# 第六讲 流水线处理器

## 601-流水线的基本原理

> 流水线作为一种生产管理的模式，对于
>
> 提高生产效率有着非常大的帮助，最早是兴起于汽车制造厂， 现在已在很多的行业得到了广泛的应用。
>
> 那在处理器设计当中，也借鉴了流水线的概念，以提升性能。
>
> 今天我们就来看一看流水线处理器是如何设计的。

> 这位老朋友大家还记得吧？他曾经为我们展示过精湛的厨艺，
>
> 那今天我们就再来欣赏一下他做菜的过程。他做菜分为这么几步， 首先是洗菜，然后是切菜，
>
> 第三步是炒菜，最后是装盘。我们假设
>
> 这每一步都要花费1分钟，那这位厨师做一道菜就需要4分钟， 那如果他总共做四道菜，就需要16分钟，
>
> 现在呢，他掌勺的这个小餐馆生意非常好，客人很多，但是大家抱怨上菜太慢了，这怎么办呢？
>
> 而我正好是这个餐馆的老板，我就得思考如何解决这个问题。首先我
>
> 发现，在我能请得起的厨师当中，这位大厨已经是动作很快的了， 所以我就想从生产管理模式方面做一些改变。
>
> 那现在当然是非流水线的操作方式，那如果要改造成流水线的操作。
>
> 我们先来看一看这个厨房里有哪些用具， 我们有洗菜用的水池，切菜用的刀和板，炒菜用的厨具，
>
> 还有最后装盘的容器，那在刚才那位大厨的工作过程中，在每一个时刻他都只能使用其中- 的一样，
>
> 另外三样都是空闲的，那看来从硬性条件来说，是适合改造成流水线的操作了。
>
> 那我刚才说了，我这是一个小本经营，我没办法供四个厨师分别来做这四件事情，
>
> 可以从另一个角度来说，如果我雇得起四个这样的大厨，我还不如让 他们同时做四道菜呢，没必要这么麻烦，引入什么流水线的操作。
>
> 那怎么办呢？那我就把这个大厨给解雇了，然后用同样的钱雇来四个人，
>
> 现在大厨很全能，什么都会干，而新雇的这四位每一位只会干一种工作，
>
> 那好，现在人员设备配置到位，那我就希望这四个步骤能够流水化地操作起来。
>
> 他们每一个人在做完自己手头的工作之后，将完成的成果交给下一个阶段，
>
> 继续进行下一步的操作，当然这样的交接需要一个统一的指挥，所以还得有一个发号施令的人，
>
> 这个人是不用新雇的，刚才那位大厨每四分钟做完一道菜，所以
>
> 这位司号员每四分钟吹一次号，以指挥那位大厨去做下一道菜。
>
> 而现在我们假设每个步骤和刚才那个大厨工作的方式一样，
>
> 同样也要花费一分钟的时间，所以这司号员要改成每一分钟吹一次号，

流水线上的工作人员每听到号响，就将自己的工作成果 转交给下一阶段的人，当然，他必须保证在号响之前，

已经顺利地完成了自己的工作。

这个司号员就好比CPU当中的时钟，如果这么来看， 那我们现在的时钟频率已经提升为原来的4倍了。

那我们就来看一看我们新改造的这个厨房是如何工作的， 现在要做的菜单已经送过来了，工作开始。

首先第一道菜的原料送到洗菜的环节， 这时候后面的各个环节都处于空闲状态；

一分钟之后，洗菜完成，第一道菜的原料进入切菜环节， 与此同时，第二道菜的原料进入洗菜环节；

然后再过了一分钟，第一道菜已经切完了，与此同时，第二道菜也洗完了，

当迎来下一次号响的时候，第一道菜就会进入炒菜环节， 第二道菜进入切菜环节，同时第三道菜进入洗菜环节；

然后再过一分钟，四道菜都进入了流水线当中， 这个时候整条流水线的各个环节都开始工作了，

那我们就可以说，这个流水线已经被填满了，而之前的这个过程就是填充流水线的过程；

那再过一分钟，第一道菜就完成了所有的工序，可以上菜了。

而且之后每过一分钟，我们都可以上一道菜， 第2道，第3道，

然后是第4道，7分钟后，这四道菜都完成了。

那因为我们目前的任务只有四道菜，所以在刚才流水线 被填满之后，又经历了一个流水线被排空的过程。

当然如果这餐馆的客人很多，源源不断的有要做的菜单送来，

那这个流水线就可以一直保持着充满的状态，每分钟都能送出一盘菜来。

那我们就来分析一下这个流水线的性能。

现在我们采用这样的流水线的方式，做四道菜用了7分钟，平均每道菜用时不到2分钟，

而且在流水线填满之后，可以做到每一分钟上一道菜。

而之前采用非流水线的方式，是每四分钟才能上一道菜， 那如果我们能保证流水线长期处于填满的状况，

那现在的性能就可以达到原先的4倍，而我们的硬件资源的投入并没有明显的变化。

当然我们要注意的是，采用流水线的方式，虽然可以做到每分钟上一道菜，

但是单独针对某一道菜，其实还是需要4分钟，这个时间并没有缩短。

那看完了这个厨房的例子，我们再来看一看流水线的原理是如何运用到真实的处理器结构上的。

这是我们之前已经构建的单周期的处理器，我们是可以正确地运行一些MIPS的指令的，

而MIPS指令的执行，可以分为这样5个步骤，第一步是取指，

第二步是译码，第三步是执行，第四步是访存，第五步是回写。

那我们还是结合数据通路图来看一看这样的步骤。

相比刚才的结构图，我们做了一点小小的变化，也就把IFU内部的结构进行适度的展开，

对照这个图，取指的阶段，就是用PC的值去访问指令存储器，从而得到指令的编码，

同时还需要生成PC的更新值。在译码阶段，不仅需要把指令编码进行分解，

而且还需要从寄存器堆当中读出所需的寄存器的值。第三步执行，主要在

ALU当中完成，对于算术逻辑运算指令，就是完成对应的运算，

而对于访存指令，则是计算出访存的地址。第四步是访存，对于load指令

是从数据存储器当中读出对应的数据，而对于store指令， 则是将数据送到数据存储器当中去，

而其他指令在这一步没有实质的操作。最后一步是写回，对于要改写

寄存器的指令，在这一步会将数据写入到寄存器堆当中指定的位置，

我们要注意的是，虽然分成了这五步，但只是为了便于描述而已，

所有的信号都必须要在这条指令执行的过程中保持稳定，例如从PC寄存器送到指令存储器的

这个地址信号，如果在指令执行完成前，它就发生了改变，那指令存储器送出的指令编码 也就会发生改变，

从而造成寄存器堆选取了不同编号的寄存器， 送出了不同的值，ALU也可能执行了不同的操作，

那这条指令就可能执行错误了，所以对于单周期处理器来说， 这一条指令执行的过程中，所有的信号都是必须要保持稳定的。

而我们要进行流水线的改造的话， 我们同样也会发现，这不同阶段所用到的硬件资源，基本上是相互独立的。

如果我们能把指令存储器输出的指令编码事先保存下来，

那我们就可以提前更新PC寄存器的值，并用这新的值去指令存储器当中取出一个新的指令，

而在取新指令的同时，刚才取出的那条指令的编码就会被分解成不同位域，

而寄存器堆也会根据输入送出对应寄存器的内容，

所以跟刚才的流水线原理的分析类似，如果我们想把这些硬件资源充分地利用起来，

我们就需要把它拆分成若干个阶段。那在这个电路的结构上， 要进行拆分，我们就在每一个阶段之间添加上寄存器，

这就被称为流水线寄存器。

这些寄存器用于保存前一个阶段要向后一个阶段传送的所有的信息。

我们还是以取指到译码的这个阶段为例， 我们将指令存储器的输出接到一个寄存器上，

那当一个时钟上升沿来临的时候，指令存储器输出的指令编码就会被保存到这个寄存器当中，

那么在这个上升沿之后，指令存储器的地址输入如果发生改变， 随之影响的指令存储器的输出，也不会被存到这个寄存器当中去，

所以在这个时候，我们可以用新的PC来访问这个指令存储器。

从而得到下一条指令的二进制编码，而在这个同时，前一条指令的编码

已经在这个流水线寄存器的输出上，并且经过相应的电路，切分成不同的位域，

那其中有一个位域就会通过rs连到了寄存器堆，并且 选中对应的寄存器，把其中的内容放到busA这根信号上，

而这根信号也会被接到一个流水线的寄存器上。

那么当下一个时钟上升沿来临的时候，

当前这条指令所需要的rs寄存器的值，就会被保存到这个流水线寄存器当中，

与此同时，下一条指令的二进制编码也会保存到这个流水线寄存器中。那么在很短的 Clock-to-U

时间之后，译码阶段所看到的指令的编码就已经变成第二条指令了。

所以很快，寄存器堆得到的rs的寄存器编号也发生了改变，但是这没有关系，

第一条指令所需的寄存器的值已经保存到了这个流水线寄存器当中，

而且在这个时候，也应该会被送到了ALU的输入端， 所以这样通过添加流水线寄存器，我们就先从大体上

把这个单周期处理器改造成了一个流水线的处理器。

那我们对这个流水线的处理器进行一些简单的性能分析， 比如说我们要执行这么三条指令，

我们把时间轴画出来，从0时刻开始，每一格是200ps， 那如果是在单周期的处理器上，执行一条指令需要

这样五步，也就是取指、译码、执行、访存和写回。

假设每个阶段都正好需要200个皮秒， 那执行完这条指令，就总共过去了1000ps，

然后我们才可以执行第二条指令，又用去1000ps， 然后是执行第三条指令，那这样每条指令都需要花1000ps的时间，

那这个单周期处理器，它的时钟周期就需要被设置为1000ps，

从外界看来，这个处理器每1000ps可以完成一条指令。

而对于流水线处理器，我们同样来执行这三条指令。

先看第一条指令，那要完成这条指令所需要的步骤是一样的，

同样也需要这五步，也同样需要花1000ps的时间。

但是不同在于，在过去200ps之后，当第一条指令完成了取指阶段，而进入到译码

阶段的时候，实际上取指部件已经空闲下来，我们就可以开始第二条指令的取址工作，

也就是说第二条指令在此时，已经开始执行了。

同样，再过了200ps，第一条指令完成了译码，进入到了执行阶段，

这样第二条指令也正好完成了取指，可以进入译码阶段， 而此时，第三条指令的取指也可以开始了。

这样对一个流水线处理器，虽然一条指令总共也是需要花 1000ps，但是每200ps就可以开始执行一条指令，

而且当流水线填满之后，每200ps也就可以完成一条指令，

所以对于这样一个流水线处理器，它的时钟周期可以设为200ps，

因此，这个处理器的主频就是刚才这个单周期处理器的5倍。

当然这只是理想情况，现实中的性能提升幅度并没有这么大，

其中一个原因就是这些新插入的流水线寄存器，它自身也会带来一些新的延迟，

我们假设这些寄存器的延迟是50ps，那我们再来看一看这个处理器的性能有什么样的变化。

这是刚才没有考虑流水线寄存器延迟的情况下分析的

性能表现，那如果我们加上流水线寄存器的延迟，

> 同样还是执行这几条指令，那就需要每隔250ps 才可以开始一条新的指令，
>
> 所以时钟周期应该设为250ps，而且对于每条指令 本身来说，需要花1250ps才能够完成。
>
> 在这一点上，是比刚才在单周期处理器还要更慢一些的。
>
> 因此对于流水线处理器来说，因为各个处理部件可以并行 工作，从而可以使得整个程序的执行时间缩短，
>
> 但是流水线并不会缩短单条指令的执行时间，相反，还会增加这个时间。
>
> 因此，采用流水线的方式，实际上是提高了指令的吞吐率，
>
> 从而从整体上缩短了程序的执行时间，提高了系统的性能。

> 现在我们已经了解了流水线的基本原理， 而且分析了一个五级流水线的大致框架，
>
> 这也是早期流水线处理器的实现结构， 后来，流水线处理器的设计又发生了很多的发展和变化，
>
> 那我们在下一节将要进一步探讨这些问题。

## 602-流水线的优化

> 相对于单周期处理器， 流水线技术可以提升处理器的性能，
>
> 但是，如果仅仅按照指令执行的步骤去切分流水线的话， 不能够充分利用流水线这项技术的优势。
>
> 那如何才能挖掘流水线技术的更多潜力呢？ 我们在这一节就来探讨这个问题。

> 我们还是用这个厨房做菜的例子来对流水线进行分析。
>
> 现在我们将做菜的工作分为四步，每一步都需要一分钟。
>
> 如果不采用流水线的方式，做一道菜需要四分钟，而采用流水线的方式，
>
> 则需要4分钟多一些的时间，因为每一个阶段之间的交接还需要额外花一些时间，
>
> 对应于流水线处理器，就是各流水级之间的流水线寄存器所需要的延迟。
>
> 不过总体来说，这两种方式所用的时间还是大致相等的。
>
> 而如果做多道菜，流水线方式的优势就会逐渐体现出来，而且在流水线充满的情况下，
>
> 可以做到每一分钟上一道菜，而非流水线的方式，只能做到每4分钟上一道菜，
>
> 这样我们就会发现，如果我们划分为4级的流水线，我们可以将性能提升为原来的4倍。
>
> 不过刚才这个情况非常的理想化，其实我们很难做到每一个阶段恰好花同样的时间，
>
> 那我们不妨假设切菜这个环节非常复杂，需要花2分钟的时间，
>
> 而其他每个环节仍然只花一分钟，那我们现在应该让这个司号手隔多长时间吹一次号呢？
>
> 显然他应该每两分钟才吹一次号， 因为我们这个流水线的时钟周期必须按照各流水级当中时间最长的那一级来确定，
>
> 不然如果还按照1分钟吹一次号，那上一道菜还没有切完，下一道菜就已经洗完了，
>
> 并送到了切菜环节，那就把刚才还没有切完的菜给冲掉了，这样显然是不可以的。
>
> 所以我们只能2分钟吹一号， 像只有切菜这个环节是饱满的工作，其他环节都是干一分钟歇一分钟，
>
> 那我作为这个餐馆的老板，显然是无法接受这样的状况的。

这个问题就出在了流水线的平衡性上， 最终每个阶段花费时间不相等的流水线就被称为不平衡的流水线。

那么对于这条流水线，我们发现，虽然他还可以做到每两分钟出一道菜，

但是性能提升幅度就变小了很多，而且从单独做一道菜看来，非流水线方式

只需要5分钟，而用流水线方式，反而需要8分多钟，这样就慢了很多。

因此，如果是对于流水线处理器来说，

不平衡的流水线对于整体的指令吞吐率，和单独一条指令执行时间，都有非常不好的影响。

因此，在划分流水线的时候，应当做到每一级所花费的时间尽可能相等，

这就有可能造成这级流水线的名称和它实际所完成的工作并不完全相符。

比如说我们要求洗菜这个环节，就稍微多花一点时间把菜叶子先给撕碎了，

像切菜就可以少花一点时间。当然这只是一种方式，针对这个例子我们还可以考虑另一种方法，

那就是把切菜这个环节分为两步，

既然他需要花2分钟的时间，那我们不如把它分为2个一分钟的环节，

这样和其他环节的时间就变成一样的了，我们就可以设置其中周期为1分钟，

每过一分钟，各个环节就把手中的成果送到下一个环节去，

但我们要注意的是，这样的切分，并不是说要增加新的硬件资源，而应该

是把原有的硬件资源切分成两个部分来使用。我们就还是说切菜，

假设我们原来是要做土豆丝，那在切菜的时候，先用一把 小刀完成削皮的工作，再用一把大刀完成切丝的工作，

这样一共花了2分钟的时间。

后来我们经过分析发现，削皮大概正好用一分钟，切丝也需要用一分钟，那我们就把它切分- 成两步，

把削皮用的小刀放在第一步，切丝用的大刀放在第二步， 这样我们并不需要购买新的工具，而只是把原有的工具分隔开来。

那么这个调整后的流水线就变成了一个平衡的流水线，

对于这个流水线，我们发现，它单独做一道菜的时间已经降回到了5分钟，和非流水线的方式

基本相当，更重要的是，在连续工作的情况下，它又可以做到每一分钟上一道菜，

而非流水线的方式，只能是每5分钟上一道菜， 因此，采用流水线的方式，性能可以是原来的5倍，

如果我们切分4级流水线，那我们可以做到性能提升为原来的4倍， 而切分5级流水，就可以把性能提升到5倍，

那就着这个思路想下去，我们继续进行切分，是不是可以得到更高的性能呢?

简单来看，这样是对的，这个技术就被称为"超级流水线"，当然

这个技术并不没有它这个名字看起来那么神奇，实际上，我们是将五级流水线作为一个

基本的流水线划分，如果在五级流水线的基础上，将其中一些流水级细分为更多的阶段，

从而增加了流水线的深度，这样的流水线就会被称为超级流水线。

那么超级流水线就可以做到更高的时钟频率，从而提高了指令的吞吐率。

比如这是基础的五级流水线，其中 每个流水级的组合电路的延迟大约为200ps，

而流水线寄存器的延迟为50ps， 那这个流水线处理器的时钟周期就是250ps。

而如果我们做一个十级的流水线，而且恰好能将这个五级流水线当中的每一级平均地切为两段，

那这个处理器的时钟周期就是100ps，加上流水线寄存器的50ps，一共是150ps。

那显然，使用这样的超级流水线技术可以带来明显的性能提升，

但是流水线的级数是越多越好吗？我们还是要来深入地分析一下。

对于五级流水线来说，其执行单条指令的延迟是1250ps，

而对于这个十级流水线，它执行单条指令的延迟就变成了1500ps，

因此，我们切分流水线之后，提高了时钟的频率，

从而也提高了指令的吞吐率，但是单条指令的执行时间确实变长了的，这是因为我们增加- 了更多的

流水线寄存器，在五级流水线当中，流水线寄存器的延迟大约占20%的比例， 而在十级流水线当中，

因为每级的组合逻辑电路的延迟减半了，但是流水线寄存器的延迟是不会发生变化的，

因此，流水线级数划分的越多，流水线寄存器的延迟所占的比例就会越高， 从而导致单条指令的延迟越来越大。

而且不仅如此，当流水线级数变多之后，填满一个流水线所需要的指令就会变多，

而这些同时处在流水线当中的指令，他们之间的关系 也就会变得更加复杂，从而会带来更多的负面影响，

这些负面的影响我们在后面还会进行深入的分析， 但我们的结论是很明确的，流水线的级数显然不是越多越好。

那我们再来看看现实中的情况。

比如说93年的奔腾处理器，它就采用了五级的流水线， 而MIPS则是在更早就实现了流水线处理器，

包括有五级的，和后来的八级的。

因为MIPS在设计之初就充分考虑了用流水线进行实现，

所以它很容易通过流水线以及超级流水线的技术获得性能上的提升，

而X86指令系统在设计的时候，并没有考虑这一点，而且由于 它自身比MIPS复杂得多，也很难进行流水线的切分，

因此在80年代到90年代初，

以MIPS为代表的RISK处理器性能提升很快，对X86处理器构成了巨大的威胁，

不过后来英特尔在非常困难的情况下，找到了一个应对的方法， 这个方法从95年的奔腾Pro开始实现，

奔腾Pro是一个12级流水线的处理器，也有一些划分方法认为它是14级的。

它的核心要点在于，在处理器内部，用硬件将 复杂的X86指令切分成简单的RISK指令，

从而可以运用这些先进的RISK处理器的技术，又保证了兼容之前用X86指令编写的软件，

那么经过几番争斗之后，作为SISK代表的X86还是战胜了来势凶猛的RISK处理- 器厂商，

虽然它用的也正是RISK的技术。而现在 作为RISK代表的ARM，在那时还并没有多大的影响力，

至少在计算机的领域是这样的。这是97年的ARM9，采用了五级流水线的形式，

这个流水线的划分和MIPS基本上是一样的，后来到02年，ARM推出了8级流水线的A- RM11，

这处理器凭借它低功耗的优势，在嵌入式领域获得了广泛的应用，

但是由于其性能的不足，还是难以进入个人计算机的市场。

那么在这个阶段，英特尔在战胜了众多RISK厂商之后， 又和MD等厂商开始了X86体系结构内部的斗争。

那么在这个时期，处理器的流水线级数变得越来越多，

从10多级变到20多级，直到04年的奔腾4达到了顶峰，共有31级。

我们刚才已经分析过，流水线的级数并不是越多越好，过深的流水线反而会降低性能。

那英特尔显然不会不知道这一点，但他为什么还要这么做呢?

其中一个重要的原因就是流水线加深之后，可以带来的一个很明显的变化—时钟频率的提升。

虽然我们现在知道了频率高不代表性能好，但其实很多普通的消费者并不清楚这一点，

而且在那几年，英特尔和MD等厂商，在广告中都是以处理器的频率

作为主打的性能指标，这就造成了那时消费者购买计算机时，

都只问"你这个计算机频率是多少？"，结果这就变成了这家出一个2G的， 过几天那一家就出个2.5G的，那你出个2.5G的，

再过几天，我就出个3G的，谁的频率高，来买的人就多， 这就是那几年非常著名的CPU频率大战，

这也带动了那时主流处理器的流水线深度变得越来越深。

当然这样的事情是不能一直做下去的，它总归有一个限度，所以到后来频率大战打不下去之后，

流水线的深度也就逐渐地回落了，比如06年 的Core

2是14级，08年的Core i7是16级， 而到现在，13年的Core

i7在正常工作情况下，也是14级， 那这是桌面处理器的主要情况。

而随着乔布斯推出的iphone和ipad引爆了另外一个市场，

我们也来看一看它的流水线深度，比如09年的Cortx-A8，

这是ARM11后一代的处理器，它有13级流水线呢，

而10年的Cortex-A9，则是8-11级的流水线，我们可以看到它的流水线级数不- 升反降，

而11年的Cortex-A15，则是15级流水线， 到13年的Cortex-A57，也仍然保持了这个数目，

而A9，A15和A57，这些处理器，或者是与它们同架构的

一些别的处理器，广泛应用于中高端的智能手机和平板电脑。

所以从这里也可以看出，现在的主流处理器，其流水线深度基本维持在15级左右。

现在我们已经了解到，通过 增加流水线的深度，可以有效地提高时钟的频率，

从而提升指令的吞吐率，不过这个方法也有很大的限制，

> 现在我们已经很难通过继续增加流水线的深度来获得性能上的提升了。
>
> 那如果我们想继续挖掘流水线处理器的潜力，
>
> 就还得寻找其他的优化方案。

## 603-超标量流水线

> 超标量流水线在现代的处理器当中得到了广泛的应用
>
> 也就是这种技术给我们带来了出色的处理器性能。
>
> 那究竟什么才是超标量流水线，它又是如何工作的呢？ 我们这一节将一起探讨这个问题。

> 这个是我们之前已经构建好的一条流水线，但是如果我们觉得它的吞吐率还不够高， 其中一个改进的方向就是加深流水线，
>
> 而与之相对另一个改进的方向就是拓宽流水线，而拓宽流水线的这个方案我们就称为超标量。
>
> 在现在通常来说超标量的结构是只有两条或者有两条以上可以并行工作的流水线，
>
> 那好就这个厨房的例子，现在为了满足更多客人的需求， 我们就需要把这条做菜的流水线进行拓宽，
>
> 我们从左往右来看首先我把这个洗菜的池子扩大一倍，这样就可以同时洗两道菜的原料了。
>
> 那与之相匹配的后面的各个环节都得加倍，
>
> 这样我们就有了两条可以并行工作的流水线，这就是一条超标量流水线。
>
> 在处理器设计当中我们也经常简称为超标量。
>
> 那么与超标量流水线相对，以前的单条流水线就可以被称为标量流水线，
>
> 那采用了超标量结构的处理器，我们也常称之为超标量处理器。
>
> 那么就结合这个例子简单的来看一看超标量流水线是如何工作的。
>
> 既然我们这个洗菜的池子扩大了一倍，那么同时就可以进行两道菜的操作，那么当
>
> 一分钟过去之后，洗菜这个环节就会将这两份菜的原料分别送到切菜1这个环节的两位操作人- 员手中，
>
> 而与此同时第三道菜和第四道菜则会进入洗菜环节。
>
> 这样在每一个环节都有两道菜菜在同时的并行向前， 那到了五分钟的时候就可以同时完成两道菜。
>
> 而对于处理器，这也常被称为双发射的结构， 那如果照着这个方式再扩展一条流水线就变成三发射。
>
> 再扩大一条就会变成四发射，这就是超标量流水线的基本工作原理。
>
> 那我们再来看一些实际的例子，奔腾是第一款采用超标量技术的x86CPU。
>
> 这就是奔腾的流水线结构，它是一个双发射的五级流水线， 这两条流水线分别被命名为U流水和V流水，
>
> 它们共用取指和译码的部件但是有自己独立的地址生成逻辑，ALU
>
> 以及数据高速缓存的接口，那在一个时钟周期内可以同时发送两条指令，分别到U流水和- V流水。
>
> 当然超标量技术并不是到这个时候才诞生的， 而是在很多年前就在计算机当中获得了应用。

这是历史上第一台超级计算机CDC6600，

那么在这台计算机当中，有十个并行的功能部件，因此也被认为是最早采用了超标量技术的- 计算机。

不过这些功能部件内部并没有采用流水线，所以从这里我们可以看出 超标量技术和流水线技术实际上是相互独立的，

只不过在现在采用了超标量技术的CPU也都是使用流水线的技术。

那虽然CDC6600没有采用流水线，但它的下一代产品 CDC7600就在这些并行的功能部件内实现了流水线的技术。

那我们再回到现在，ARM Cortex-A9的流水线结构。

它每个时钟周期可以发射四条指令， 根据指令的不同，总共会经过八到十一级流水线。

与奔腾类似的是在流水线的前端比如说 取指，译码并没有分成多条流水线而是采用统一的部件。

当然我们要知道这些部件虽然看上去是一个，但它实际上比标量流水线要大得多，

比如说取指部件至少一次要能取来四条指令甚至更多，

而译码部件一次也至少应该完成四条指令的译码，而到了流水线的后端，

才会从结构表示上体现出多条并行流水线的形态。

然后我们再来看Core i7的超标量流水线。

这个流水线就更为复杂，这个是指令高度缓存 也就相当于我们在流水线原理当中提到的指令存储器，

我们可以看到每个周期从指令存储器当中会取回128个比特，也就是十六个字节，

因为x86指令长度是不固定的，所以首先要经过一个 指令长度的译码器，分解出到底哪几个字节是一条指令。

那么在这一点上类似的指令系统就体现出了明显的优势， 它每条指令都是定长的，不用额外进行这样的识别工作。

那么还有一点值得一提的是在译码器当中通过硬件会将x86的指令转换成更为简单的指令，

这些指定被称为V操作，那从这里可以看出有三个简单的译码器，

用于对那些比较简单的x86指令进行转换，每条指令对应一个V操作。

而那些非常复杂的指令则会通过这个复杂的译码器转换成多条V操作，

而这些V操作都是类似于risk指令的格式， 这样在它流水器的后半部分看到的都是risk格式的简单指令了。

那这张图还只是流水线的前半部分，

我们再画出后半部分。在这里我们也可以看出多条并行的流水线，

而且因为在这个流水线当中运行的是V操作，都是采用了risk的编码风格，

所以这里也可以充分运用大量面向risk处理器研发出的高级流水线的技术。

这也就是为什么我们现在经常说x86虽然是一个CISCO的指令系统， 但它实际上是用risk的方式去实现的。

那这个CPU就是一个四日发射十六级流水的超标量流水线，

那我们再来比较一下超标量流水线和之前的标量流水线，

> 最开始从单周期处理器到流水线处理器主要考虑的是时间并行性上的优化，
>
> 通过对现有硬件进行切分，只是增加了少量的流水线寄存器以及部分的
>
> 控制信号的改动，那原本串行执行的指令在一定程度上并行起来，
>
> 而从标量流水线到超标量流水线则是主要考虑了空间并行性上的优化，
>
> 这是让不同的指令同时在不同的流水线上运行，
>
> 那么简单的看来每增加一个发射数就需要增加一条流水线的硬件资源。
>
> 我们最后来看一看超标量流水线与多核的关系。
>
> 从原理上讲它们都是在空间并行性方面寻求的优化， 我们还是看一个实际的例子。
>
> 这就是Core i7，也是我们现在熟知的多核超标量处理器。那么来看看它内部的结构。
>
> 首先我们来看一个概念叫做处理器核。这部分实际上就包含了 我们之前介绍的那些数据通路控制信号等等。
>
> 当然还需要包含指令和数据的高度缓存对应了 我们原理结构当中的指令存储器和数据存储器。
>
> 那为了提高性能现在的处理器当中一般还配备了二级的高速缓存。
>
> 这些部件的关系非常紧密，我们通常也就把这一部分称为一个处理器核。
>
> 那么刚才看到那个四发射十六级流水线的结构图就是在只这么一个处理器核
>
> 内部的结构，那么可以说这一个处理器核就是一个超标量流水线的处理器核，
>
> 而在单核的时代这个部分结构就单独制造出了一个芯片，就是以前的单核CPU，
>
> 那现在我们把这样同样的结构复制多份，然后再加上一些共享的存储部件
>
> 就构成了一个多核的CPU，这里面有四个核儿那就是一个四核的CPU， 每一个核内部都是一个超标量流水线的结构，
>
> 这就是我们现在通常说的多核CPU和超标量流水线之间的关系。

> 现在我们已经从整体框架 层面了解了流水线处理器的发展演变的过程，
>
> 而且也知道了现代最先进流水线处理器的大体结构。
>
> 但是仅仅了解这些是不够的，
>
> 流水线的方式引入处理器之后它自身也带来了一些新的问题， 我们必须了解并解决这些问题，
>
> 这是我们下一节将要探讨的主题。

## 604-流水线的冒险

> 流水线技术之所以能提高性能 究其本质是利用了时间上的并行性，
>
> 那它让原本应该先后执行的指令
>
> 在时间上一定程度的并行起来，然而这也会带来一些冲突和矛盾，
>
> 进而可能引发错误，这就是我们这一节所需要探讨的问题。
>
> 首先我们来看什么是冒险。
>
> 在流水线当中我们希望每一个时钟周期都有一条指令进入流水线开始执行，
>
> 但是在某些情况下下一条指令无法按照预期开始执行， 那这种情况就被称为冒险，冒险分为三种，

一是结构冒险。在这里结构是指硬件电路当中的某个部件，

如果这条指令所需要的硬件部件还在为之前的指令工作无法为这条指令提供服务，

那就产生了结构冒险。第二种是数据冒险，

如果这条指令需要某个数据而之前的指令正在操作这个数据，那这条指令就无法执行，

这种情况称为数据冒险。第三种是控制冒险，如果现在要执行哪条指令，

是由之前指令的运行结果来决定的，而现在之前指令的结果还没有产生，那就导致了控制冒险。

那下面我们会逐个分析这些冒险。首先来看结构冒险， 我们还是通过一个指令执行的例子来讲，

这是一条指令在流水线中执行的步骤，那我们是用每一步动作的缩写来表示的，

那现在我们换一种描述的方式用这每一步所用到的硬件部件来表示， 那么注意到取指阶段需要用到指令存储器，

译码阶段需要用到寄存器堆，执行阶段需要用ALU，访存阶段需要用数据存储器，

而写回阶段也需要用寄存器堆。

用这样的方式我们比较容易看出在哪些时刻会有可能出现硬件部件的争抢情况。

那么假设执行的这么一段指令的序列，第一条是一个Load的指令，后面是若干其他的指令，

那么期望这些指令依次进入流水线开始执行， 那么注意在是第四个时钟周期Load的指令要从存储器中读取数据，

而与此同时取直部件也要从存储器当中读取第三条指令的编码，

那如果我们这个系统当中指令和数据是存放在同一个存储器当中的，

而对于一个存储器在同一个时刻只能接受一个读操作，那这里

就会发生结构冒险，那如果我们只能使用同一个存储器怎么来解决这个问题呢?

其实方法很简单，既然不能同时读，那就不读好了，

那在这个时钟周期首先让Load的指令去读存储器你获得它所需要的数据，

而取指部件这时不读存储卡而是让流水线停顿。

在所谓停顿，也不是什么都不管，必须要将相关的控制信号视为不改变及其状态的值， 那这种设置我们就称为一个空泡。

这个表示也是形象化的描述了流水线停顿这一事情。

这样这个结构冒险就被消除了，而第三条指令的取指被延后到下一个周期才开始。

当然我们也会说这样会不会和第一条指令的访存产生冲突呢? 那当然如果第一条指令也是访存指令，

那还是会发生结构冒险。那流水线还需要再停顿一个周期，

第三条指定要等到下一个周期再进行取指。如果连续出现几条访存指令， 那后面流水线就会连续的停顿，

这样效率很低，但是从另一个角度讲这是一种非常安全 又简便的方法，用这种方法其实可以解决各种冒险，

当然既然它效率比较低，我们还是要尽量避免让流水线停顿。

所以在现在的处理器当中我们通常还是将指令和数据分别放在不同的存储器当中，

就是靠在存储器当中设置独立的指令高度缓存和数据高度缓存来实现的。

我们还是要强调的在计算机中主存储器也就是内存是统一存放指令和数据的，

这也是冯诺依曼结构的要求，只是在CPU当中 的一级高速缓存会采用指令和数据分别存放的方式，

那这种结构冒险我们现在就已经解决了。

我们再来看下一种情况，还是这段代码，

如果再过了一个时钟周期，这时候iii的指令就要将从存储器中读出的数写入寄存器堆了。

而与此同时我们注意到第三条指令也在读寄存器堆，

那这里就出现了两条指令同时要对一个硬件部件进行操作的情况。

要解决这个冒险我们就得让寄存器堆同时支持读寄存器和写寄存器，

那我们是如何让寄存器堆提供这样的功能的呢？其实这和寄存器堆本身的特性有关，

相对来说寄存器堆的读写速度比较快， 我们假设读或者写寄存器的延迟为100ps，而其他部件比如说ALU的延迟就就比较大，-

视为200ps， 那么我们就可以在前半个时钟周期用于完成寄存器堆的写，

后半个时钟周期用来完成读操作，并且在寄存器堆上设置独立的读写口。

这样就可以在一个时钟周期内同时完成了读和写的操作。

那这两种机构冒险的情况在我们设计这个处理器的初期就已经避免了，

我们在选择处理器的组建时就考虑到了不同指令对各个部件的需求， 从而使得不同的指令不至于争抢同一个硬件部件。

虽然从这个例子看来我们用这个处理器并没有结构冒险的情况，

但是如果要设计一个新的处理器，结构冒险仍然是我们优先要考虑并解决的问题。

那好我们再来看一个数据冒险的例子。比如说这段指令的代码第一条减法指令，

它的运算结果会放倒t0寄存器当中，而下一条指令需要将t0寄存器作为加法运算的一个原- 操作数，

从这段代码的功能看来加法指令所用的t0寄存器的内容

显然应该是减法指令的运算结果。但是在流水线处理器上，

加法指令开始执行时，这条减法指令的运算结果可能还没有写到t0寄存器当中去， 我们结合图示来进行说明。

减法指令需要到第五个周期也就写回这个周期才会将运算结果写到t0寄存器当中去，

而加法指令在第三个周期也就它自己的译码这个阶段就需要读出t0寄存器，

那从这里就可以看出，这条加法指令需要用前一条指定的运算结果， 但是在这个时刻这个运算结果还没有写回到寄存器当中去，

这就产生了数据冒险，

如果不做任何处理任由加法指令去读取寄存器堆， 那此时得到的t0寄存器的值肯定不是由前面这条减法指令运算得出的，

这样就会导致这个程序运行结果的错误。那么如何来解决这个问题呢？

还记得我刚才提过那个万能的方法吗？只要遇到冒险我们用上它就能解决。

> 对了，这个方法就是让流水线停顿。
>
> 既然你结果还没有产生，那我就等，等到你结果产生，
>
> 根据这个流水线的结构我们需要让流水线停顿两个周期，这样在
>
> 加法指令读寄存器堆的时候，减法指令已经将运算的结果写回到了t0寄存器当中去，
>
> 所以加法指令读到的是正确的数值。
>
> 那么再来看一个控制冒险的例子，这段代码第一条指令
>
> 是条件分支指令，后续跟了若干的指令，那我们也结合图示来说明，
>
> 我们要注意的是在第二个时钟周期，处理器就应该去取下一条指令了， 但这个时候实际上并不知道是否真的会发生分支，
>
> 这条分支指令一直要到执行阶段结束，才能知道分支的条件是否成立，
>
> 也就在600ps这个时候，而处理器希望在200ps的时候就去取下一条指令，
>
> 这里就产生了控制冒险，因为这个区指定的 动作如何进行应该由上一条地指令的运行结果来决定，
>
> 而上一条指令的运行结果至少要到两个时钟周期之后才能产生，
>
> 那在还没有确定是否发生分支的情况下如何进行下一次的取值呢？ 那如果单纯只想解决这个冒险，而不考虑性能的损失的话，
>
> 我们还可以用那个万能的方法， 就是让流水线停顿。
>
> 我们需要插入两个空泡，那么在执行阶段结束之后，我们就知道要从哪个地方开始取新的- 指令了。
>
> 这样就可以解决这个控制冒险。

> 现在我们已经知道了这些冒险所带来的影响，
>
> 如果不好好解决这些问题，流水线处理器就没有使用的价值，
>
> 所以我们之后还需要对其中的一些重点 做深入的分析。

## 605-数据冒险的处理

> 在程序当中， 我们经常会对同一个变量进行反复的使用和修改。
>
> 那这样对于流水线处理器来说，就会经常出现数据冒险的情况。
>
> 我们必须很好的应对和解决。
>
> 在这一节，我们就来看一看有哪一些不同的解决方法。
>
> 我们先来看这个数据冒险的例子，
>
> 产生这个数据冒险，是因为第二条加法指令会用到第一条减法指令的运算结果。
>
> 但是在流水线当中，这条加法指令在读取t0寄存器的时候， 它前一条减法指令还没有把运算结果写到t0寄存器当中去，
>
> 所以这里就存在一个数据冒险。要解决这个数据冒险，最简单的方法，实际上是在软件层面进- 行解决。
>
> 假设我们这个处理器的流水线并不能解决这样的数据冒险， 那其实，我们只要通过编程的手段，人为的将这条加法指令退后执行，

让他读取寄存器堆的时间，退后到减法指令寄存器堆之后。

那这应该怎么做呢，我们有一条指令叫做nop，它的

作用是什么也不干，我们就在这个减法指令和加法指令之间插入两个nop指令。

这两个nop指令只是简单的通过流水线，并占用了相对的时间。

那这样刚才的这个数据冒险至少是不存在了。

而因为这两个nop指令的作用，加法指令退后了两个周期才进入流水线，

那么当这条加法指令需要读寄存器堆堆时候，前面堆减法指令已经完成了对寄存器堆堆写。

那加法指令就可以从寄存器堆当中读到正确堆t0的值，从而完成正确的加法运算。

所以解决这个数据冒险最简单的方法就是插入nop指令，但是这个方法也有很大的问题。

首先，到底应该几个nop指令，这是和流水线的结构相关的。

如果我们这一段程序放在这个5g流水线上是正常运行的，那过几天，

又出了一个更新的处理器，它的流水线是8g的，那这个程序放上去，可能运行就会发生错误。

因为流水线变身之后，解决数据冒险需要的周期数可能会变多。

所以插入nop的这个方法可行，但是并不好。

在一般情况下，我们还希望对软件屏蔽硬件的这些时限细节。

那既然加了两个nop指令能够解决问题，那么就可以尝试在硬件上完成相同的工作。

那刚才通过插nop的方法，其实已经给我们提供了借鉴，

我们只要发现存在这样的数据冒险，我们就在硬件的流水线上让各个控制信号都变成执行no- p指令一样的值。

那在这两个周期，就会产生流水线停顿的效果。而这些和nop指令效果一样的控制信号，

它们所产生的状态，就成为一个空泡。那这个空泡随着时钟周期一级一级往后面穿，

从效果上来看，和nop指令在流水线当中一级一级的执行是一样的。

只是区别在于，这样的信号是由硬件来产生的。

那现在又有了一个新的问题，如果刚才是在软件中插入了nop指令，

那对于这个流水线来说，它是严格的按照取回一条指令进行执行，这样的方式来运转的。

那现在需要在硬件上自动的插入空泡，那就需要一个方式来检测是否出现了数据冒险。

当然这也不难， 如果我们不是看这一段程序代码，而是看处理器当中的

这五个部件，那我们怎么来判断存在数据冒险呢。

所谓数据冒险，就是当前有一条指令要读寄存器， 而它之前的指令要写寄存器，但又没有完成，

所以我们只用检查，在译码这个阶段，需要读的寄存器的编号， 这个通过链接在寄存器读口的信号就可以得到。

然后我们再检查后面各个阶段，

其实在每一级，都有些信号能够表明这条指令是否要写某个寄存器，以及要写哪个寄存器。

因此，我们只需要检查后面每一个阶段所要写的寄存器的编号，和当前译码阶段，

所要读读寄存器的编号，是否有相同。如果存在相同，那就是有数据冒险。

那只要出现来数据冒险，我们就在流水线中插入空泡。

这样我们就能通过硬件来解决数据冒险的问题。

但是，在实际的编程当中，这种先写了一个寄存器，然后很快使用的状况是经常出现的。

如果说每次出现，我们都要让流水线停顿的话，对性能的影响就太大了。

所以我们不能只追求做对，还要要求做好。我们还是希望流水线不要停顿。

那这个就是最初我们分析的样子，减法指令在800频秒之后才开始写寄存器，

而加法指令最晚在500频秒的时候就要去读寄存器。

我们无法逆转这个时间，所以我们肯定不能把800频秒才有的数送到500频秒的这个- 时间去。

但是我们可以换一个角度想一想。这条减法指令的运行结果真的是在这个时候才有的吗?

实上减法运算是在执行阶段由ALU这个部件完成的，

所以最晚在600频秒的这个时候，要写到t0寄存器当中到这个数已经运算完成了。

所以从时间角度来看，在600频秒之后，我们都可以得到t0寄存器的最新的值，

而对于这条加法指令，它真的需要使用t0寄存器的值是在它的执行阶段，

也就是ALU的部件需要用t0的值作为其中的一个输入， 那这个阶段是在600频秒之后才开始的，

我们完全可以将减法运算的结果交给这个加法运算作为输入。

那这种方法，就叫做数据前递。也就是 上一条指令将自己的运算结果往前传递到下一条指令去，

那我们刚才已经分析过，在600频秒的时候，ALU的输出结果已经是t0的值了，

那在600频秒的这个时钟上前过去之后，t0的这个值会被保存到执行和访存之间的这个流- 水线寄存器当中去。

我们如果把它传递给ALU的输入，就可以正确的完成后面这条加法运算了。

那既然从时间上是可行的，我们就可以来看一看硬件上怎么来修改。

这条减法指令在执行完运算以后，运算结果已经保存到了这个寄存器当中。

那现在，这条减法指令进入到访存阶段， t0的值将会通过这个阶段传到下一级流水线寄存器。

而与此同时，加法指令正在执行阶段，它需要将t0寄存器的值送到ALU的一个输入端。

那显然，它的上一个阶段从寄存器堆当中读到的值，肯定不是最新的。

现在这个最新的值在访存阶段的连线上。

所以我们从硬件连线上可以把这个信号引回来，从新引导ALU的输入端。

当然，这里我们还需要增加一个多选器， 而且我们刚才也讲过，如何去判断在流水线当中出现了数据冒险。

那我们就可以用这样的判断结果作为这个多选器的选择信号， 在出现数据冒险的时候，我们选择这个前递的信号，

那当然，这条加法指令也有可能在第二个原操作数上使用了t0寄存器。

所以这个前递的信号还应该传送到ALU的另一个输入端， 当然在这里也需要加上多选器来进行选择，

那这样的方式就被成为前递。它还有个名称叫作旁路。

那从根本上来说，前递和旁路指的都是这件事情。只不过是观察和描述的角度不同而已。

前递是从指令执行顺序的角度来描述的，而旁路则是从电路的结构角度来描述。

本来前一条指令应该将运行的结果写入到寄存器堆，然后再交给后一条指令使用，

而我们现在搭建来一条新堆通路，相当于绕过了寄存器堆，直接进行了数据堆传递，

所以从硬件时限的角度来看，这是一个旁路。那这就是前递和旁路的关系。

那我们进一步来看，其实不仅仅在这个点可以建立旁路，我们在下一个流水级也可以建立旁路。

那这条旁路在什么情况下会用上呢？我们还是结合一个例子来看。

这个例子前两条指令和刚才的那个例子是一样的， 在此基础上我们又写出了第三条指令，

这是一个与操作，那么它其中的一个原操作数也是t0，那我们结合实践来看，

对于这条与操作指令，它真的要开始运算的时候，是在800频秒之后。那在这个时候，前面- 这条减法指令

已经完成了访存阶段，所以t0寄存器的最新值 现在是放在访存阶段和写回阶段之间的流水线寄存器当中的，

那我们就需要用到刚才的结构图当中紫色的旁路的线，

用来将t0的内容传递到ALU到输入端，从而让这条与运算指令及时端运行。

那如果再往后一条指令又用到了t0，都会怎么样呢？

那么这个标着3的指令在800频秒之后的这个时钟周期正好进入了译码阶段，

它会在这个周期的后半部分读取寄存器，那么在这个时候， 减法指令已经将t0的值写入到了寄存器堆中，

所以对于这个3号指令，如果它用到了t0这个寄存器，

它就可以按照正常的操作，从寄存器堆当中读出t0寄存器读值，而不需要使用前递的技术。

所以对于这样运算指令，我们建立的这两组旁路的通路，就已经可以解决数据冒险了。

但是还是有一种例外的情况，

我们通过一个新的例子来看，在这个例子当中， 前三条指令还是和刚才一样，第四条是一个low的指令，

它也会用到t0寄存器，但是我们刚才已经分析过了，这个时候并不存在数据冒险。

而这条low的指令是要把存储器当中的一个数取出来，存放到t1寄存器当中去。

而它之后，一条或运算指令会使用t0寄存器的值，

那这种情况就是一条low的指令之后跟了一条指令，

会使用low的指令的目的寄存器。那在这种情况下，也会发生数据冒险。

它有个专门的名称，叫作load - use冒险。那么这种冒险是否也可以用前递的技术来- 解决呢？

实际上是做不到的，那我们来分析一下为什么做不到。

对于这一条low的指令，我们来看要保存到t1寄存器的值，究竟是什么时候才得到的，

对于刚才的运算指令，需要写回寄存器的值，是在执行阶段，也就是同过ALU运算而得。

但是对于low的指令，用ALU是计算要访存的地址，

而要写回寄存器堆堆数，是在访存阶段的结束才会得到， 所以是在1400频秒这个地方，我们才会得到t1寄存器的值。

而对于下面这一条或运算指令，我们最晚也得在1200频秒这个地方，

得到t1这个寄存器的值，从而让ALU可以进行正确的运算。

因此，这就要求我们将1400频秒 这个地方得到的数，传递到之前1200频秒这个时刻。

那时光倒流的事情我们是做不到的。

所以我们只能让信号沿着时间轴向前传递，而绝不可能向后传递，

因此，无论我们怎么修改电路，也无法构造出 这样一条前递的通路。那我们应该怎么来解决这个load - use的这个冒险呢?

其实说难很难，说简单也就很简单。还是用我们那个万能的方法，

既然我们不能返回到更早的时间，那我们只能让这条或运算指令多等一个周期，

这样它就可以在1400频秒之后才需要这个t1寄存器的值。

而此时，low的已经完成了从数据存储器当中取出数的操作，

这就可以通过刚才我们已经建立的第二组旁路通道，也就是用

紫色的连线表达的这个旁路通道，将t1寄存器的内容传送到ALU的输入端口。

那当然，既然我们要让或运算指令延后一个周期，

我们就必须在流水线中插入空泡，让流水线产生一次停顿，

所以对于这种冒险，我们需要用流水线停顿再加上数据前递的方式来解决。

那这个解决方案没有让流水线获得最高的指令吞吐率，

这当然是一个遗憾，但是保证指令执行正确才是我们的首要目标。

所以我们也只能接受这样的方案了。

现在，对于一个基本的流水线结构，我们已经能够处理数据冒险了。

但是，如果继续增加流水线的深度， 或者扩展成超标量流水线，又会出现新的数据冒险的情况。

当然，与之对应的又有很多精巧的解决方案。

如果你对此感兴趣，还可以进一步的深入学习。

## 606-控制冒险的处理

转移指令由于其自身的特殊性，总是会给我们带来一些麻烦，

那对于流水线处理器来说，更是如此，

转移指令会带来更多不良的影响。那我们应该如何应对和解决呢？ 这一节我们就来探索这个问题。

我们先来看一看转移指令对

流水线的影响。这是一条时间轴，每一小格都代表着一个时钟周期，

那对于五级流水线来说，5个时钟周期执行完一条指令，但是每一个时钟周期都可以读入一- 条指令，

并且从第五个时钟周期之后，每个时钟周期也都可以完成一条指令。

如果流水线始终处于这样充满的状态，那就达到了我们 流水线的性能目标，也就是获得最大的指令吞吐率。

那我们来看一看这样一段程序代码，我们假设在T1的这个时钟周期去取指的就是这条ad- d指令，

那么到了第二周期，取指部件取回来add指令，并交给译码部件进行译码，

与此同时，取指部件开始取下一条指令，也就是这条sub指令。然后到T3周期， 接着取下一条指令，就是这条beq指令。

那通过这段程序我们可以看出，这里很可能会有一段循环， 但这只是从我们旁观者的视角，我们能看到所有的程序代码，

而对于处理器来说，现在正处于T3这个周期时，它正在去取下一条指令，它根本不知道这条- 指令是什么。

当T3这个周期的取指工作完成之后，虽然这条beq指令的指令编码被取回，

也是在T4周期这条beq指令被送到译码部件， 而取指部件则会依次去取下一条指令，

那就是这条load指令。

当取回这条load指令的时候，beq指令译码也已经完成，但我们仍然不知道它的转移条- 件是否满足，

也就是s3、s4这两个寄存器是否相等，所以这时候我们只能继续取指，

那再往下取回的就是这条store指令，就是当T5这个时钟周期完成的时候，

beq这条指令也完成了执行的工作，也就是比较完成S3和S4这两个寄存器的值，

这时我们才能知道是否要发生这次转移。那我们假设转移的条件是满足的，

那这样已经进入流水线的这条load和store指令，实际上是不应该被执行的， 那我们只能把它清除，然后重新从正确的地址开始取指，

也就是取回这条减法指令，然后再依次取到这条条件转移指令。但是现在我们所购到处理器

并不能记住刚才曾从某个地方取回了这条条件转移指令，所以在这时，处理器只是简单地去- 取指令，

因此它会仍然继续往下取指，再会取到这条load指令，然后再取到一条store指令，

只有当这条store指令被取回的时候，刚才取到的这条beq指令才会执行完成，

那处理器可能又发现原来是要发生转移的，那必须把load和store指令清除掉，然后- 重新取指，

那么就发现在这个循环的执行过程中，总是反复地执行了 两条正确的指令，然后取回了两条不应该被执行的指令，

从这个图上来看，就有两个红色的椭圆和两个黑色的椭圆交替出现。

那在这段循环执行的过程中，实际上有50%的性能就被浪费了， 这也是因为转移指令本身和流水线的模式是冲突的，

因为转移指令会改变指令的流向， 而流水线则希望能够依次地取回指令，将流水线填满。

那如果这种情况是非常罕见的，也许我们还可以容忍，但实际上转移指令是非常常用的指令。

通过对大量程序的分析可以看出，大约每隔4到7条指令就会

有一条转移指令，转移指令所占的比例大约为15%-25%，

而且转移指令往往会导致若干条不应该被执行的指令进入流水线， 而清除这些指令则会带来时钟周期的损失，

那我们把转移指令所占的比例乘上转移指令带来的时钟 周期的损失，就可以大致地测算出转移指令对性能的影响。

那对于比较简单的流水线来说，转移指令带来的损失可能还不大，

但是我们知道现代的处理器都是超标量深度流水的处理器， 例如像Core

i7是4发射16级流水， 我们可以简单地认为，流水线在充满的时候，可能会有4乘以16，总共64条指令在

流水线中。而再看智能手机当中经常使用的ARM Contex-A15处理器，

这是一个3发射15级流水线的处理器，那我们也可以简单地认为 在流水线中，总共有3乘以15，45条指令在同时执行。

那一旦出现转移指令，就有可能导致其后的几十条指令都是不应该被执行的，

所以说，流水线越深，超标量数越多，转移指令带来的影响就越大，

如果不解决这个问题，那我们花费大量的精力设计的这些深度流水线和超标量结构都将失- 去意义。

那我们就来深入地分析一下转移指令的影响。

在执行转移指令的时候，如果确实发生转移，那就需要将其后按顺序

预取进入流水线的这些指令废除，也被称为“排空流水线”，

然后从转移目标地址重新获取指令。那细分来看，我们主要要做两项工作，

一是要判断要不要转移，也就是转移的条件是否成立，

如果执行了一条转移指令，但实际不需要发生转移，那刚才按顺序进入流水线的指令就不需要- 被废除。

第二个问题是转移到哪里， 也就是我们为生成目标地址所需要做的工作，那想要消除

转移指令带来的影响，我们就要对每一条转移指令都解决这两个问题。

现在我们把转移指令进行分类的列举， 我们按两种分类的方法，可以把转移指令分为4类。

首先来看无条件转移当中的直接转移， 这里我们列举了X86和MIPS当中的无条件直接转移指令，

其中我们来看一个例子，这条 j

指令，它是一条无条件转移指令，也就是 执行到这条指令时，一定会发生转移，而且它还是叫直接转移指令，

这就是说，它转移的目标地址是在指令编码中直接给出的。

与之相对的是间接转移，这里也有一些例子，我们来看其中的一条。

这条jr指令，它也是无条件转移指令，执行到它的时候，一定会发生转移，

但它转移的目标地址并没有在指令编码中直接给出，而是放在了一个寄存器当中，

所以需要先去读取这个寄存器的内容，才能得到转移的目标地址，

这就是间接转移。然后我们再来看条件转移， 像这条beq指令，它需要比较t0和t1这两个寄存器的

内容是否相等，如果相等则转移，不相等就不转移， 而且它也是直接转移类的指令，

它的转移目标地址是直接在指令编码当中给出的。

那我们就以这三条指令为例，来看一看如何消除转移指令带来的影响。

首先来看无条件直接转移， 在MIPS当中，这是一条j型指令，

对于这条指令，我们不用判断要不要转移，我们只需要考虑转移到哪里这个问题。

那这条指令目标地址的计算方法是这样的， 首先这条指令的编码当中，带有一个26位的立即数，

这个数就是要转移的目标地址的主体部分， 但是我们的目标地址应该是32位的，

所以还差6位，在差的6位当中，低两位我们用0补上，因为目标地址肯定是四字节对齐的，

地址的低两位肯定是0，然后还缺4位，我们通过当前的PC寄存器计算而得。

先将PC寄存器的内容加4，得到的这个32位数，取其高4位，

和26位地址以及最低的两位的0连接起来，构成了一个32位的数，这就是转移的目标地址。

那我们可以看到，在这个目标地址的计算方法只与两个内容有关，

1是当前PC的值，2是这条指令本身的编码，那我们结合处理器的结构图进行分析，

当这条j指令处在取指阶段的时候，指令存储器会送出指令的

编码，如果我们增加一些简单的电路，就能判断出这是一条j指令，

同时我们将这条指令编码当中的第26位取出来，在低位加2个0， 然后在这个PC更新的部件当中，已经会完成PC加4的工作，

那我们再将这个PC加4的高4位取出，然后拼接而得到一个32位的数，

这就是我们要更新的PC的值，也就是这条转移指令的目标地址。

那这些工作都可以在一个时钟周期内完成， 并将这个要更新的PC值送到PC寄存器的输入端，

那在下一个时钟上升沿到来的时候，PC寄存器就可以采样到这个要更新的PC的值，

那在下一个时钟周期，PC寄存器送出的就是这条转移指令的目标地址了。

这样对这条j指令来说，它所需要的转移目标地址在取指阶段就可以获得， 流水线不用停顿。

我们再看来无条件的间接转移，也就是jr指令，这条指令是一条R型指令，

它的转移目标地址的计算方法是用指令编码当中的rs域指定一个

一个寄存器的编号，用这个编号从寄存器堆当中，取出对应寄存器的内容。

那我们还是结合这个流水线的结构图来看。

因为这是间接转移，所以在取指阶段得到指令编码之后，并不能获得转移的目标地址，

因此取指部件至少要等待一个周期。

那当这条JR指令进入到译码阶段后，指令编码当中的 rs域就会送到寄存器堆，然后得到对应的寄存器的内容，

那如果我们在这里把busA这个信号连接到PC的更新部件，那在JR这条指令的

译码阶段结束的时候，转移的目标地址就可以送到PC 寄存器的输入端了。

当下一个时钟上升沿来临的时候，这个地址就可以存到PC寄存器当中去，

然后在下一个时钟周期，送到指令存储器，因此对于这条指令来说，因为

我们在译码阶段才能获得转移目标地址，所以流水线需要停顿一个周期。

那暂时就先这样，我们先接着看其他指令。

条件转移指令，它是一条I型指令，这条指令目标地址的计算方法是这样的。

首先比较rs和rt所指向的寄存器的内容，如果它们相等，它们目标

地址是在指令编码当中的16位立即数，进行符号扩展，然后乘以4， 再加上当前PC的内容，再加4，

而如果这两个寄存器的比较结果是不相等，那新的PC的值就只是当前PC值加4，

那不管寄存器比较的结果是否相等，那这个新的PC的值都只跟当前的PC值

和指令编码的内容相关，而这两项内容在取指阶段都是可以确定的。

所以这么看来，目标地址的生成不会造成流水线的停顿， 而问题在于，是否要转移，这个条件的判断，

我们还是结合结构图来看一看。

因为要判定转移是否成立，需要比较两个寄存器的内容， 而寄存器的内容，我们只能在译码阶段才能获得，

这样与刚才的间接转移类似，我们也得让流水线停顿一个周期，才可以获得这两个寄存器- 的内容。

但是与刚才间接指令不同的是，即使到译码阶段的结束，我们依然不能知道转移的条件是- 否成立，

因为我们还需要到执行阶段，将ALU来对这两个数进行比较，从而得到比较的结果。

所以在这个结构下，我们需要让流水线停顿 两个周期，才能知道转移条件的判定结果。

其实要等到执行阶段结束，无非是要对两个32位数进行比较，

而比较两个数相等是一个非常简单的功能，不需要用到ALU这么复杂的部件， 那我们就可以在译码阶段进行一些小的改造。

我们在寄存器堆的输出，busA和busB这两个信号给它连接一个额外的比较电路，

这个电路是很简单的，速度也很快，不至于影响整个译码阶段的时间。

那我们把比较的结果再送到PC的更新部件，那这样在译码阶段 结束的时候，我们就可以将下一条指令的地址送到PC寄存器了。

那经过这样的改动，条件转移指令也只需要让流水线停顿一个周期， 就可以让指令正确地执行了。

那通过上面的分析，我们发现，不同的转移指令带来的控制冒险是不一样的，

那经过我们的改进之后，无条件的直接转移可以让流水线不停顿的。

而无条件的间接转移以及条件转移都不得不让流水线停顿一个周期， 才能消除控制冒险的影响。

但是如果我们还想进一步地消除这个影响，不让流水线停顿，是否可以做到呢?

那我们就来介绍一个简单的方法，就是延迟转移技术，

我们结合这张代码来进行分析，这里有一条条件转移指令， 在它之前依次是减法、加法和抑或指令，

那按照通常的规则，这些指令依次进入流水线执行，当执行到这条beq指令的时候，

如果t1、t2两个寄存器的内容相同，就会跳到Next所指向的地方，

在beq进入流水线之后，必须还需要再等一个周期，才能知道转移条件是否满足，

那流水线必须停顿一个周期，那我们现在就是想办法把这个浪费的周期重新利用起来。

既然我们从硬件上现在无法解决这个问题，那我们不妨就修改这指令行为的定义，我们就规定- ，它之后的

> 那条指令是一定会被执行的，如果是这样，流水线中就不会出现被浪费的那个周期了。
>
> 但是我们还要注意，这样的修改不应该改变程序本来想要达到的结果，
>
> 所以我们就需要修改一下这段代码，我们要在这个beq指令之后填上一条一定会被执行- 的指令，
>
> 那我们只能往上走，但是之前的这条减法指令和加法指令，它们的运算结果正好
>
> 是beq指令所要比较的这两个寄存器， 所以这条加法指令和减法指令必须在beq指令之前执行。
>
> 而我们再往上看，这条抑或指令与我们的判定条件没有关系，
>
> 现在我们就把这条抑或指令挪到beq指令之后，因为我们现在已经修改了转移指令的定义，- 那我们在流水线的
>
> 硬件结构上，就可以确定地将beq之后的这条指令进入流水线， 而当这条抑或指令完成取指进入译码阶段的时候，
>
> 这条beq指令的条件判断也已经完成。
>
> 如果条件成立，这时候就可以从Next所指向的这个地方开始取下一条指令了， 否则也可以顺序地取下一条指令，
>
> 但不论是哪一种情况，流水线都不会发生停顿。

> 对于一个流水线处理器来说， 流水线级数越深，流水线结构越复杂，
>
> 转移指令带来的影响就会越大， 而且，我们现在也没有非常完美的解决方案，
>
> 但也正因为如此，在如何处理转移指令这个方面， 有了很多很有意思的研究成果，
>
> 大家如果感兴趣，可以进一步深入地学习和了解。