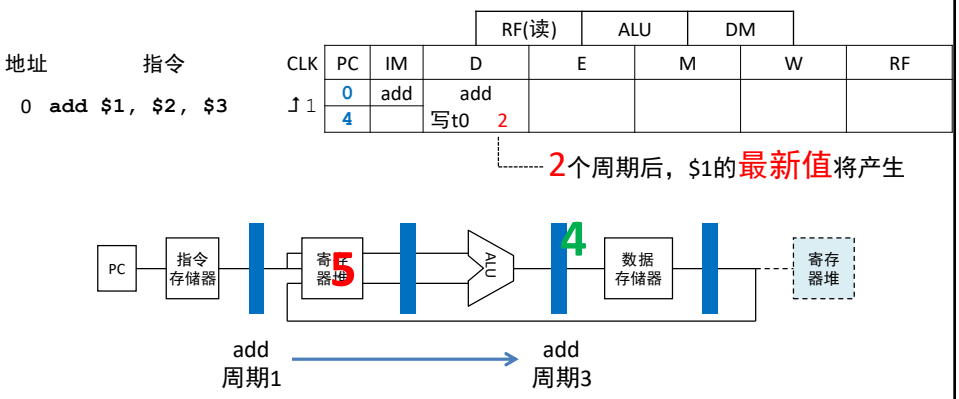
事实：功能部件是寄存器值的最终使用者

- 示例：假设RF中的$1值为5，add的计算结果为4
  - 对sub来说，只要送给ALU的值是4而不是5，就能确保正确执行



从计算结果产生角度观察1/3

- 示例：周期1~add进入流水线
  - 2个时钟周期后，ALU的计算结果(4)将保存在M级寄存器
  - 这意味着add进入M级寄存器



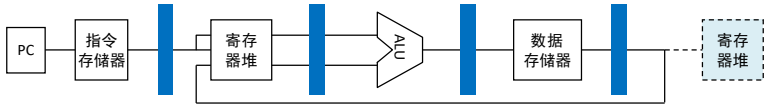
※假设：$1的值为5，add的计算结果为4

从功能部件使用寄存器值角度观察2/3

- 示例：周期2~add进入E级
  - 经过1个时钟周期后，结果产生的时间减少1个周期

地址	指令
0	add $1, $2, $3
4	sub $4, $1, $2

			RF(读)	ALU	DM		
CLK	PC	IM	D	E	M	W	RF
↑1	0	add	add 写$1 2				
	4	sub					
↑2	4	sub		add 写$1 1			
	8	or					



add

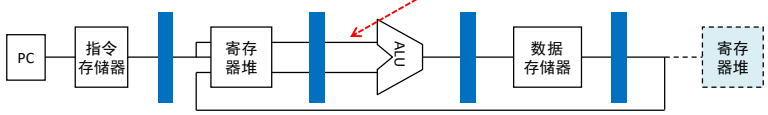
从功能部件使用寄存器值角度观察3/3

- 示例：周期2~sub进入流水线
  - 当sub进入D级后，则sub将在下个周期（周期3）进入E级
  - 这意味着从ALU角度，再经过1个周期就要用$1的值

地址	指令
0	add $1, $2, $3
4	sub $4, $1, $2

			RF(读)	ALU	DM		
CLK	PC	IM	D	E	M	W	RF
↑1	0	add	add 写$1 2				
	4	sub					
↑2	4	sub	sub 读$1 1	add 写$1 1			
	8	or					

1个周期后，ALU服务于sub，要用$1值



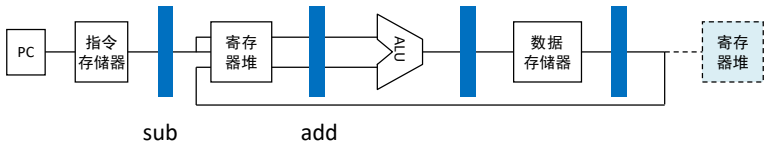
sub add

从功能部件使用寄存器值角度观察

- 示例：周期2~sub进入流水线
  - 对于add-sub来说，在下个周期，add产生了最新计算结果（即$1值），而sub要使用$1值
  - 因此，若有一种方法将add计算结果传递至ALU，则sub可正确执行

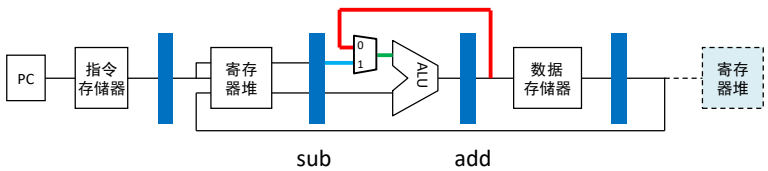
```
地址        指令
0  add $1, $2, $3
4  sub $4, $1, $2
```

			RF(读)	ALU	DM		
CLK	PC	IM	D	E	M	W	RF
↑1	0	add	add 写$1 2				
	4	sub					
↑2	4	sub	sub 读$1 **1**	add 写$1 **1**			
	8	or					



数据冒险解决策略—转发1/2

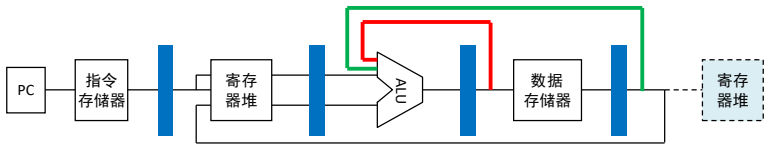
- 转发（或旁路）：将尚未写入RF但已经暂存在流水线寄存器中的计算结果传递给相关功能部件的技术
- 示例：M级向ALU的A输入端的转发
  - 这个转发电路就能很好的解决add-sub间的数据冒险



- 转发的2个基本要点
  - 前递：将M级(后级)保存的计算结果向E级(前级)的ALU(功能单元)传递
  - 选择：如果E级指令(sub)读的寄存器与M级指令(add)写的寄存器是同一寄存器时，则选择转发的计算结果为操作数，否则选择E级传递的寄存器值

数据冒险解决策略—转发2/2

- 在设计转发时，要考虑完备性
- 示例：ALU的A输入端
  - ALU处于E级，因此M级和W级均可能保存有前序指令的计算结果
  - 只有将M级和W级均转发到E级，才确保了相关组合指令的正确执行



- M→E转发：能够解决的相关例子

```
add $1, $2, $3
sub $4, $1, $6
```

- W→E转发：能够解决的相关例子

```
add $1, $2, $3
xxx
or  $5, $1, $2
```

Load导致的数据冒险

- 周期1：lw进入流水线
  - 3个周期后，W级将保存lw从DM中读出的数据

			RF(读)	ALU	DM		
地址	指令	CLK	PC	IM	D	E	M	W	RF
0	lw $t0, 0($t1)	↑1	0	lw	lw				
4	sub $t3, $t0, $t2	↑2	4	sub	写t0 3				
8	and $t5, $t0, $t4	↑3							
12	or $t7, $t0, $t6	↑4							
16	add $t1, $t2, $t3	↑5							

Load导致的数据冒险

- 周期2：sub进入流水线
  - sub再过1个周期到达E时，ALU要用$t0
  - 而lw还需要2个周期才能产生结果

			RF(读)	ALU	DM		
地址 指令	CLK	PC IM	D	E	M	W	RF
0 lw $t0, 0($t1)	↑1	0 lw 4 sub	lw 写t0 3				
4 sub $t3, $t0, $t2	↑2	4 sub 8 and	sub 读t0 1	lw 写t0 2			
8 and $t5, $t0, $t4	↑3						
12 or $t7, $t0, $t6	↑4						
16 add $t1, $t2, $t3	↑5						

- 结论：新值产生太晚！除暂停sub，无任何办法能消除这个冲突

Load导致的数据冒险

- 周期3：插入NOP
  - 所谓的NOP，就是清空E级，相当于插入空拍
  - 注意，除了清空E级，还需要冻结PC（即PC必须保持不变）

			RF(读)	ALU	DM		
地址 指令	CLK	PC IM	D	E	M	W	RF
0 lw $t0, 0($t1)	↑1	0 lw 4 sub	lw 写t0 3				
4 sub $t3, $t0, $t2	↑2	4 sub 8 and	sub 读t0 1	lw 写t0 2			
8 and $t5, $t0, $t4	↑3	8 and 8 and	sub 读t0 1	nop	lw 写t0 1		
12 or $t7, $t0, $t6	↑4						
16 add $t1, $t2, $t3	↑5						

- 由于插入了NOP，因此冲突解除
  - 转发机制将在周期4发挥作用

Load导致的数据冒险

- 周期4-1：sub到达E
  - lw：从DM中读出的数据($t0新值)被写入W
  - sub：选择来自W级转发的计算结果

地址	指令	CLK	PC	IM	D	E	M	W	RF
					RF(读)	ALU	DM		
0	lw $t0, 0($t1)	↑1	0 4	lw sub	lw 写t0 3			计算结果	
4	sub $t3, $t0, $t2	↑2	4 8	sub and	sub 读t0 1	lw 写t0 2			
8	and $t5, $t0, $t4	↑3	8 8	and and	sub 读t0 1	nop	lw 写t0 1		
12	or $t7, $t0, $t6	↑4	8 12			sub 读t0 0	nop	lw 新t0 0	
16	add $t1, $t2, $t3	↑5							

Load导致的数据冒险

- 周期4-2：and进入流水线
  - 注意，lw-xxx-and同样需要转发，用W级计算结果替代and读出的$1
  - 但是，这个转发的目的地不是功能单元，而是E级流水线寄存器

地址	指令	CLK	PC	IM	D	E	M	W	RF
					RF(读)	ALU	DM		
0	lw $t0, 0($t1)	↑1	0 4	lw sub	lw 写t0 3			计算结果	
4	sub $t3, $t0, $t2	↑2	4 8	sub and	sub 读t0 1	lw 写t0 2			
8	and $t5, $t0, $t4	↑3	8 8	and and	sub 读t0 1	nop	lw 写t0 1		
12	or $t7, $t0, $t6	↑4	8 12	and or	and 读t0 1	sub 读t0 0	nop	lw 新t0 0	
16	add $t1, $t2, $t3	↑5							

Load导致的数据冒险

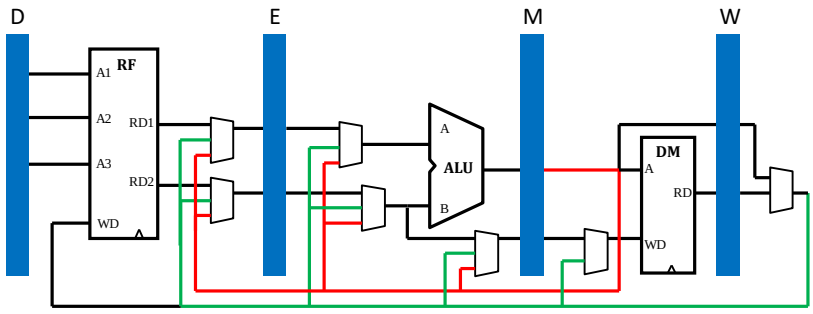
- 周期5：or进入流水线
  - 由于流水线中没有写$t0的指令了，这意味着RF中的$t0是最新值
  - or指令可以安全的从RF中读取$t0

地址	指令	CLK	PC	IM	D (RF(读))	E (ALU)	M (DM)	W (DM)	RF
0	lw $t0, 0($t1)	↑1	0 4	lw sub	lw 写t0 3				
4	sub $t3, $t0, $t2	↑2	4 8	sub and	sub 读t0 1	lw 写t0 2			
8	and $t5, $t0, $t4	↑3	8 8	and and	sub 读t0 1	nop	lw 写t0 1		
12	or $t7, $t0, $t6	↑4	8 12	and or	and 读t0 1	sub 读t0 0	nop	lw 写t0 0	
16	add $t1, $t2, $t3	↑5	12 16	or add	or 读t0 1	and	sub	nop	更新t0

详细的转发电路及其控制

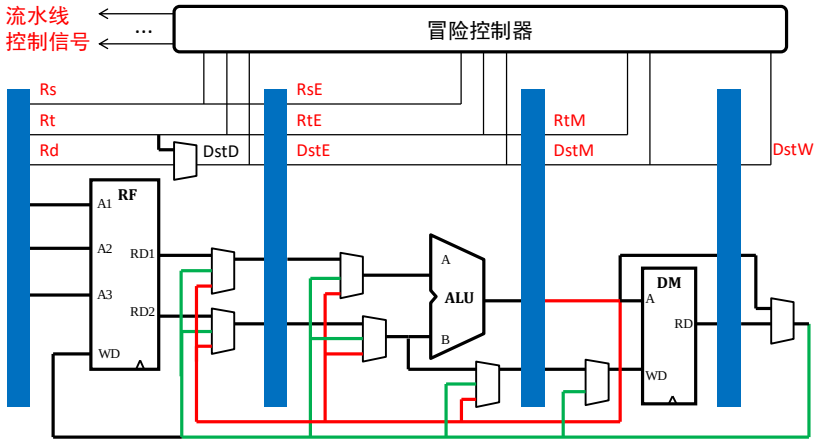
- 有多少种转发？
  - 最新结果：可能出现在M和W
  - RS寄存器值：可能出现在E、ALU
  - RT寄存器值：可能出现在E、ALU、M、DM

转发源	转发目的
M级 ALU计算结果	E级中的RS寄存器值 E级中的RT寄存器值 ALU的A ALU的B
W级 回写结果	E级中的RS寄存器值 E级中的RT寄存器值 ALU的A ALU的B DM的WD



冒险控制器

- 基本功能：在主控制器的配合下，检测和分析各类冒险，并控制流水线的执行
- 先讨论应对数据冒险的基本控制方法



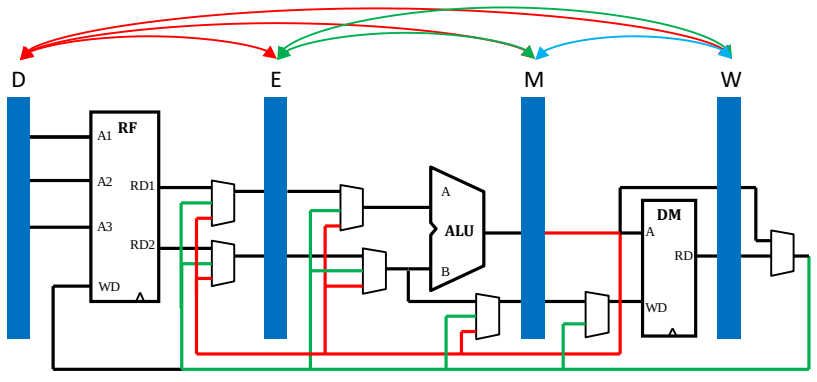
数据冒险的检测与控制

- 基本策略

D级　分析与E/M/W的相关性。若转发无法解决则暂停，直至转发可以解决相关

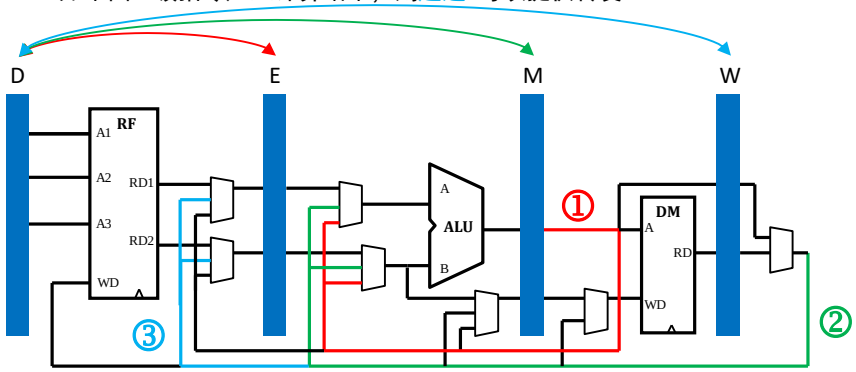
E级　E级已无暂停问题，只需要分析与M/W的相关性，决定是否转发即可

M级　同E级



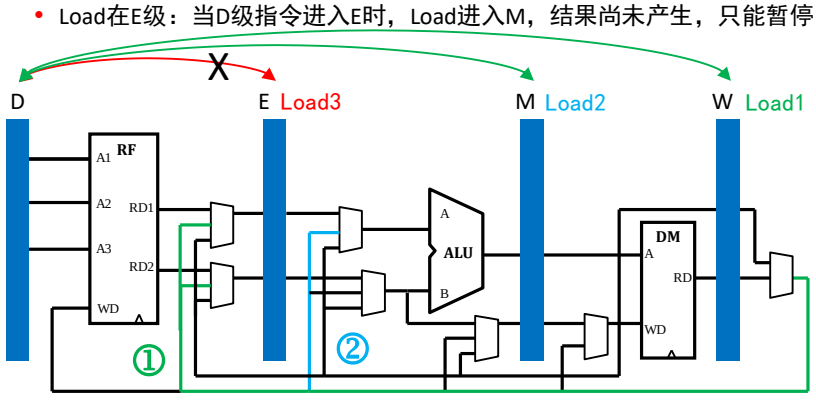
数据冒险的检测与控制：暂停(以RS为例)1/10

- 对于D级指令来说，哪些前序指令会导致暂停？
  - 计算类指令（如add，addi）
    - 如果由E级指令产生计算结果，则通道①可以提供转发
    - 如果由M级指令产生计算结果，则通道②可以提供转发
    - 如果由W级指令产生计算结果，则通道3可以提供转发



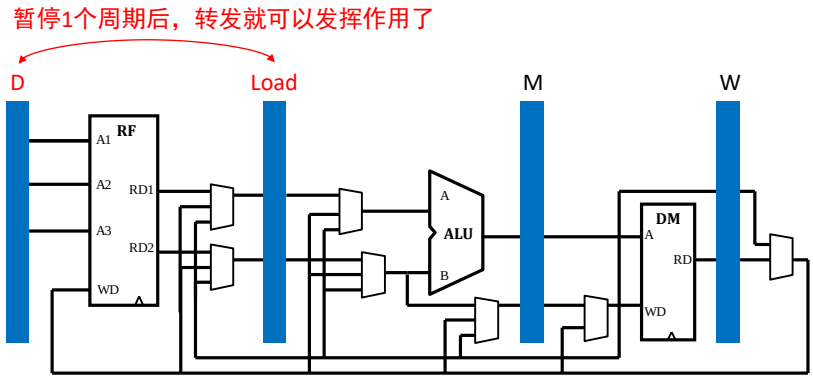
数据冒险的检测与控制：暂停(以RS为例)2/10

- 对于D级指令来说，哪些前序指令会导致暂停？
  - Load指令（如lw/lh/lhu/lb/lbu）
    - Load在W级：由于结果已经在W级了，因此通道①转发
    - Load在M级：当D级指令进入E时，Load的结果存入W，通道②转发
    - Load在E级：当D级指令进入E时，Load进入M，结果尚未产生，只能暂停



数据冒险的检测与控制：暂停(以RS为例)3/10

- 对于D级指令来说，哪些前序指令会导致暂停？
  - Load指令（如lw/lh/lhu/lb/lbu）
- 结论：Load在E级则需要暂停；其他情况均可以通过转发

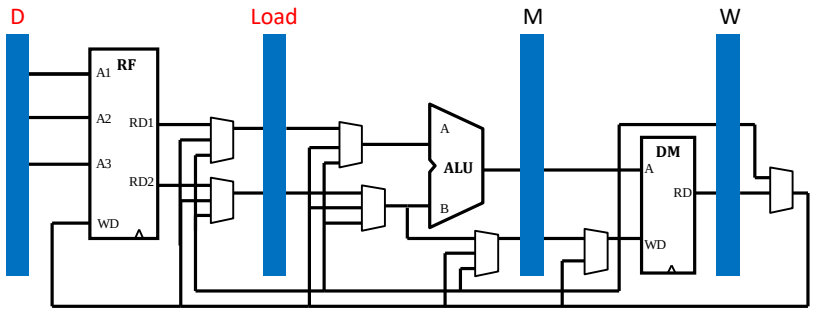


数据冒险的检测与控制：暂停(以RS为例)4/10

- 暂停的检测：暂停有哪些条件？
  - 条件1：D级要读寄存器
  - 条件2：E级为load类指令
  - 条件3：读的寄存器与写的寄存器相同

Q1：哪些指令读寄存器？

Q2：哪些指令是load类指令？



数据冒险的检测与控制：暂停(以RS为例)5/10

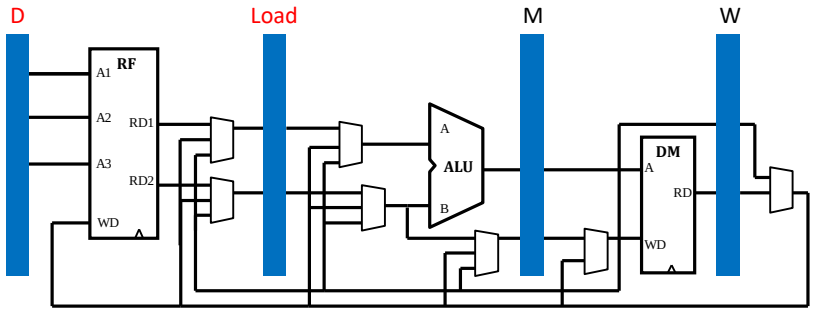
- 暂停的检测：表达式

```
Stall = (add + sub +…+ lw) & lwE & (rs==DstE)
```

寄存器编号相同

E级为Load类

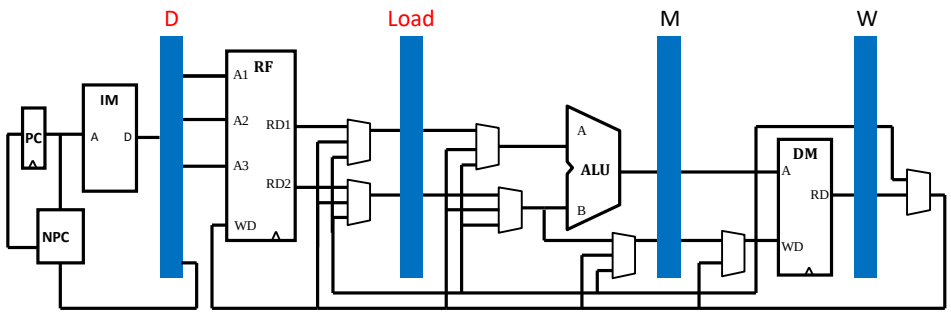
D级会读rs寄存器的指令



数据冒险的检测与控制：暂停(以RS为例)6/10

- 暂停的执行动作
  - ①冻结D：sub继续被保存
  - ②清除E：指令全为0，等价于插入NOP
  - ③禁止PC：防止PC继续计数，PC应保持为PC+4

地址	指令
0	lw $t0, 0($t1)
4	sub $t3, $t0, $t2
8	and $t5, $t0, $t4
12	or $t7, $t0, $t6
16	add $t1, $t2, $t3

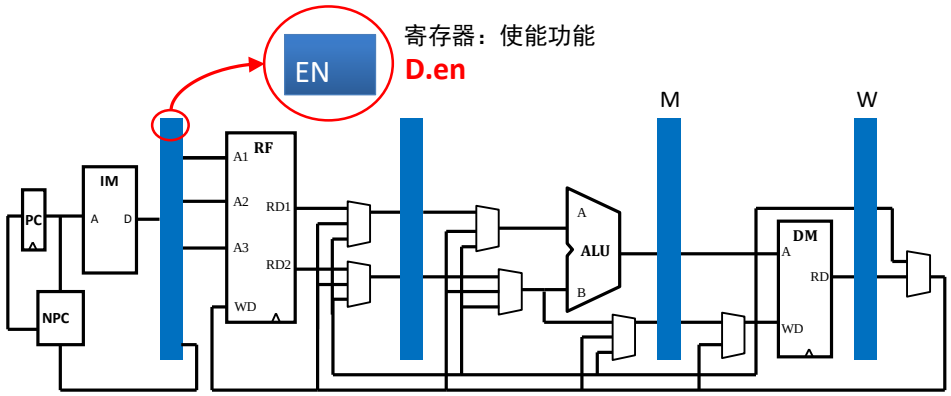


68

数据冒险的检测与控制：暂停(以RS为例)7/10

- 暂停的执行动作
  - ①冻结D：sub继续被保存
  - ②清除E：指令全为0，等价于插入NOP
  - ③禁止PC：防止PC继续计数，PC应保持为PC+4

数据通路	将D寄存器修改为使能型寄存器
控制器	增加D寄存器使能信号D.en

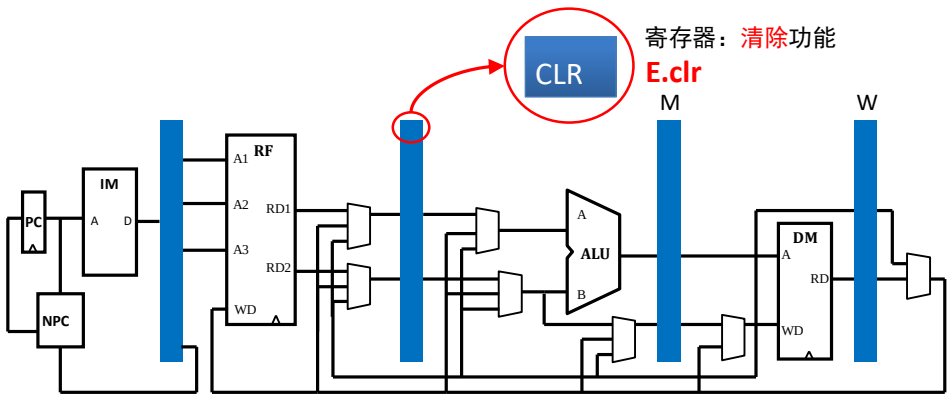


69

数据冒险的检测与控制：暂停(以RS为例)8/10

- 暂停的执行动作
  - ①冻结D：sub继续被保存
  - ②清除E：指令全为0，等价于插入NOP
  - ③禁止PC：防止PC继续计数，PC应保持为PC+4

数据通路	将E寄存器修改为复位型寄存器
控制器	增加E寄存器清除信号E.clr

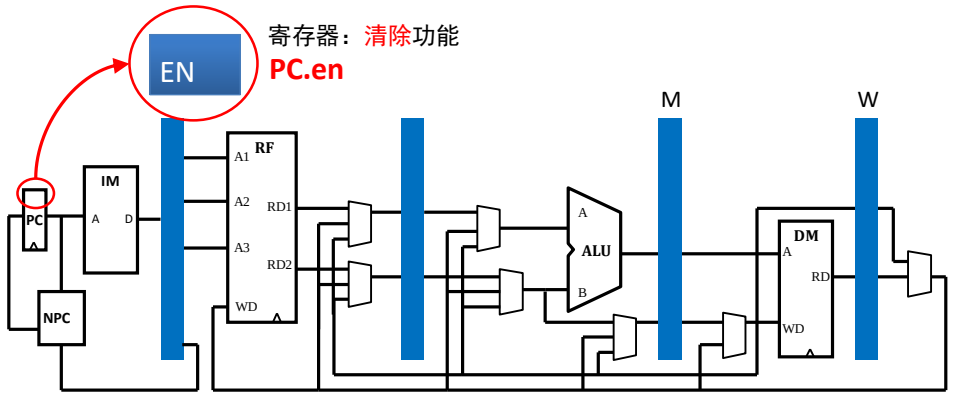


70

数据冒险的检测与控制：暂停(以RS为例)9/10

- 暂停的执行动作
  - ①冻结D：sub继续被保存
  - ②清除E：指令全为0，等价于插入NOP
  - ③禁止PC：防止PC继续计数，PC应保持为PC+4

数据通路	将PC修改为使能型寄存器
控制器	增加PC使能信号PC.en

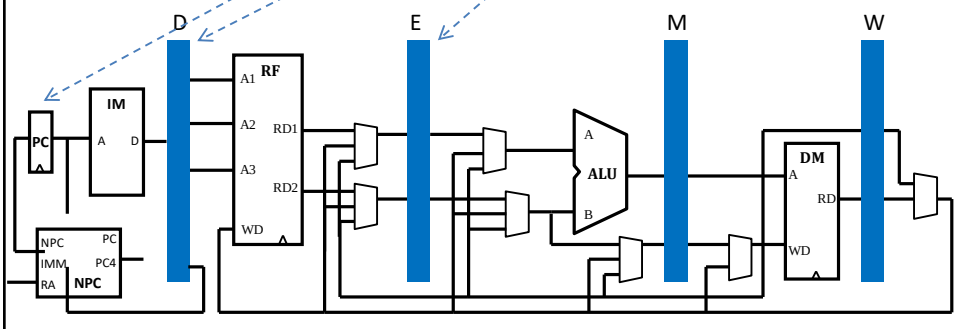


71

数据冒险的检测与控制：暂停(以RS为例)10/10

- 暂停的执行动作：控制信号表达式
  - ①冻结D：sub继续被保存
  - ②清除E：指令全为0，等价于插入NOP
  - ③禁止PC：防止PC继续计数，PC应保持为PC+4

```
PC.en = Stall
D.en  = Stall
E.clr = Stall
```



72

编译优化

- load类指令后继指令如果与load相关，则需要暂停一个周期
- 这个暂停周期被称为加载延迟槽（load delay slot）
- 如果利用编译技术将一条无load相关的指令放置在load类指令后，则就不需要暂停了！
  - 这就是编译优化

编译优化

- 编译优化：重排序指令序列从而避免load类相关
- C代码：**A=B+E; C=B+F;**

目录

- 流水线概述
- 流水线数据通路
- 流水线控制
- 流水线冒险
  - 控制冒险
- 流水线性能分析
- 3种CPU模型对比

控制冒险

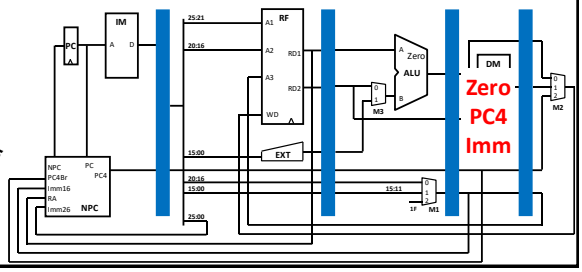
- 分支指令（beq，bne）影响控制流
  - 顺序取指还是转移取指取决于ALU的比较结果
  - 在比较结果产生以前，D级取指无法确保是正确的
- 简单方案：暂停分支指令直至产生正确的PC值
  - 问题：需要等待多长时间？

B指令冒险造成的停顿代价

地址	指令
0	beq $1, $3, 24
4	and $12, $2, $5
8	or $13, $6, $2
12	add $14, $2, $2
28	lw $4, 100($7)

CLK	PC	IM	RF(读) D	ALU E	DM M	DM W	RF
↑1	0 4	beq and	beq				
↑2	4 4	and and	nop(1)	beq			
↑3	4 4	and and	nop(2)	nop(1)	beq zero=1		
↑4	4 28	and lw	nop(3)	nop(2)	nop(1)		
↑5	28 32	lw XXX	lw	nop(3)	nop(2)	nop(1)	

- 如无措施，则须插入3个NOP
  - Zero、PC4及偏移均在M级，故PC在clk4才能加载正确值
  - D级在clk5才能存入转移指令(即lw)



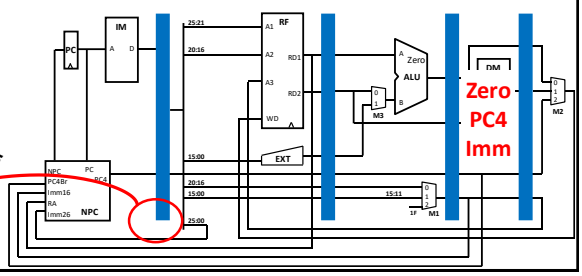
B指令冒险造成的停顿代价

地址	指令
0	beq $1, $3, 24
4	and $12, $2, $5
8	or $13, $6, $2
12	add $14, $2, $2
28	lw $4, 100($7)

CLK	PC	IM	RF(读) D	ALU E	DM M	DM W	RF
↑1	0 4	beq and	beq				
↑2	4 4	and and	nop(1)	beq			
↑3	4 4	and and	nop(2)	nop(1)	beq zero=1		
↑4	4 28	and lw	nop(3)	nop(2)	nop(1)		
↑5	28 32	lw XXX	lw	nop(3)	nop(2)	nop(1)	

- 如无措施，则须插入3个NOP
  - Zero、PC4及偏移均在M级，故PC在clk4才能加载正确值
  - D级在clk5才能存入转移指令(即lw)

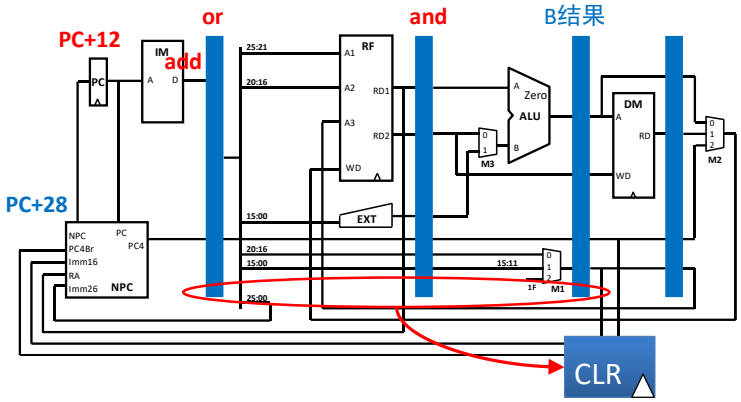
Q：Clr的表达式？ CLR



方案1：假定分支不发生

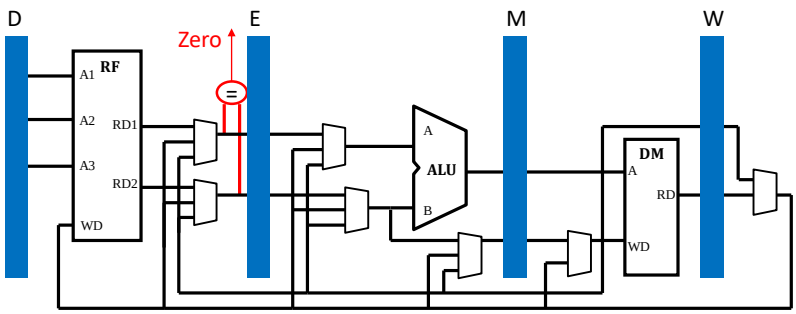
- 即使在D级发现是B指令也不停顿
- 根据B指令结果，决定是否清除3条后继指令
  - 清除，即使得and/or/add不能前进

地址	指令
0	beq $1, $3, 24
4	and $12, $2, $5
8	or $13, $6, $2
12	add $14, $2, $2
28	lw $4, 50($7)



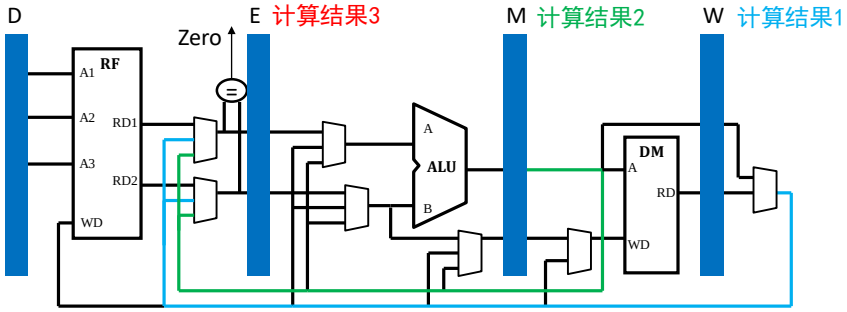
方案2：缩短分支延迟1/3

- 在D阶段放置比较器，尽快得到B指令结果
  - B指令结果可以提前2个clock得到
  - B指令后继可能被废弃的指令减少为1条
    - 当需要转移时，清除D级即可



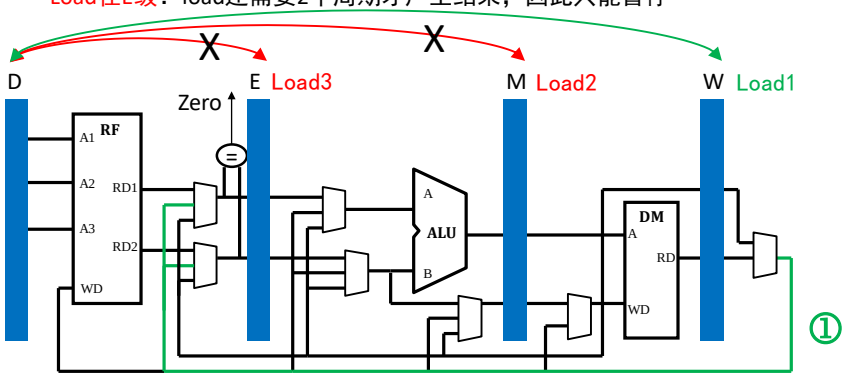
方案2：缩短分支延迟2/3

- 比较器前置后会产生数据相关（可能依赖于前序指令的结果）
  - 前序指令是计算类指令（如add，addi）
    - 依赖W级指令的计算结果1：从W转发数据
    - 依赖M级指令的计算结果2：从M转发数据
    - 依赖E级指令的计算结果3：只能暂停



方案2：缩短分支延迟3/3

- 比较器前置后会产生数据相关（可能依赖于前条指令的结果）
  - 前序指令是load类指令
    - Load在W级：结果已经产生，转发！
    - Load在M级： load还需要1个周期才产生结果，因此只能暂停
    - Load在E级：load还需要2个周期才产生结果，因此只能暂停

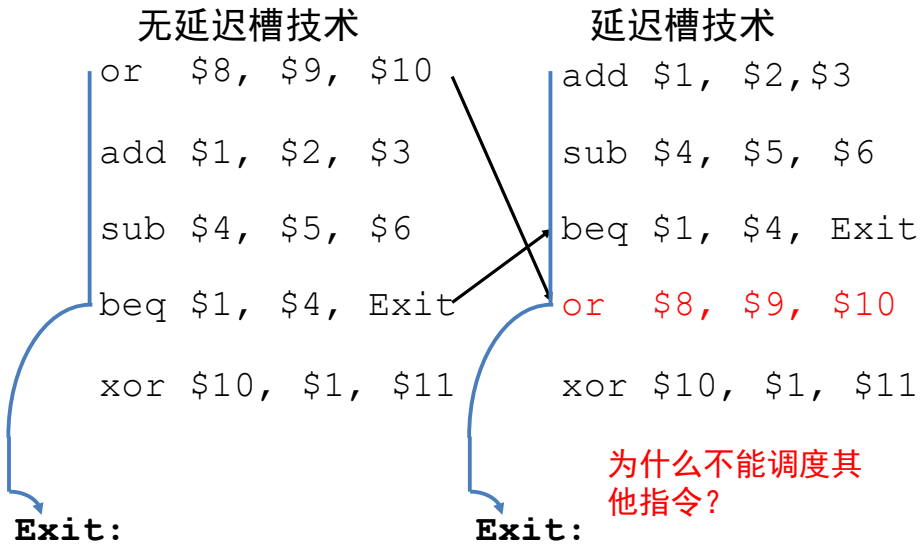


方案3：分支延迟槽1/3

- 无论是否执行转移方向的指令，都执行分支指令后面的那条指令
  - 这一技术就是分支延迟槽（Branch delay slot）
- 最坏情况：在分支指令后面放置一条nop
- 最好情况：编译从分支指令前面的指令中调度一条无关指令到分支指令后面
  - 编译优化必须确保不能改变程序逻辑
- J、Jal、Jr：是否也可以采用这一技术？

方案3：分支延迟槽2/3

- 示例



方案3：分支延迟槽3/3

- 分支指令的PC+8问题
  - 常规上，分支指令是以PC+4为基地址加偏移来计算转移地址
  - 由于在分支指令后面人为的放一条指令，因此转移地址就多了4
  - 支持延迟槽的CPU需要自动用PC+8作为基地址
- Jal指令的PC+8问题
  - 对于jal，可以同样在其后人为的放一条指令。同样，返回地址多了4
  - 为此，保存在R[31]的地址就不能是PC+4，而必须是PC+8！
    - 否则`jr $31`会返回至延迟槽指令，而不是延迟槽下面的那条指令

目录

流水线冒险小结

- 冒险降低流水线效率
  - 导致暂停
- 结构冒险
  - 由于功能部件争用所致
- 数据冒险
  - 需要等待前序指令的执行结果
- 控制冒险
  - 控制流的方向确定前，下条指令无法确定
  - 分支指令和跳转指令延迟槽

目录

流水线CPI

- 理论上，流水线每个时钟周期能执行完1条指令，即

$$CPI_{理想}=1$$

- 然而，冒险会导致流水线暂停，即

$$CPI_{实际}>1$$

- 因为不同的冒险带来不同的暂停代价，所以$CPI_{实际}$与运行程序各类指令的占比相关

流水线CPI

- 示例：计算流水线的CPI
  - 某程序指令分布如下：load占25%，store为10%，分支指令为11%，R型计算类指令为54%
  - 假设：①load导致暂停概率为40%，且暂停代价为1个时钟周期。②分支指令预测成功率为75%，但预测失败就需要暂停1个时钟周期
- 解
  - load：无数据相关时，load的CPI为1；有数据相关时，CPI为2

$$CPI_{load}=1\times(1-40\%)+2\times40=1.4$$

  - store：CPI_{store}为1
  - 分支：预测成功，分支的CPI为1；预测失败，分支的CPI为2

$$CPI_{分支}=1\times75\%+2\times(1-75\%)=1.25$$

  - R型：$CPI_{R型}$为1

$$CPI=CPI_{load}\times25\%+CPI_{store}\times10\%+CPI_{分支}\times11\%+CPI_{R型}\times54\%=1.1275$$

目录

3种CPU模型对比

- 执行周期
  - 单周期：恒为1
  - 多周期：可变；某些指令的周期数最长，如load类
  - 流水线：理想为1，但冒险存在导致大于1
- 时钟频率
  - 单周期：关键路径存在导致最低
  - 多周期：最快（比流水线快的原因在于没有转发电路以及控制简单）
    - 理论上，如果各段延迟相同，则N倍单周期。但实际很难达到
  - 流水线：类似多周期，但转发和控制导致频率下降

3种CPU模型对比

- 性能计算
  - 单周期：仅与执行的指令数及时钟周期宽度相关
  - 多周期：不仅与时钟周期宽度相关，还与指令频度及其相应的CPI相关
  - 流水线：与多周期相同
- 设计要点
  - 单周期：数据通路是基础；控制信号仅与指令功能相关
  - 多周期：数据通路通过切割关键路径来提升时钟频率；控制信号还与时间相关
  - 流水线：数据通路除了继承多周期提升频率的思路外，还通过转发解决数据冒险；控制信号分为主控制与冒险控制两大部分
    - 主控制：与单周期、多周期相同，只与单条指令相关
    - 冒险控制：主要解决各类冒险带来的转发与暂停