计算机系统结构

第3章 流水线技术

目录

3.4 流水线的相关与冲突

3.4.1 一条经典的5段RISC流水线

首先讨论在非流水情况下是如何实现的

1. 一条指令的执行过程分为以下5个时钟周期：

- 取指令周期（IF）
  - 以程序计数器PC中的内容作为地址，从存储器中取出指令并放入指令寄存器IR；
  - 同时PC值加4（假设每条指令占4个字节），指向顺序的下一条指令。
- 指令译码/读寄存器周期（ID）

  对指令进行译码，并用IR中的寄存器地址去访问通用寄存器组，读出所需的操作数。

- 执行/有效地址计算周期（EX）

  不同指令所进行的操作不同：

  - load和store指令：ALU把指令中所指定的寄存器的内容与偏移量相加，形成访存有效地址。
  - 寄存器－寄存器ALU指令：ALU按照操作码指定的操作对从通用寄存器组中读出的数据进行运算。
  - 寄存器－立即数ALU指令：ALU按照操作码指定的操作对从通用寄存器组中读出的操作数和指令中给出的立即数进行运算。
  - 分支指令：ALU把指令中给出的偏移量与PC值相加，形成转移目标的地址；同时，对在前一个周期读出的操作数进行判断，确定分支是否成功。

- 存储器访问 / 分支完成周期（MEM）

  该周期处理的指令只有load、store和分支指令。其它类型的指令在此周期不做任何操作。

  - load和store指令

  load指令：用上一个周期计算出的有效地址从存储器中读出相应的数据；

  store指令：把指定的数据写入这个有效地址所指出的存储器单元。

  - 分支指令

    分支“成功”，就把转移目标地址送入PC。

    分支指令执行完成。

- 写回周期（WB）

  ALU运算指令和load指令在这个周期把结果数据写入通用寄存器组。

  ALU运算指令：结果数据来自ALU。

  load指令：结果数据来自存储器。

在这个实现方案中：

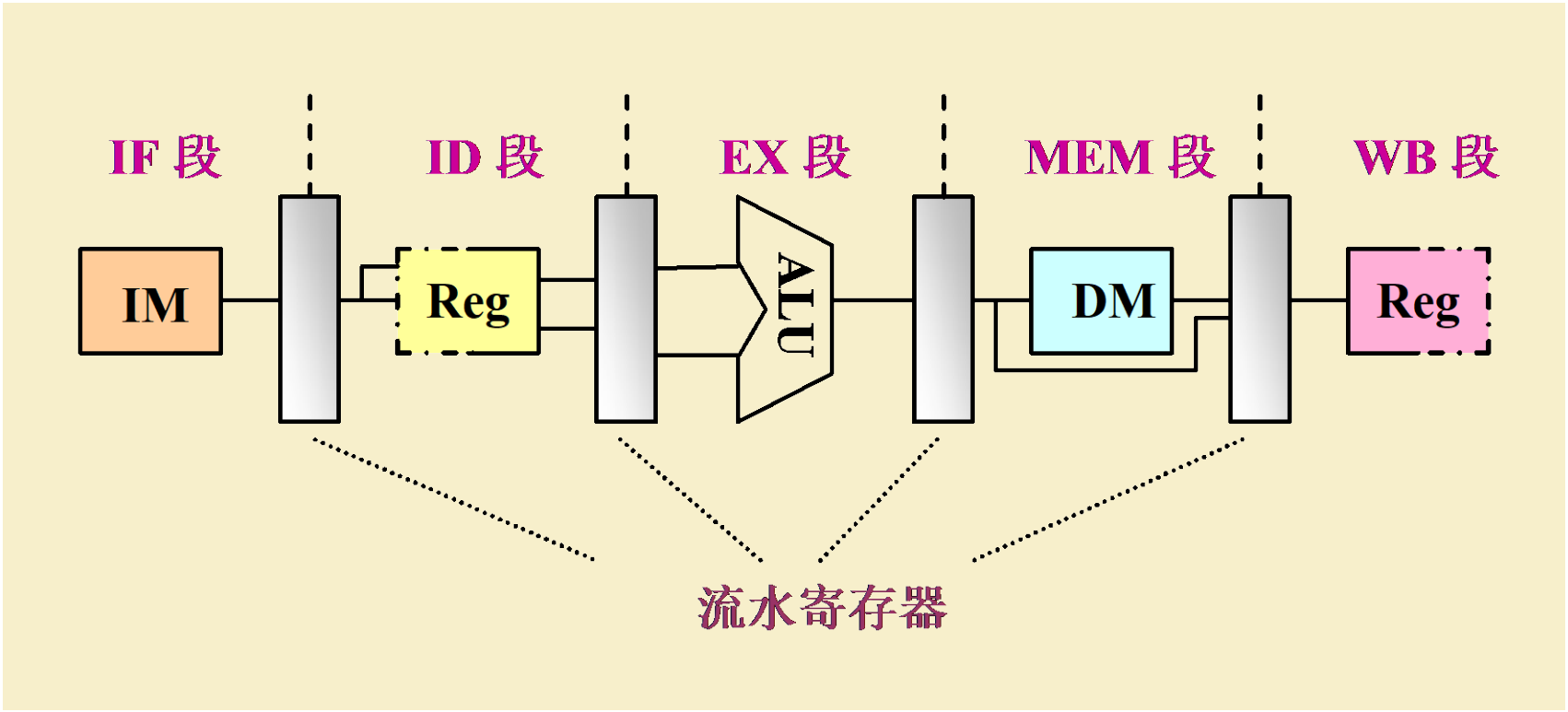
- 分支指令需要4个时钟周期

（如果把分支指令的执行提前到ID周期，则只需要2个周期）；

- store指令需要4个周期；
- 其它指令需要5个周期才能完成。

2. 将上述实现方案修改为流水线实现

- 一条经典的5段流水线
  - 每一个周期作为一个流水段；
  - 在各段之间加上锁存器（流水寄存器）。



3. 采用流水线实现时，应解决以下几个问题：

- 要保证不会在同一时钟周期要求同一个功能段做两件不同的工作。

  例如：不能要求ALU同时做有效地址计算和算术运算。

- 避免IF段的访存（取指令）与MEM段的访存（读/写数据）发生冲突。
  - 可以采用分离的指令存储器和数据存储器；
  - 一般采用分离的指令Cache和数据Cache。
- ID段和WB段都要访问同一寄存器文件。

  ID段：读　　WB段：写

  如何解决对同一寄存器的访问冲突？

  把写操作安排在时钟周期的前半拍完成，

  把读操作安排在后半拍完成。

- 考虑PC的问题
  - 流水线为了能够每个时钟周期启动一条新的指令，就必须在每个时钟周期进行PC值的加4操作，并保留新的PC值。这种操作必须在IF段完成，以便为取下一条指令做好准备。

    （需设置一个专门的加法器）
  - 但分支指令也可能改变PC的值，而且是在MEM段进行，这会导致冲突。

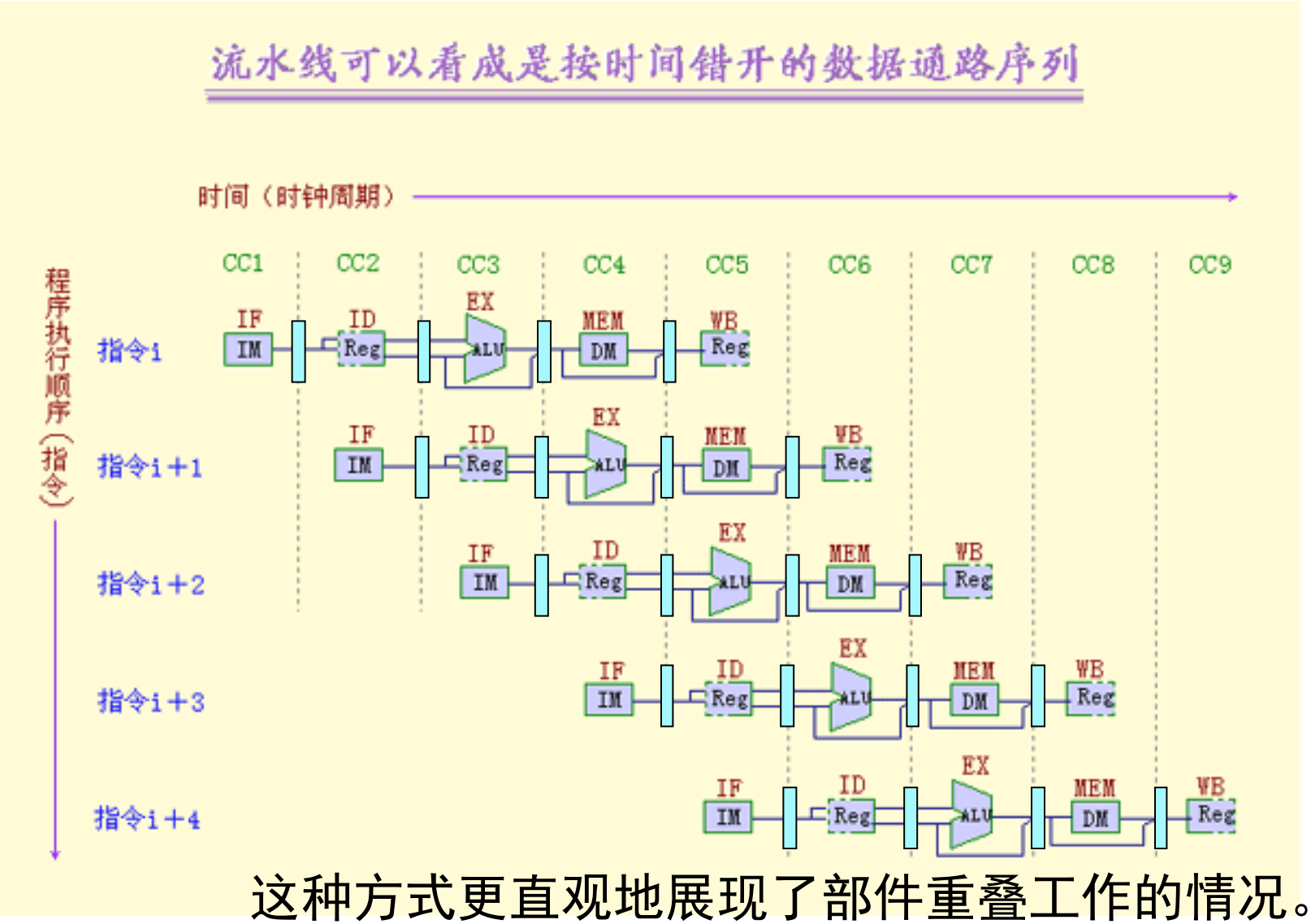
请考虑一下，如何处理分支指令？

4. 5段流水线的两种描述方式

- 第一种描述（类似于时空图）

	时钟周期								
指令编号	1	2	3	4	5	6	7	8	9
指令i	IF	ID	EX	MEM	WB				
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

- 第二种描述（按时间错开的数据通路序列）



这种方式更直观地展现了部件重叠工作的情况。

3.4.2 相关与流水线冲突

3.4.2.1 相关

- 相关(Dependence)：两条指令之间存在某种依赖关系。

  如果两条指令相关，则它们就有可能不能在流水线中重叠执行或者只能部分重叠执行。

- 相关有3种类型
  - 数据相关（也称真数据相关， Data Dependence ）
  - 名相关（Name Dependence）
  - 控制相关（Control Dependence）

1. 数据相关

- 对于两条指令i（在前，下同）和j（在后，下同），如果下述条件之一成立，则称指令j与指令i数据相关。
  - 指令j使用指令i产生的结果；
  - 指令j与指令k数据相关，而指令k又与指令i数据相关。----数据相关具有传递性。

数据相关反映了数据的流动关系，

即如何从其产生者流动到其消费者。

例如：下面这一段代码存在数据相关。

```
Loop:  L.D     F0, 0(R1)      // F0为数组元素
       ADD.D   F4, F0, F2     // 加上F2中的值
       S.D     F4, 0(R1)      // 保存结果
       DADDIU  R1, R1, -8     // 数组指针递减8个字节
       BNE     R1, R2, Loop   // 如果R1≠R2，则分支
```

- 当数据的流动是经过寄存器时，相关的检测比较直观和容易。
- 当数据的流动是经过存储器时，检测比较复杂。
  - 相同形式的地址其有效地址未必相同；
  - 形式不同的地址其有效地址却可能相同。

2. 名相关

- 名：指令所访问的寄存器或存储器单元的名称。
- 如果两条指令使用相同的名，但是它们之间并没有数据流动，则称这两条指令存在名相关。
- 指令j与指令i之间的名相关有两种：
  - 反相关：如果指令j写的名与指令i读的名相同，则称指令i和j发生了反相关。

    指令j写的名＝指令i读的名
  - 输出相关：如果指令j和指令i写相同的名，则称指令i和j发生了输出相关。

    指令j写的名＝指令i写的名

- 名相关的两条指令之间并没有数据的传送。
- 如果一条指令中的名改变了，并不影响另外一条指令的执行。

➢ 换名技术（Renaming）

- 换名技术：通过改变指令中操作数的名来消除名相关。
- 对于寄存器操作数进行换名称为寄存器换名。

既可以用编译器静态实现，也可以用硬件动态完成。

例如：考虑下述代码：

```
DIV.D    F2，F8，F4
ADD.D    F8，F0，F12
SUB.D    F10，F8，F14
```

DIV.D和ADD.D存在反相关，

进行寄存器换名（F8换成S）后，变成：

```
DIV.D    F2，F8，F4
ADD.D    S，F0，F12
SUB.D    F10，S，F14
```

3. 控制相关

- 控制相关是指由分支指令引起的相关。
  - 为了保证程序应有的执行顺序，必须严格按控制相关确定的顺序执行。
- 典型的程序结构是“if-then”结构。
- 请看一个示例：

```
if p1 {
        S1;
    };
S;
if p2 {
        S2;
    };
```

S1与p1控制相关，S2与p2控制相关，S与p1和p2均无关。

- 控制相关带来了以下两个限制：
  - 与一条分支指令控制相关的指令不能被移到该分支之前。否则这些指令就不受该分支控制了。

    对于上述的例子，then 部分中的指令不能移到if语句之前。
  - 如果一条指令与某分支指令不存在控制相关，就不能把该指令移到该分支之后。

对于这个例子，不能把S移到if语句的then 部分中。

```
if p1 {
        S1;
    };
S;
if p2 {
        S2;
    };
```

3.4.2.2 流水线冲突

流水线冲突(Pipeline Hazard)是指对于具体的流水线来说，由于相关的存在，使得指令流中的下一条指令不能在指定的时钟周期执行。

流水线冲突有3种类型：

- 结构冲突：因硬件资源满足不了指令重叠执行的要求而发生的冲突。
- 数据冲突：当指令在流水线中重叠执行时，因需要用到前面指令的执行结果而发生的冲突。
- 控制冲突：流水线遇到分支指令和其它会改变PC值的指令所引起的冲突。

带来的几个问题：

- 导致错误的执行结果。
- 流水线可能会出现停顿，从而降低流水线的效率和实际的加速比。

基本解决方法：暂停部分指令执行。

当一条指令被暂停时，在该暂停指令之后流出的所有指令都要被暂停，而在该暂停指令之前流出的指令则继续进行（否则就永远无法消除冲突）。

1. 结构冲突

- 在流水线处理机中，为了能够使各种组合的指令都能顺利地重叠执行，需要对功能部件进行流水或重复设置资源。
- 如果某种指令组合因为资源冲突而不能正常执行，则称该处理机有结构冲突。
- 常见的导致结构冲突的原因：
  - 功能部件不是完全流水
  - 资源份数不够

- 结构冲突举例：访存冲突

有些流水线处理机只有一个存储器，将数据和指令放在一起，访存指令会导致访存冲突。

- 解决办法Ⅰ：插入暂停周期

（“流水线气泡”或“气泡”）

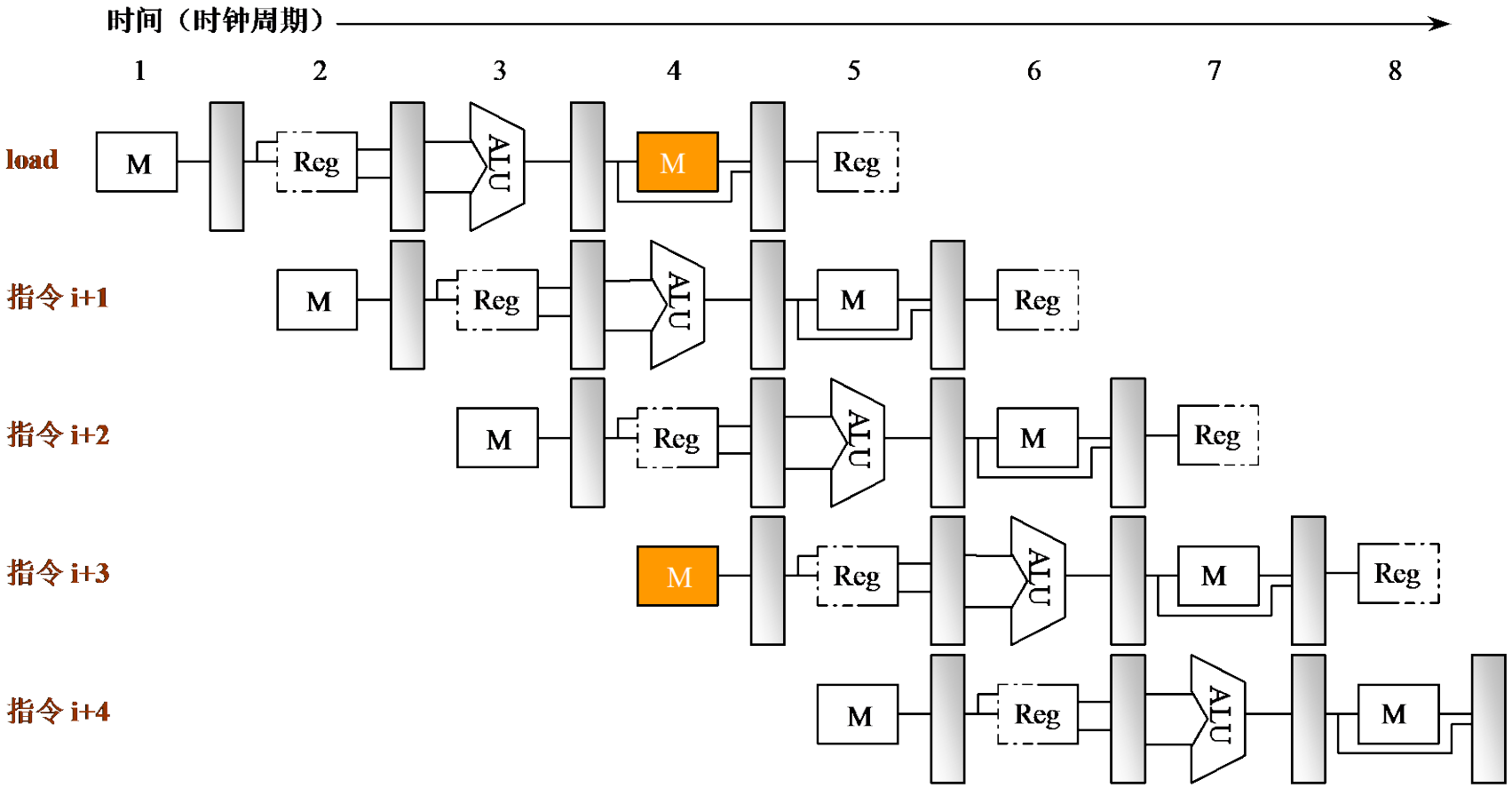
引入暂停后的时空图

- 解决方法Ⅱ：

设置相互独立的指令存储器和数据存储器

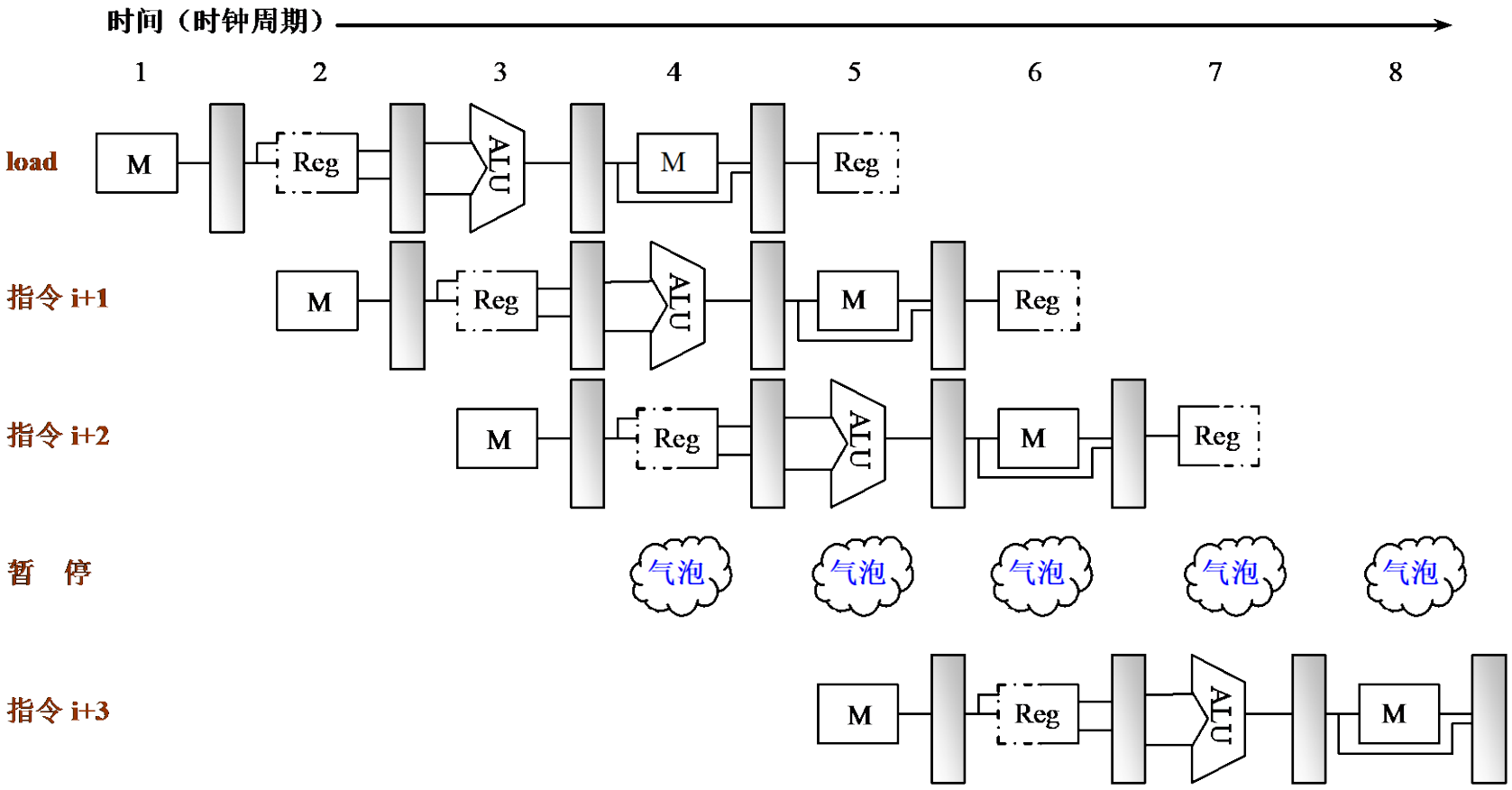
或设置相互独立的指令Cache和数据Cache。

3.4 流水线的相关与冲突



由于访问同一个存储器而引起的结构冲突

3.4 流水线的相关与冲突



为消除结构冲突而插入的流水线气泡

引入暂停后的时空图

指令编号	时钟周期									
	1	2	3	4	5	6	7	8	9	10
指令i	IF	ID	EX	MEM	WB					
指令i+1		IF	ID	EX	MEM	WB				
指令i+2			IF	ID	EX	MEM	WB	WB		
指令i+3				stall	IF	ID	EX	MEM	WB	
指令i+4						IF	ID	EX	MEM	WB
指令i+5							IF	ID	EX	MEM

有时流水线设计者允许结构冲突的存在

主要原因：减少硬件成本

如果把流水线中的所有功能单元完全流水化，或者重复设置足够份数，那么所花费的成本将相当高。

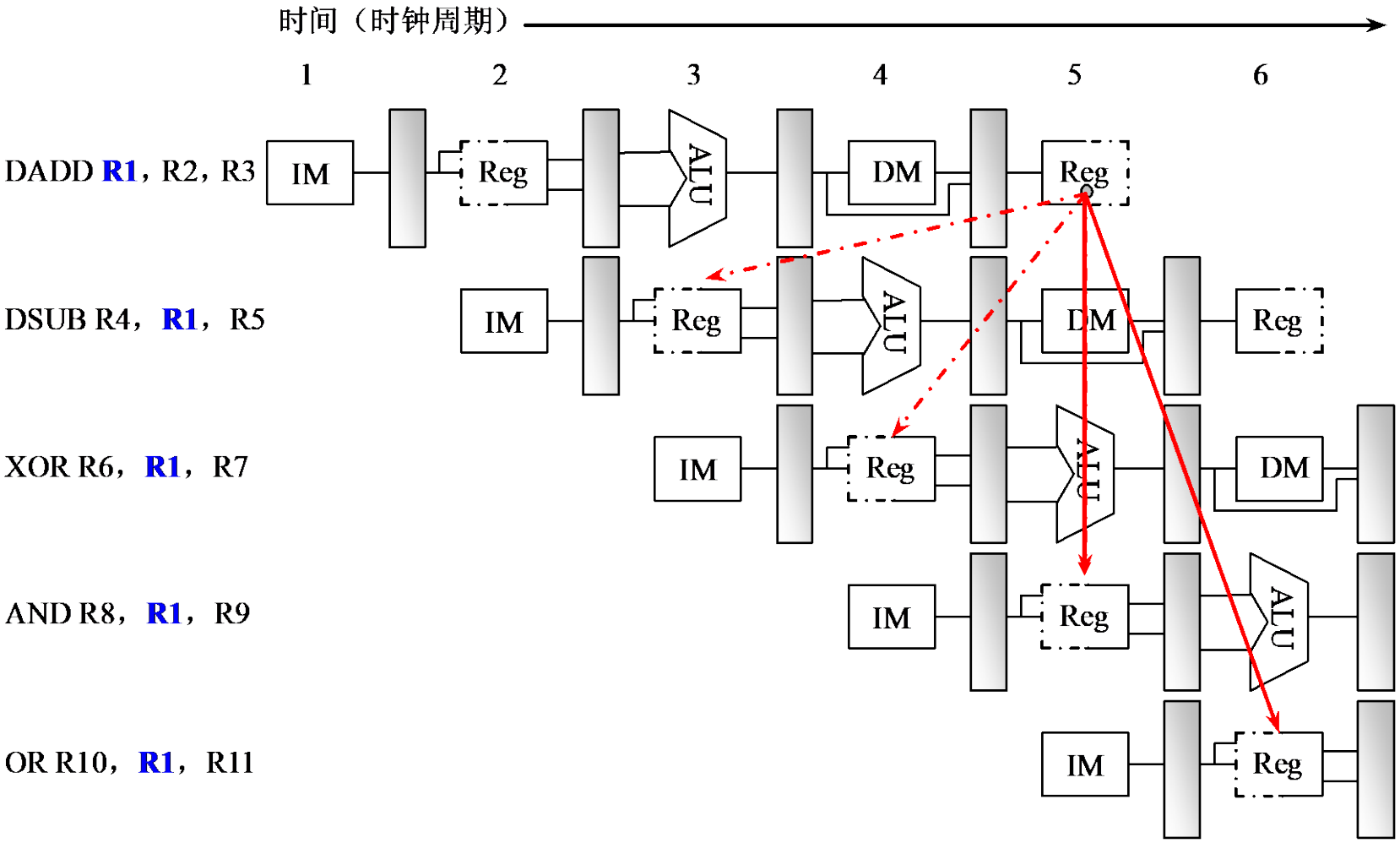
2. 数据冲突

当相关的指令靠得足够近时，它们在流水线中的重叠执行或者重新排序会改变指令读/写操作数的顺序，使之不同于它们串行执行时的顺序，则发生了数据冲突。

举例：

```
DADD  R1，R2，R3
DSUB  R4，R1，R5
XOR   R6，R1，R7
AND   R8，R1，R9
OR    R10，R1，R11
```

3.4 流水线的相关与冲突



流水线的数据冲突举例

- 根据指令读访问和写访问的顺序，可以将数据冲突分为3种类型。

考虑两条指令i和j，且i在j之前进入流水线，可能发生的数据冲突有：

- 写后读冲突（RAW，Read After Write）

  在 i 写入之前，j 先去读。j 读出的内容是错误的。这是最常见的一种数据冲突，它对应于真数据相关。

- 写后写冲突（WAW，Write After Write）

  在 i 写入之前，j 先写。最后写入的结果是 i 的。这种冲突对应于输出相关。

  写后写冲突仅发生在这样的流水线中：

  - 流水线中不只一个段可以进行写操作；
  - 指令被重新排序了。

前面介绍的5段流水线不会发生写后写冲突。（因只在WB段写寄存器）

- 读后写冲突（WAR，Write After Read）

  在 i 读之前，j 先写。i 读出的内容是错误的！

  由反相关引起。

  这种冲突仅发生在这样的情况下：

  - 有些指令的写结果操作提前了，而且有些指令的读操作滞后了；
  - 指令被重新排序了。

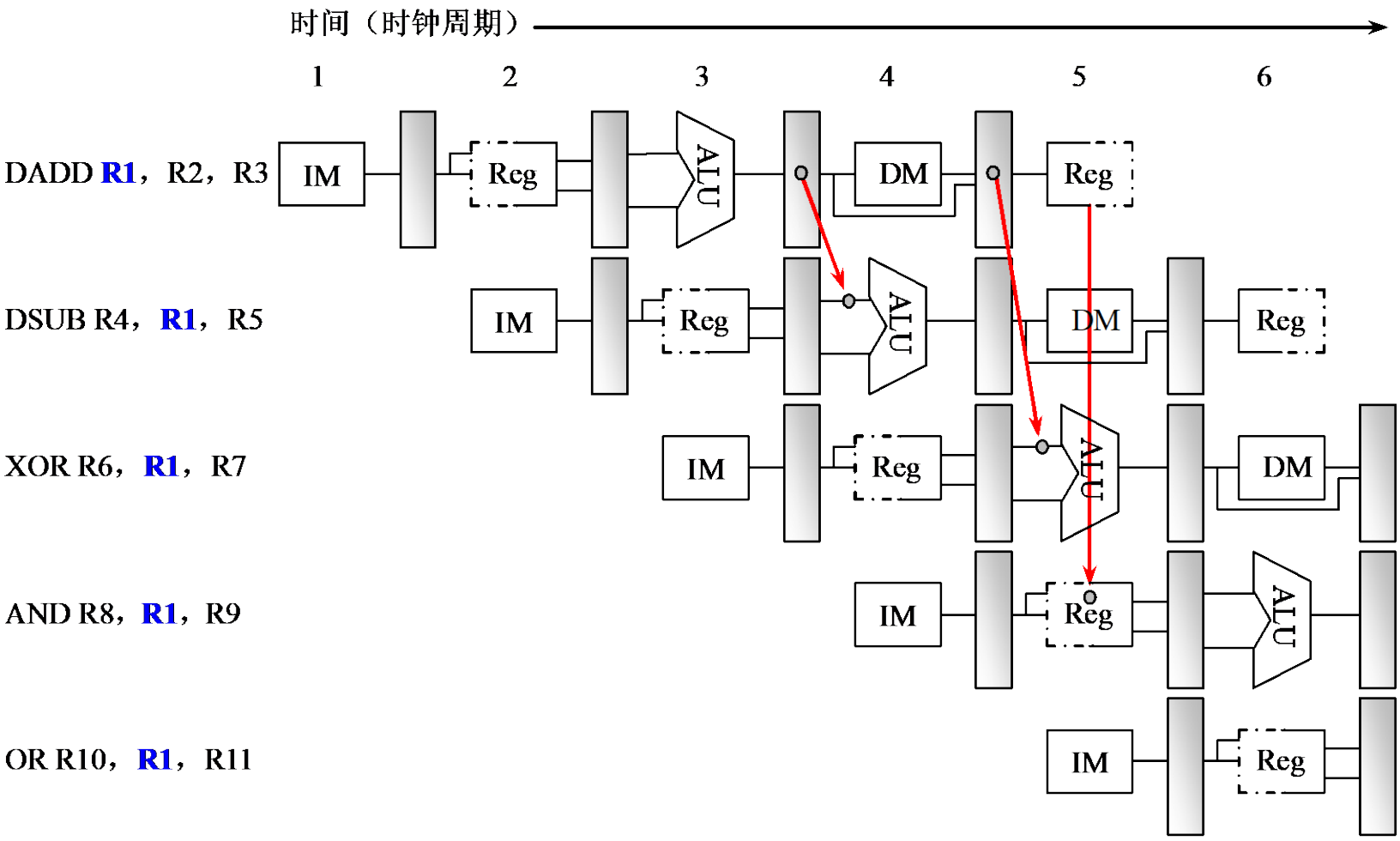
- 通过定向技术（Forwarding）减少数据冲突引起的停顿

  （定向技术也称为旁路或短路）

  - 关键思想：在计算结果尚未出来之前，后面等待使用该结果的指令并不真正立即需要该计算结果，如果能够将该计算结果从其产生的地方直接送到其它指令需要它的地方，那么就可以避免停顿。

  - 采用定向技术消除上例中的相关

3.4 流水线的相关与冲突



采用定向技术后的流水线数据通路

- 定向的实现
  - EX段和MEM段之间的流水寄存器中保存的ALU运算结果总是回送到ALU的入口。
  - 当定向硬件检测到前一个ALU运算结果写入的寄存器就是当前ALU操作的源寄存器时，那么控制逻辑就选择定向的数据作为ALU的输入，而不采用从通用寄存器组读出的数据。
- 结果数据不仅可以从某一功能部件的输出定向到其自身的输入，而且还可以定向到其它功能部件的输入。

- 需要停顿的数据冲突
  - 并不是所有的数据冲突都可以用定向技术来解决。

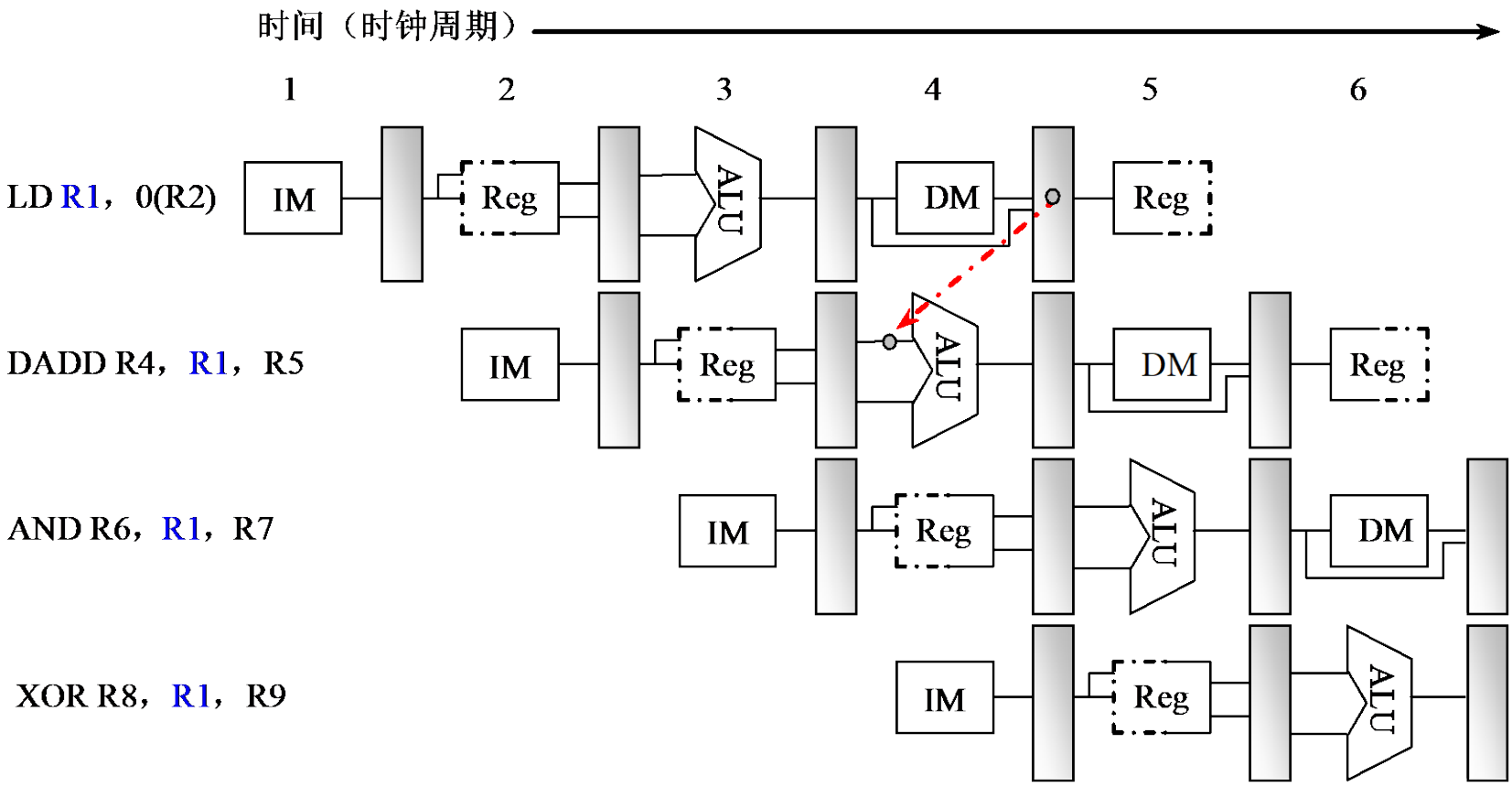
举例:

```
LD    R1，0（R2）
DADD  R4，R1，R5
AND   R6，R1，R7
XOR   R8，R1，R9
```

  - 增加流水线互锁机制（Pipeline Interlock），插入“暂停”。

作用：检测发现数据冲突，并使流水线停顿，直至冲突消失。

3.4 流水线的相关与冲突



无法将LD指令的结果定向到DADD指令

- 依靠编译器解决数据冲突

  让编译器重新组织指令顺序来消除冲突，这种技术称为指令调度（**Instruction Scheduling**）或流水线调度（**Pipeline Scheduling**）。

- 举例：

请为下列表达式生成没有暂停的指令序列：

A＝B＋C ；

D＝E－F ；

假设载入延迟为1个时钟周期。

调度前的代码			调度后的代码		
LD	Rb,	B	LD	Rb,	B
LD	Rc,	C	LD	Rc,	C
DADD	Ra, Rb,	Rc	LD	Re,	E
SD	Ra,	A	DADD	Ra, Rb,	Rc
LD	Re,	E	LD	Rf,	F
LD	Rf,	F	SD	Ra,	A
DSUB	Rd, Re,	Rf	DSUB	Rd, Re,	Rf
SD	Rd,	D	SD	Rd,	D

停顿

停顿

3. 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。
- 处理分支指令最简单的方法：
  “冻结（Freeze）”或者“排空（Flush）”流水线

  优点：简单
  - 前述5段流水线中，改变PC值是在MEM段进行的。
    给流水线带来了3个时钟周期的延迟

简单处理分支指令：分支成功的情况

分支指令	IF	ID	EX	MEM	WB					
分支目标指令		IF	stall	stall	IF	ID	EX	MEM	WB	
分支目标指令+1						IF	ID	EX	MEM	WB
分支目标指令+2							IF	ID	EX	MEM
分支目标指令+3								IF	ID	EX

这个5段流水线中，ID段检测到分支指令，暂停其后所有指令的执行，直到MEM段，确定新PC值。

给流水线带来了3个时钟周期的延迟

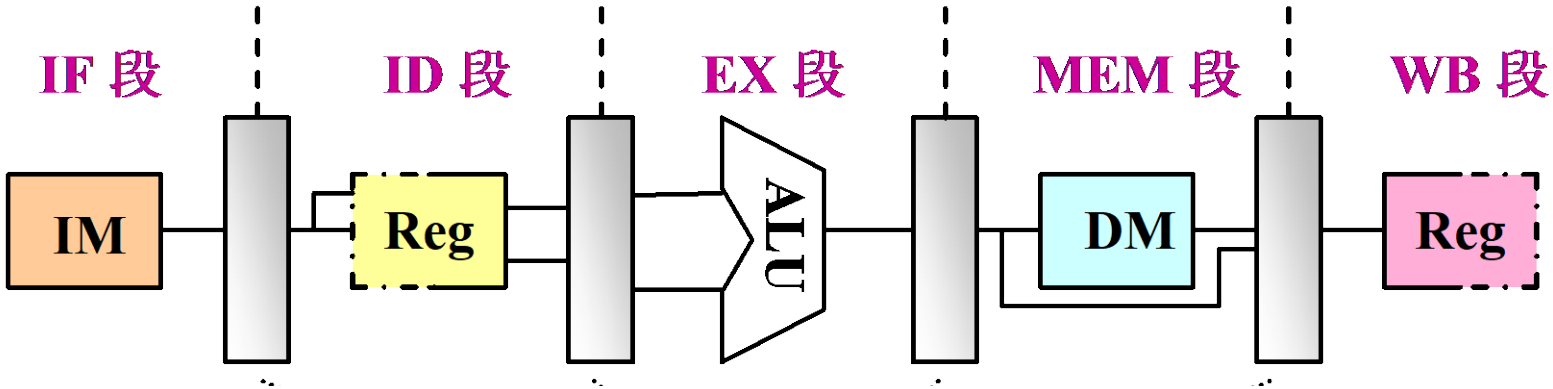
- 把由分支指令引起的延迟称为分支延迟。
- 分支指令在目标代码中出现的频度
  - 每3～4条指令就有一条是分支指令。

  假设：分支指令出现的频度是30％

  流水线理想 CPI＝1

  那么：流水线的实际 CPI ＝ 70%*1+30%*（1+3）=1.9

- 可采取两种措施来减少分支延迟。
  - 在流水线中尽早判断出分支转移是否成功；
  - 尽早计算出分支目标地址。



分支指令要做两件事：

判断分支是否成功+分支目标地址

MEM段 =》 ID段

下面的讨论中，我们假设：

这两步工作被提前到ID段完成，即分支指令是在ID段的末尾执行完成，所带来的分支延迟为一个时钟周期。

- 3种通过软件（编译器）来减少分支延迟的方法

共同点：

- 对分支的处理方法在程序的执行过程中始终是不变的，是静态的。
- 要么总是预测分支成功，要么总是预测分支失败。

- 预测分支失败
  - 允许分支指令后的指令继续在流水线中流动，就好象什么都没发生似的；
  - 若确定分支失败，将分支指令看作是一条普通指令，流水线正常流动；
  - 若确定分支成功，流水线就把在分支指令之后取出的所有指令转化为空操作，并按分支目地重新取指令执行。

要保证：分支结果出来之前不能改变处理机的状态，以便一旦猜错时，处理机能够回退到原先的状态。

DLX流水线对分支的处理过程

分支指令 i(失败)	IF	ID	EX	MEM	WB				
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

分支指令 i(成功)	IF	ID	EX	MEM	WB				
指令 i+1		IF	ID	idle	idle	idle			
分支目标 j			IF	ID	EX	MEM	WB		
分支目标 j+1				IF	ID	EX	MEM	WB	
分支目标 j+2					IF	ID	EX	MEM	WB

预测分支失败

- 预测分支成功

假设分支转移成功，并从分支目标地址处取指令执行。

起作用的前题：先知道分支目标地址，后知道分支是否成功。

前述5段流水线中，由于判断分支是否成功与分支目标地址计算是在同一流水段完成的，所以这种方法没有任何好处。

- 延迟分支（Delayed Branch）

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。

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”了流水线原来必需插入的暂停周期。

分支延迟指令的调度

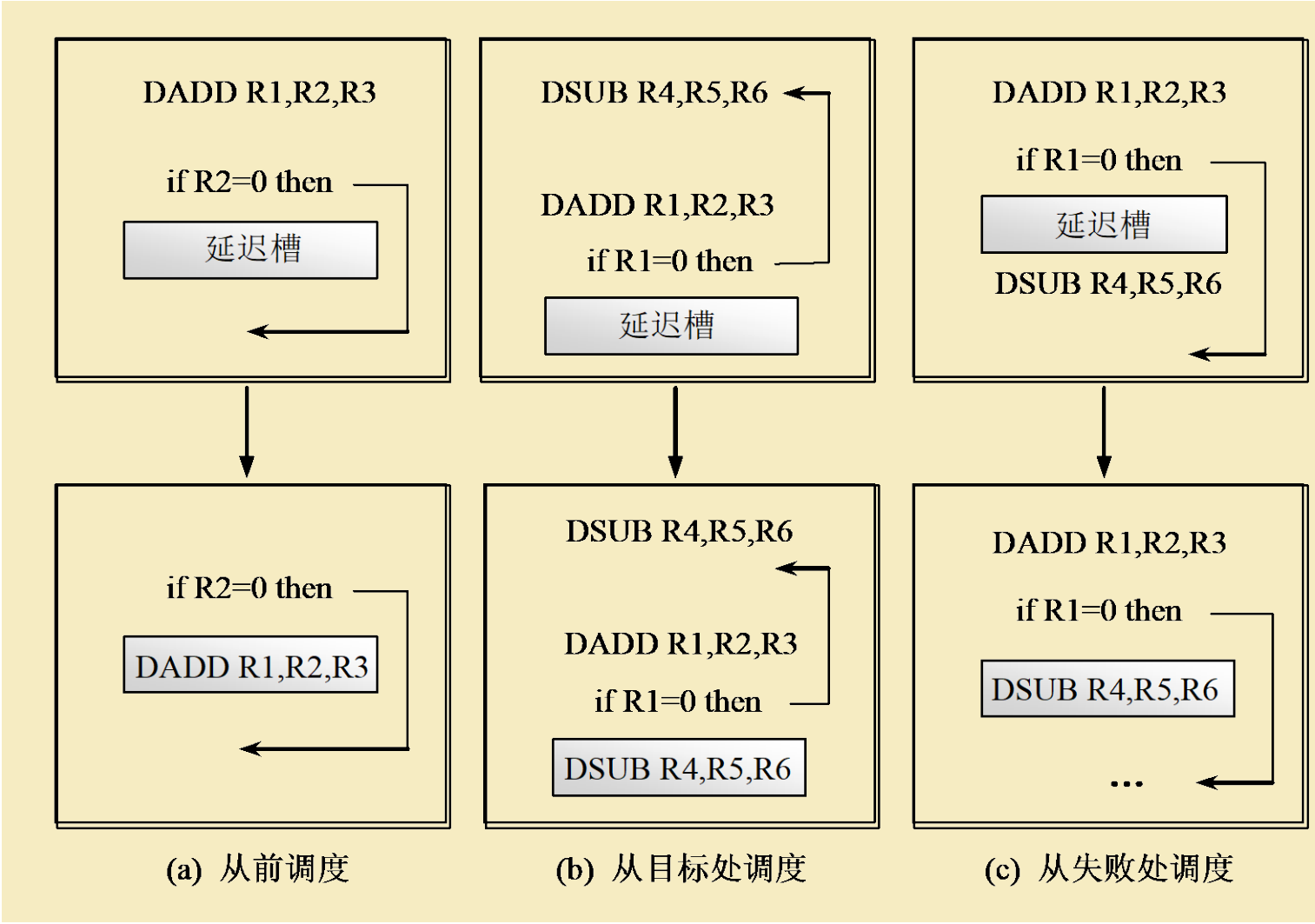
任务：在延迟槽中放入有用的指令

由编译器完成。能否带来好处取决于编译器能否把有用的指令调度到延迟槽中。

三种调度方法：

- 从前调度
- 从目标处调度
- 从失败处调度

调度前和调度后的代码


DADD R1,R2,R3
if R2=0 then
延迟槽
DSUB R4,R5,R6
DADD R1,R2,R3
if R1=0 then
延迟槽
DADD R1,R2,R3
if R1=0 then
延迟槽
DSUB R4,R5,R6
if R2=0 then
DADD R1,R2,R3
DSUB R4,R5,R6
DADD R1,R2,R3
if R1=0 then
DSUB R4,R5,R6
DADD R1,R2,R3
if R1=0 then
DSUB R4,R5,R6
...
(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（Canceling or Nullifying）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。

“预测成功-取消”分支的执行过程

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

3.5 流水线的实现

3.5.1 MIPS的一种简单实现

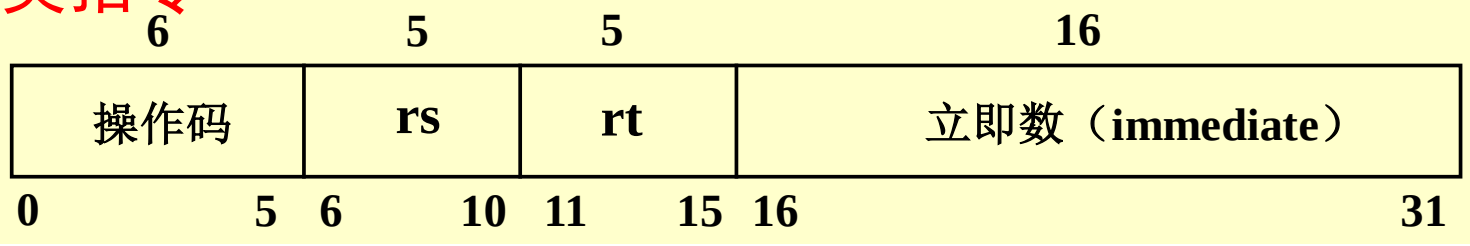
1. 实现MIPS指令子集的一种简单数据通路。
   - 该数据通路的操作分成5个时钟周期
     - 取指令（IF）
     - 指令译码/读寄存器（ID）
     - 执行/有效地址计算（EX）
     - 存储器访问/分支完成（MEM）
     - 写回 （WB）
   - 只讨论整数指令的实现

     （包括：load和store，等于0转移，整数ALU指令等。）

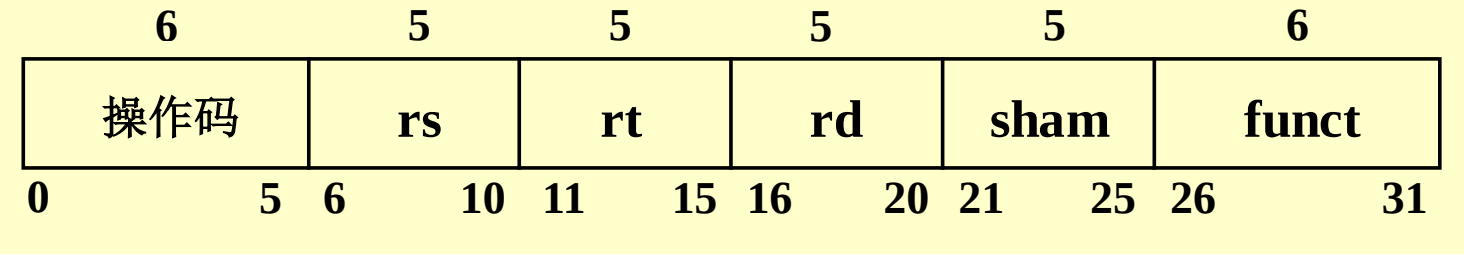
MIPS的指令格式

操作码字段以及rs、rt字段都是在固定的位置。
这种技术称为固定字段译码技术。

I类指令

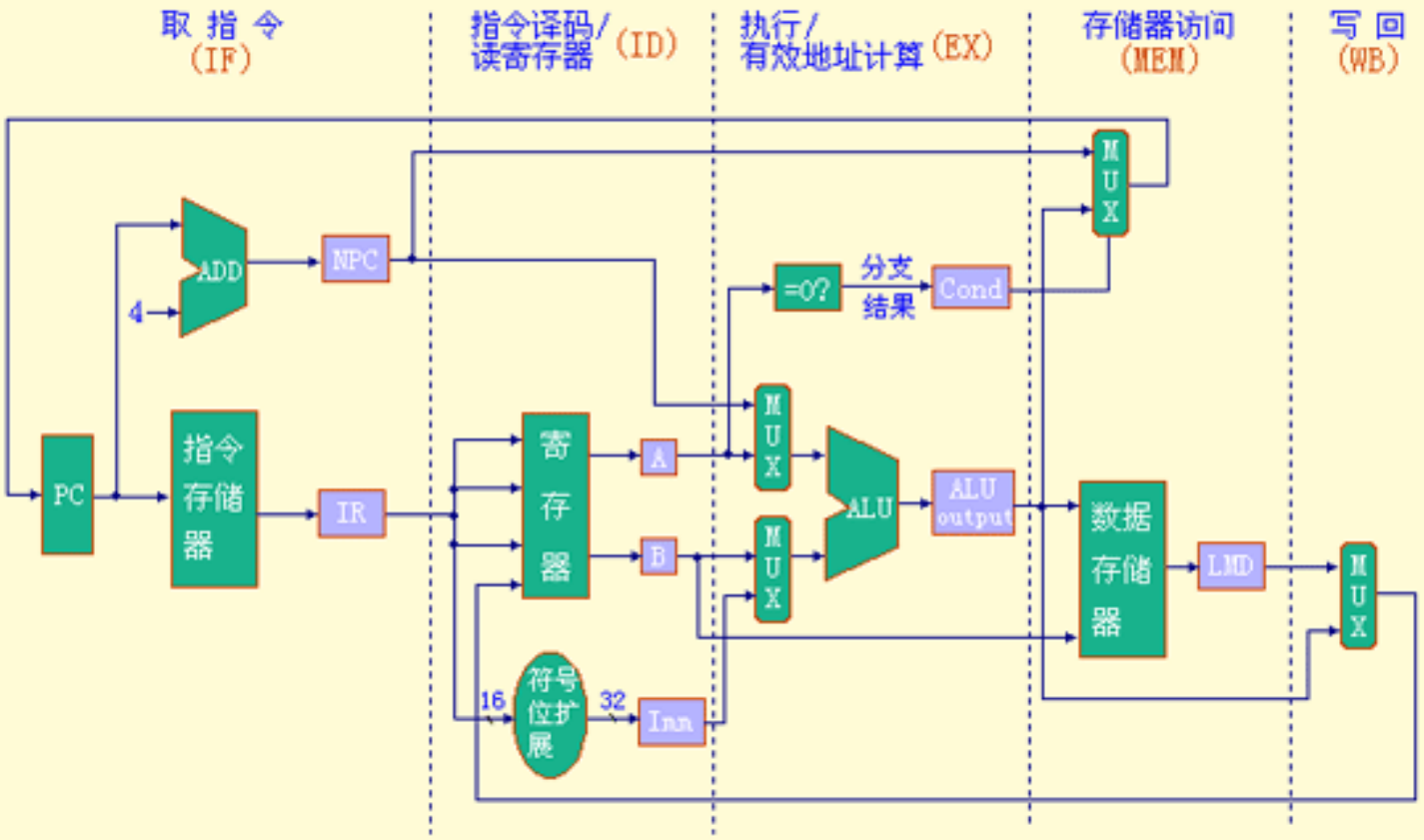


R类指令



J类指令

实现DLX指令的一种简单数据通路



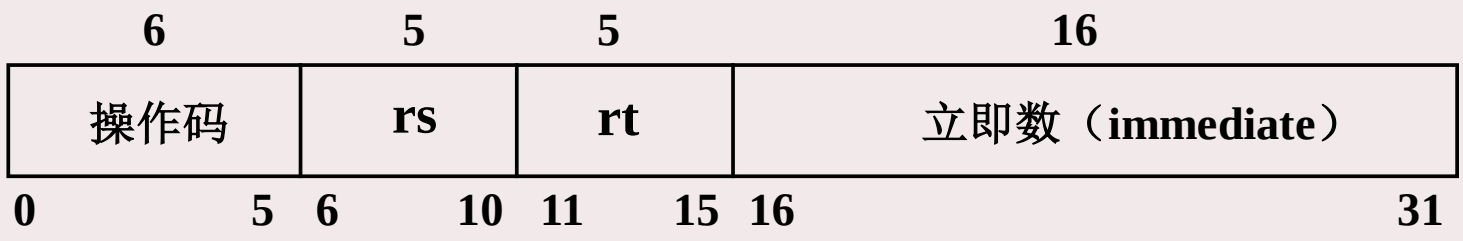
- 通用寄存器：指令之间保存结果。如PC、通用寄存器组、存储器单元等。

- 临时寄存器：在单条指令的执行过程中保存中间结果。
  - **NPC**：下一条程序计数器，存放下一条指令的地址。
  - **IR**：指令寄存器，存放当前正在处理的指令。
  - **A**：第一操作数寄存器，存放从通用寄存器组读出来的操作数。
  - **B**：第二操作数寄存器，存放从通用寄存器组读出来的另一个操作数。
  - **Imm**：存放符号扩展后的立即数操作数。
  - **Cond**：存放条件判定的结果。为“真”表示分支成功。
  - **ALUo**：存放**ALU**的运算结果。
  - **LMD**：存放**load**指令从存储器读出的数据。

2. 一条MIPS指令最多需要以下5个时钟周期：

- 取指令周期（IF）
  - IR←Mem[PC]
  - NPC←PC+4
- 指令译码/读寄存器周期（ID）
  - A ← Regs[rs]
  - B ← Regs[rt]
  - Imm ← （（IR_{16}）^{16}##$IR_{16..31}$） IR的低16位符号扩展

I类指令



指令的译码操作和读寄存器操作是并行进行的。

原因：在MIPS指令格式中，操作码字段以及rs、rt字段都是在固定的位置。这种技术称为固定字段译码技术。

- 执行/有效地址计算周期（EX）

  不同指令所进行的操作不同：

  - load指令和store指令

    ALUo←A + Imm

  - 寄存器－寄存器ALU指令

    ALUo←A funct B

  - 寄存器－立即值ALU指令

    ALUo←A op Imm

  - 分支指令

    ALUo←NPC+（Imm<<2）；

    cond←（A = = 0）

将有效地址计算周期和执行周期合并为一个时钟周期，这是因为MIPS指令集采用load / store结构，没有任何指令需要同时进行数据有效地址的计算、转移目标地址的计算和对数据进行运算。

- 存储器访问/分支完成周期（MEM）
  - 所有指令都要在该周期对PC进行更新。

    除了分支指令，其它指令都是做：PC←NPC
  - 在该周期内处理的MIPS指令仅仅有load、store和分支三种指令。

  - load指令和store指令

    LMD←Mem[ALUo]

    或者 Mem[ALUo]←B
  - 分支指令

    if （cond） PC ←ALUo else PC←NPC

- 写回周期（WB）

  不同的指令在写回周期完成的工作也不一样。

  - 寄存器－寄存器ALU指令

    Regs[rd]← ALUo

  - 寄存器－立即数ALU指令

    Regs[rt]← ALUo

  - load指令

    Regs[rt]← LMD

3.5.2 基本的MIPS流水线

每一个时钟周期完成的工作看作是流水线的一段，每个时钟周期启动一条新的指令。

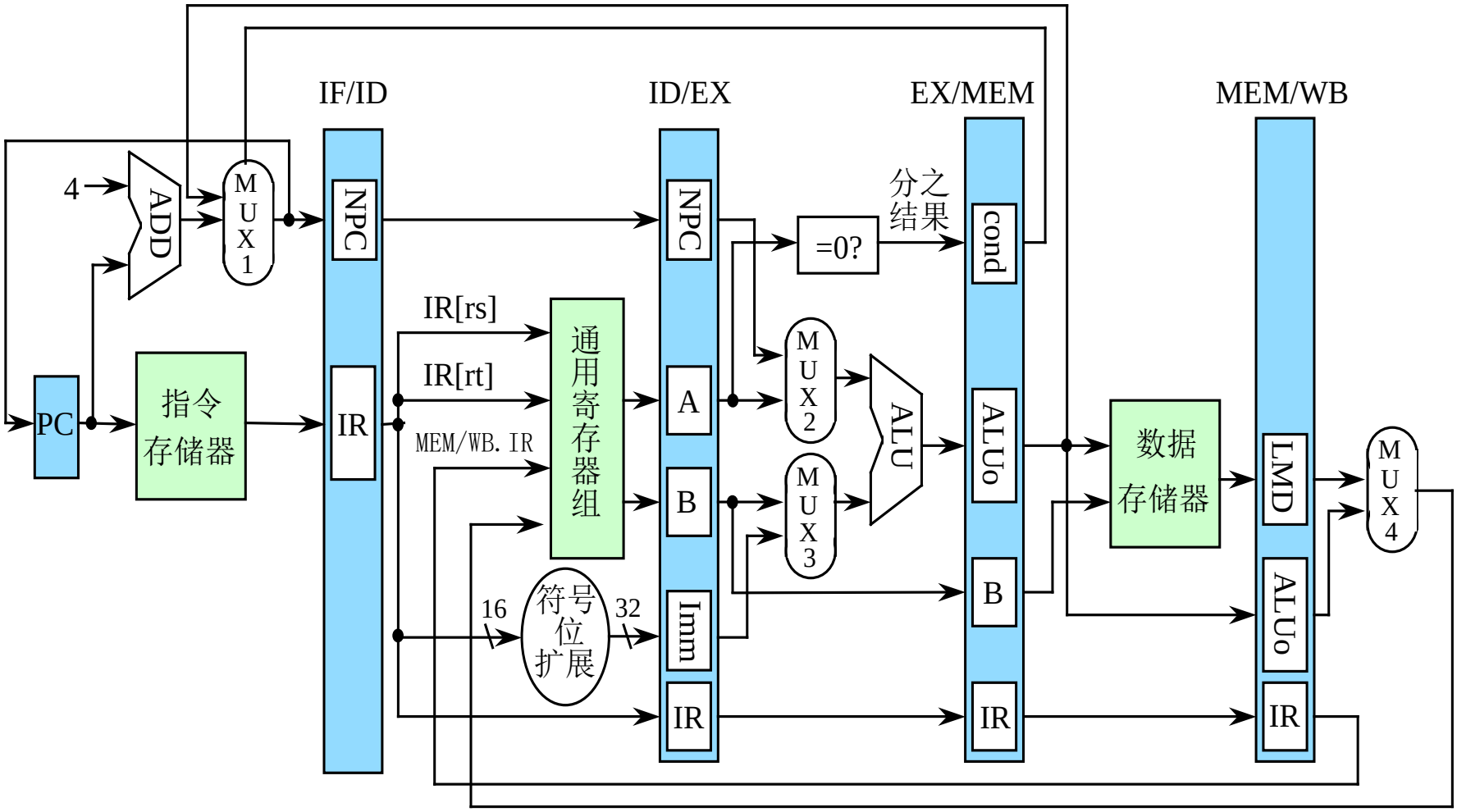
1. 流水实现的数据通路主要做了以下改动：

- 设置了流水寄存器
  - 段与段之间设置流水寄存器
  - 流水寄存器的名称

    用其相邻的两个段的名称拼合而成。

    例如：ID段与EX段之间的流水寄存器用ID/EX表示
  - 每个流水寄存器是由若干个寄存器构成的

3.5 流水线的实现



流水实现的数据通路

- 寄存器的命名形式为：x.y
- 所包含的字段的命名形式为：x.y[s]

  其中：x：流水寄存器名称

  y：具体寄存器名称

  s：字段名称

  例如：

  ID/EX.IR：流水寄存器ID/EX中的子寄存器IR

  ID/EX.IR[op]：该寄存器的op字段（即操作码字段）
- 流水寄存器的作用
  - 将各段的工作隔开，使得它们不会互相干扰。
  - 保存相应段的处理结果。

例如：

EX/MEM.ALUo：保存EX段ALU的运算结果

MEM/WB.LMD：保存MEM段从数据存储器读出的数据

- 向后传递后面将要用到的数据或者控制信息

所有有用的数据和控制信息每个时钟周期会随着指令在流水线中的流动往后流动一段。

- 增加了向后传递IR和从MEM/WB.IR回送到通用寄存器组的连接。

- 将对PC的修改移到了IF段，以便PC能及时地加4，为取下一条指令做好准备。

2. 每一个流水段进行的操作

- $IR[rs] = IR_{6..10}$
- $IR[rt] = IR_{11..15}$
- $IR[rd] = IR_{16..20}$

流水线的每个流水段的操作

流水段	所有指令类型		
IF	IF/ID.IR ← Mem[PC] IF/ID.NPC, PC ← （if （（EX/MEM.IR[op] == branch ）& EX/MEM.cond）{EX/MEM.ALUo} else {PC+4}）;		
ID	ID/EX.A ← Regs[IF/ID.IR[rs]]; ID/EX.B ← Regs[IF/ID.IR[rt]]; ID/EX.NPC ← IF/ID.NPC; ID/EX.IR ←IF/ID.IR; ID/EX.Imm ← （$IF/ID.IR_{16}$）^{16}##$IF/ID.IR_{16..31}$;		
EX	ALU 指令	load/store 指令	分支指令
	EX/MEM.IR ← ID/EX.IR; EX/MEM.ALUo ← ID/EX.A *funct* ID/EX.B 或 EX/MEM.ALUo ← ID/EX.A *op* ID/EX.Imm;	EX/MEM.IR ← ID/EX.IR; EX/MEM.ALUo ← ID/EX.A + ID/EX.Imm; EX/MEM.B←ID/EX.B;	EX/MEM.IR ← ID/EX.IR; EX/MEM.ALUo ← ID/EX.NPC + ID/EX.Imm<<2; EX/MEM.cond ← （ID/EX.A ==0）;

流水线的每个流水段的操作

流水段	任何指令类型		
	ALU 指令	load/store 指令	分支指令
MEM	MEM/WB.IR ←EX/MEM.IR; MEM/WB.ALUo ← EX/MEM.ALUo;	MEM/WB.IR ← EX/MEM.IR; MEM/WB.LMD ← Mem[EX/MEM.ALUo]; 或 Mem[EX/MEM.ALUo] ← EX/MEM.B;	
WB	Regs[MEM/WB.IR[rd]] ← MEM/WB.ALUo; 或 Regs[MEM/WB.IR[rt]] ← MEM/WB.ALUo;	Regs[MEM/WB.IR[rt]] ← MEM/WB.LMD;	

3. 流水线的控制

- 主要是如何控制四个多路选择器MUX。
  - **MUX2**

if （ID/EX.IR[op]==“分支指令”）

{ MUX2_output＝ID/EX.NPC };

else MUX2_output＝ID/EX.A;

//MUX2_output表示MUX2的输出

  - **MUX3**

if （ID/EX.IR[op]==“寄存器－寄存器型ALU指令”）

{ MUX3_output＝ID/EX.B };

else MUX3_output＝ID/EX.Imm;

//MUX3_output表示MUX3的输出

- **MUX1**

if （（ID/EX.IR[op]==“分支指令”）& EX/MEM.cond）

{ MUX1_output＝EX/MEM.ALUo };

else MUX1_output＝PC+4;

//MUX1_output表示MUX1的输出

- **MUX4**

if （ID/EX.IR[op]==“load”）

{ MUX4_output＝MEM/WB.LMD };

else MUX4_output＝MEM/WB.ALUo

//MUX4_output表示MUX4的输出

- 第5个多路器：从MEM/WB回传至通用寄存器组的写入地址应该是从MEM/WB.IR[rd]和MEM/WB.IR[rt]中选一个。
  - 寄存器－寄存器型ALU指令：选择MEM/WB.IR[rd]；
  - 寄存器－立即数型ALU指令和load指令：选择MEM/WB.IR[rt]。

➢ 解决数据冲突的问题

- 所有的数据冲突均可以在ID段检测到。

  如果存在数据冲突，就在相应的指令流出ID段之前将之暂停。

  完成该工作的硬件称为流水线的互锁机制。

- 在ID段确定需要什么样的定向，并设置相应的控制。

  降低流水线的硬件复杂度。（不必挂起已经改变了机器状态的指令）

- 也可以在使用操作数的那个时钟周期的开始检测冲突和确定必需的定向。
- 检测冲突是通过比较寄存器地址是否相等来实现的。

举例： load互锁

由于使用load的结果而引起的流水线互锁称为load互锁。

在ID段检测是否存在RAW冲突
（这时load指令在EX段）

ID/EX中的操作码 （ID/EX.IR[op]）	IF/ID中的操作码 （IF/ID.IR[op]）	比较的操作数字段
load	RR ALU	ID/EX.IR[rt]=IF/ID.IR[rs]
load	RR ALU	ID/EX.IR[rt]=IF/ID.IR[rt]
load	load、store ALU立即数或分支	ID/EX.IR[rt]=IF/ID.IR[rs]

- 若检测到RAW冲突，流水线互锁机制必须在流水线中插入停顿，并使当前正处于IF段和ID段的指令不再前进。
  - 将ID/EX.IR中的操作码改为全0
    （全0表示空操作）
  - IF/ID寄存器的内容回送到自己的入口

➢ 定向逻辑

- 要考虑的情况更多
- 通过比较流水寄存器中的寄存器地址来确定

例如：

- 若：（ID/EX.IR.op==RR ALU)&（EX/MEM.IR.op==RR ALU）&（ID/EX.IR[rt]==EX/MEM.IR[rd]）

  则：EX/MEM.ALUo定向到ALU的下面一个输入

- 若：（ID/EX.IR[op]==RR ALU）&（MEM/WB.IR[op]==load）&（ID/EX.IR[rt]==MEM/WB.IR[rt]）

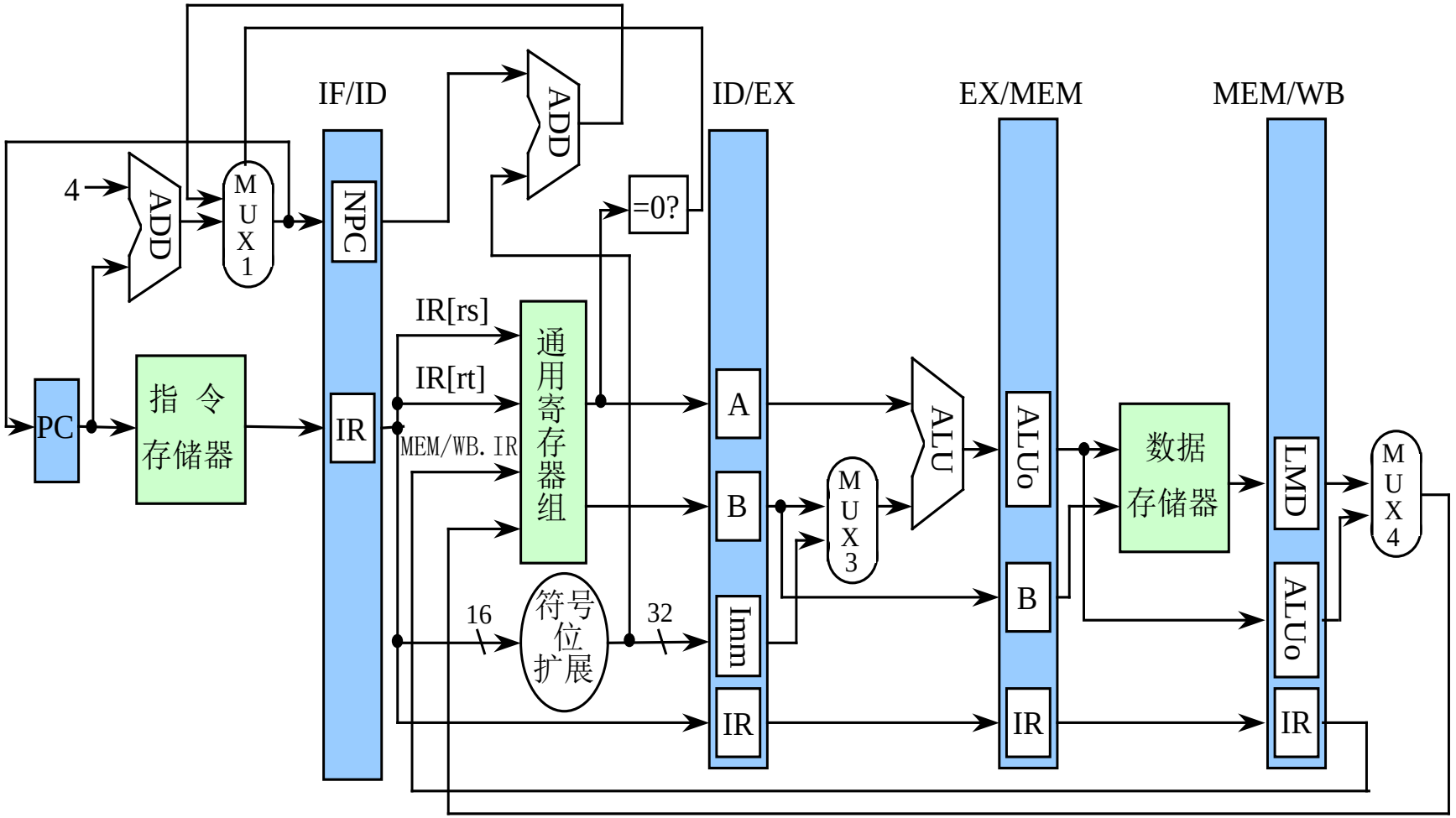
  则：把MEM/WB.LMD定向到ALU的下面一个输入

4. 控制冲突

- 分支指令的条件测试和分支目标地址计算是在EX段完成，对PC的修改是在MEM段完成。
- 它所带来的分支延迟是3个时钟周期。
- 减少分支延迟：

  （把上述工作提前到ID段进行）

  - 在ID段增设一个加法器：计算分支目标地址
  - 把条件测试“=0？”的逻辑电路移到ID段
  - 这些结果直接回送到IF段的MUX1
  - 改进后的流水线对分支指令的处理



为减少分支延迟，改进后的流水线数据通路

改进后流水线的分支操作

流 水 段	分 支 指 令 操 作
IF	IF/ID.IR ← Mem[PC]; IF/ID.NPC, PC ← (if((IF/ID[op]= =branch)&((Regs[IF/ID.IR[rs]] = = 0)) {IF/ID.NPC + $(IF/ID.IR_{16})^{16}$ ## $(IF/ID.IR_{16..31} << 2)$ } else {PC+4});
ID	ID/EX.A ←Regs[IF/ID.IR[rs]]; ID/EX.B← Regs[IF/ID.IR[rt]]; ID/EX.IR ← IF/ID.IR; ID/EX.Imm ← $(IF/ID.IR_{16})^{16}$ ## $IF/ID.IR_{16..31}$;
EX	
MEM	
WB	

小结

3.1 流水线的基本概念

时空图、分类

3.2 流水线的性能指标

吞吐率、加速比和效率

3.3 非线性流水线的调度

预约表、禁止表、冲突向量、状态转移图

小结

3.4 流水线的相关与冲突

5段RISC指令流水线

相关：数据相关、名相关、控制相关

冲突：结构冲突

数据冲突（RAW、WAW、WAR　定向技术）

控制冲突（分支延迟）

调度：分支预测、延迟槽

3.5 流水线的实现

IF、ID、EX、MEM、WB