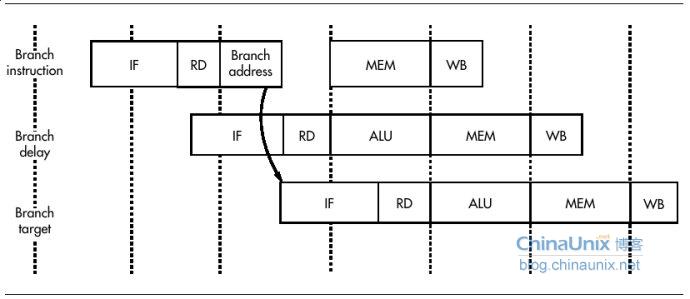
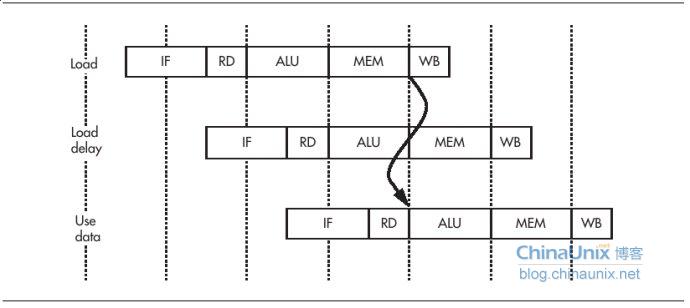
[mips 精确异常和延时槽](http://blog.chinaunix.net/uid-26817832-id-3146408.html)

一、非精确异常  
在多发射乱序执行的流水线 CPU 上，从指令进入流水线到异常事件的发生，期间要经过若干流水级，此时 PC 的值已指向其后的某条指令，在实现非精确异常的 CPU 上就把此时的 PC 值作为引起异常指令的所在（为了表达的方便，记为 eptr）。  
  
简单地说就是 eptr 的指向，并非真正的引起异常的指令之所在，而是其后面的某条指令所在。  
  
二、精确异常  
简单地说就是 eptr 的指向就是真正引起异常的指令之所在  
  
而实现精确异常的 CPU，则在最后指令提交时 (commit) 按指令流的顺序提交，异常的产生也在该指令提交时，这样就能精确计算出引起异常的指令相对于当前 PC 的偏移，从而保证精确异常。  
  
也就是说当异常产生时，之前的预备工作（即取指，译码，当然PC随之增长）便被废弃。CPU从异常中返回时，再重新做读取和译码的工作。  
  
总之，不管是何类异常，eptr 之前的所有指令都会被执行完成 (commit之后)，eptr 之后的指令不会被执行。  
  
三、mips 延迟槽  
  
引入延时槽主要目的就是提高流水线的效率，分为以下两种：  
  
1、分支延时槽  
分支延迟槽 (Branch delay slot)，简单地说就是位于分支指令后面的一条指令，不管分支发生与否其总是被执行，而且通过下面的图可以看出位于分支延迟槽中的指令先于目标指令提交 (commit即执行)。  
.......  
.......  
jal 48 <---- 48 代表跳转的目标地址 jal会置ra寄存器的值  
nop / li a1, 4 <---- 这地方就是一个延迟槽   
....... <----- ra 寄存器的值变为这个地方的地址  
.......  
[](http://blog.chinaunix.net/attachment/201209/19/26817832_1348057914136l.jpg)  
进一步理解：  
流 水线中，分支指令执行时因为确定下一条指令的目标地址（紧随其后 or 跳转目标处？）一般要到第 2 步以后，在目标确定前流水线的取指级是不能工作的，即整个流水线就“浪费”（阻塞）了一个时间片，为了利用这个时间片，在体系结构的层面上规定跳转指令后 面的一个时间片为分支延迟槽（branch delay slot）。位于分支延迟槽中的指令总是被执行，与分支发生与否没有关系。这样就有效利用了一个时间片，消除了流水线的一个“气泡”。  
如下图所示，当我们把延时槽的指令去掉以后，你会发现我们浪费了一个时钟周期。（还需要注意一点就是通过ALU一条特殊路径可以使分支跳转地址提前半个时钟周期获得，加上延时槽指令取指阶段的半个时钟周期刚好为一个时钟周期，填充了我们浪费的一个时钟周期）  
  
说明：延时槽当中的指令有两个用途：  
1）、用来传递子函数调用的参数  
2）、延时槽指令的结果作为跳转指令的条件。  
  
2、存储延时槽  
数据加载指令的数据在下一条指令的ALU阶段启动之后才能从缓存或内存中取得，于是下一条指令不能使用该数据，如果下一条指令强制性的引用该数据，那么该指令先停止运行（在ALU阶段），等这条数据载入指令完成了后再开始运行。  
如下图所示：  
  
[](http://blog.chinaunix.net/attachment/201209/19/26817832_1348057982KTmm.jpg)

四、延迟槽异常返回地址是上一条指令的地址  
简单说，一般CPU的分支跳转指令流是：分支跳转指令->目标跳转地址的指令。  
但MIPS的分支跳转指令流是：分支跳转指令 -> 延时槽指令 -> 目标跳转地址的指令，在中间操作插入了延时槽指令。  
如果PC在延时槽地址中断后，中断返回时返回延时槽指令地址的话，重新执行的指令流为：延时槽指令 -> (延时槽指令地址 + 4)地址的指令，没有跳转了！  
这样完全不是原来被打断的指令流，为了恢复原来的指令流需要将延时槽前面的跳转指令重新装入流水线（会再次执行延时槽中的指令，不过不一定在产生异常了）。  
所以在延时槽中断后返回的地址是前面跳转指令的地址。

zhouxin：若不加载延时槽前面的跳转指令的话，就不能执行跳转指令了。