# 分支预测

术语：

# 1 概述

学习分支预测技术，首先我们需要了解分支预测产生的原因---控制冒险。为了确保预取正确指令（即取指操作取出正确的指令）所造成的延迟，我们称之为控制冒险。

以五级流水为例，当执行分支跳转指令的时候，为了正确的预取指令（此时下一条指令可能是跳转地址的指令也可能是顺序执行的指令），我们识别当前为分支跳转指令后，暂停流水线，直到判断出下一条指令的地址是分支指令跳转目标地址还是顺序执行地址（PC+4）。

若在提交阶段（访存阶段）才将正确的地址传递给指令计数器，那么此时暂停了两个周期，即控制冒险产生了两个周期的延迟。

为什么不是三个周期？因为**只有在译码阶段才能判断出当前指令是否是分支跳转指令。**此时取指阶段已经取出来下一条指令，这等效于假设分支跳转指令不发生跳转。这种“假定分支跳转指令不发生”的方法，其实是静态分支预测技术。

同样是基于五级流水，假设在提交阶段将正确地址传递给指令计数器。当分支指令到达提交阶段时，此时流水线按照分支指令永远不会跳转的原则取出了顺序的三条指令。这种策略是预测分支总是不发生的静态分支预测。若此时分支指令不跳转，那么我们预测正确，流水线继续执行。如果此时分支指令跳转，那么我们预测错误，需要清空取出来的错误的指令，即清空取指阶段，译码阶段跟执行阶段（没有分支延迟槽指令）。若分支指令跳转的几率为50%，那么每当处理器遇到分支指令时，总有一半的几率损失掉三个周期，这显然影响了处理器的性能。

当然我们可以采用其他的办法，例如缩短延迟法跟动态分支预测。缩短延迟法在译码阶段判断是否跳转并执行分支指令。这样，对于五级流水，结合延迟槽指令机制，每次分支指令都能正确的跳转，没有损失周期（产生空泡）。这也是我们计算机组成原理实验采用的办法。

我们可以发现，在五级流水中，缩短延迟法结合延迟槽指令机制能够产生很好的效果，没有浪费周期，功耗也比较低。但是这种策略也存在不足，现代处理器通常是超标量处理器。超标量处理器往往会花费多个周期来执行取指操作。假设采用n个周期来执行取指操作，那么译码阶段处于第n+1个周期。如果仍然采用缩短延迟法跟分支延迟槽指令这种策略，此时译码阶段是分支跳转指令，前n-1个周期以“假定分支指令永远不发生”的策略进行取指，也就是顺序取指，第n个周期是延迟槽指令。那么如果分支指令会发生跳转，将会浪费n-1个周期（产生n-1个空泡），显然这严重影响了处理器的性能。

所以我们发现，缩短延迟法对IPC有较大的影响。而且我们在实践中发现，对于五级流水，缩短延迟法对主频也有较大影响。由于将分支指令的判断提前到译码阶段，所以会产生数据相关问题，例如，对于指令序列：“D: jr $1,E: addu $1, $2, $3”，产生了数据冲突，那么为了得到跳转地址，首先要经过ALU计算，然后前推到译码阶段，然后经过多路选择器，才能将下一条指令的正确地址传递给指令计数器。这条路径也有可能成为关键路径，造成较大时延压力。

所以我们可以发现，如果能够更早的确定分支指令跳转的方向，就能够减少因分支指令要跳转而浪费的周期数，从而能够提高处理器的性能。例如，在取指令的第一个周期执行分支指令。这也是我们分支预测要实现的目标。

但是这样就产生了一个问题，因为我们最早也只能在译码阶段才能确定当前指令是否是分支跳转指令。为了在取指阶段就确定当前指令为分支跳转指令，超标量处理器基于指令地址，预测当前指令是否是分支跳转指令。因为从当程序开始执行后，指令地址也已经固定下来。我们完全可以根据指令PC值来判断当前指令是否是分支指令。但在我们的实现中，也就是比较基础的五级流水结构，并不需要考虑这个问题。在我们的实现中，取指操作需要花费一个周期，并且实现了分支指令延迟槽机制，所以我们最早只能在译码阶段，也就是第二个周期，进行跳转。此时我们已经可以直接解码指令，从而判断当前指令是否是分支指令。所以我们并不需要去预测当前指令是否是分支指令。这种情况下，我们实现的分支预测的主要意义就是教学使用和减轻时延压力（当分支指令数据相关成为关键路径的时候）。

为了实现分支预测，我们首先要分析下分支指令的性质。当处理器执行分支指令时，其实完成了两部分逻辑：一个是确定方向，即确定分支指令是否跳转，一个是确定跳转地址。所以我们分支预测的工作就是预测方向和预测目标跳转地址。

1. **方向预测**。在MIPS指令集中，分支跳转指令可以分为两类，一类是无条件跳转指令j，jal等，这类指令总是跳转；另一类是有条件跳转指令beq等，只有在满足条件的情况下，才能跳转。有条件跳转指令的跳转方向虽然不是固定不变的，但是有一定的规律，从结果来看，可能跟分支指令的局部历史跳转记录有关，又或者跟全局分支指令的跳转记录有关。我们通过捕捉这种规律来预测有条件分 支跳转指令跳转的方向。
2. **目标地址预测**。同样，我们可以根据目标地址类型将分支跳转指令分为两类，一类是直接跳转地址指令beq，j等，这类指令编码的特定域中保存着目标跳转地址相关信息；另一类是间接跳转地址，这类指令的目标跳转地址存储在通用寄存器中，通过读取通用寄存器来获取相关地址信息。
   1. 关于直接跳转指令的地址预测，由于直接跳转指令的目标地址是指令相关域的值作为偏移量加上当前指令地址得到，所以是固定的值，所以我们可以使用一个cache保存直接跳转指令的目标地址，此cache通过PC值进行索引。当识别到当前指令是分支跳转指令时，结合用PC从cache中索引到的地址，可以在取指阶段的第一个周期进行跳转。
   2. 关于间接跳转指令的目标地址预测。我们可以将间接跳转指令分为RETURN指令（弹栈指令，调用子程序后跳回主程序的指令）跟其他类型间接跳转指令。在MIPS中，RETURN指令即“jr $31($ra)”，其中31号(ra)通用寄存器是用来存储函数返回地址的。这类指令通常也有比较强的规律性，也容易预测。但是我们很难预测其他类型的分支跳转指令的目标跳转地址，或者说比较麻烦，因此，处理器一般不推荐使用间接跳转指令。

接下来的内容将围绕方向预测跟目标地址预测展开。

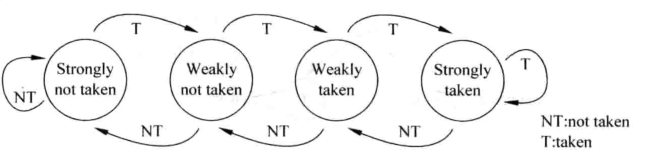
# 2 方向预测

鉴于《超标量处理器设计》这本书已经描述的很详细了，在此不做赘述，只是整理总结一下方向预测的技术以及五级流水中的实现方式。由于无条件跳转方向一定是跳转的，所以此部分内容主要是预测有条件跳转指令的方向。

知识储备：一位饱和计数器

## 2.1 基于两位饱和计数器的分支预测

基于两位饱和计数器的分支预测是从一位计数器的分支预测演变而来的。一位计数器的思想是假设此次跳转方向跟上次分支指令跳转方向一致，从而做出预测。但是如果遇到方向变化频繁的分支指令，一位计数器的分支预测效果堪称灾难级表现，例如：“TTNTNT……”，鲁棒性极差。而基于两位饱和计数器的分支预测通过增加状态数，起到缓冲作用，有比较高的鲁棒性，可以将分支方向偶尔发生变化的情况过滤掉。



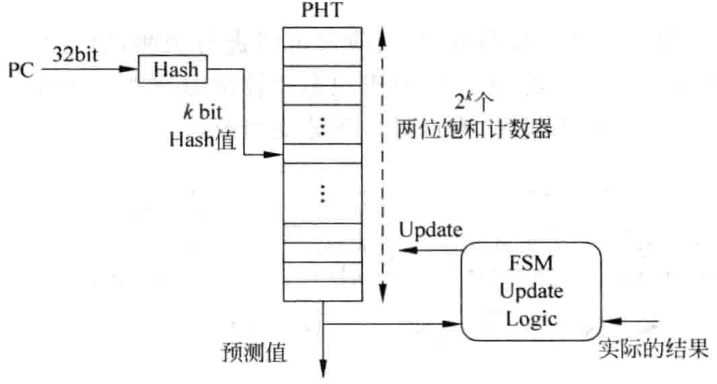
图：两位饱和计数器的**状态变化图**

接下来我们将以基于两位饱和计数器的分支预测为例，基于五级流水，简述分支预测的实现。

1. **分支预测模块内部预测技术。** 这里以基于两位饱和计数器的分支预测为例。我们的分支预测都是基于指令PC值来预测是否是分支指令，分支指令跳转方向以及分支指令目标地址。理论上讲，我们应该针对每一条指令（PC值）都保存一个饱和计数器，来捕捉跳转规律并做出预测。但这种做法是不切实际的，因为对于32位的指令地址，那么就需要2G的板上资源来实现表，这显然是不能忍受的。考虑到不是所有指令都是分支跳转指令，所以**我们可以用PC的一部分来索引由饱和计数器组成的表，这个表被称为PHT(Pattern History Table)。这样就可以大幅度的减少PHT所需资源。**

但是这样也会引入新的问题，可能多条分支跳转指令映射到同一个饱和计数器，那么可能干扰预测准确性，这种情况被称为**“别名”**。一个解决办法就是对PC进行哈希，用**哈希**后的值来索引PHT，以减少“别名”问题。此处应该认识到，其他非分支指令并不会对分支预测产生干扰，因为不会对PHT进行任何操作。

总结一下，我们通过将PC值进行哈希，然后根据哈希后的值索引PHT，得到饱和计数器的状态，若这个状态处于“Strongly Taken”或者“Weakly Taken”，那么意味着预测为“跳”，否则为“不跳”。关于预测表（此处为PHT）的更新，当我们预测正确或错误时，按照状态机规则对PC对应的饱和计数器更新即可。总体框架图如下所示。



图：基于两位饱和计数器的分支预测

1. **分支预测内部通路逻辑**
   1. **关于判断当前指令是否是分支跳转指令**。由于我们实现了分支指令延迟槽，所以我们最早只能在译码阶段进行跳转，同时译码阶段已经能够判断当前指令是否是分支指令，所以我们并不需要使用PC值来预测当前指令是否是分支指令。所以我们可以确定在五级流水中，在译码阶段执行分支指令的跳转。
   2. **关于预测分支指令的方向**。由于我们在译码阶段才执行分支指令的跳转，所以我们可以在取指阶段或者译码阶段对分支指令进行预测。建议在**取指阶段预测分支指令方向**，如果在译码阶段预测分支指令，一方面意义不大，因为可以已经可以确定方向，另一方面，查表操作时间较长，有较大时延压力。
   3. **关于判断预测结果是否正确**。由于判断有条件跳转指令是否跳转，需要读取寄存器内容，因此我们可以在译码阶段或者执行阶段判断方向预测结果是否正确。如果对主频有要求的同学，建议在执行阶段判断预测结果是否正确。因为如果在译码阶段判断方向预测结果是否正确，由于数据冒险，我们要将正确的数据前推到译码阶段，然后再执行相关运算判断条件是否成立，最后将正确的地址传递给指令计数器，这无疑增加了译码阶段的时延压力，对主频有严重影响。
   4. **关于处理错误预测（清空流水线、将指令计数器指向正确的地址）**。方向分支预测结果不一定总是正确的。当我们在译码阶段或者执行阶段发判断出方向预测错误时，需要对错误进行处理。在如果在译码阶段就判断预测结果错误，那么我们就要在执行阶段处理错误。如果在执行阶段判断出方向预测错误，那么就在提交阶段（访存阶段）处理错误预测。同样建议在提交阶段处理错误预测，若在执行阶段处理错误预测，那么意味着在译码阶段就要判断是否错误，上面已经提到过，这种方式有较大的时延压力；假设在提交阶段处理错误预测，此时在执行阶段发现预测错误，则在下一拍（即提交阶段）清空**取指，译码阶段**，同时将指令计数器置为正确的PC值，取出正确的指令。
   5. **关于分支预测的更新**。我们可以在执行阶段跟访存阶段更新PHT表。在超标量处理器中，由于执行阶段执行的指令不一定正确，此时的分支指令可能位于错误的指令序列中，所以一般在提交阶段进行更新。虽然我们是五级流水，但是考虑到异常例外处理的情况，也有可能错误的更新PHT，所以建议在提交阶段更新PHT。
   6. **关于指令计数器的更新。有关指令计数器的处理前面已经描述过。这里主要想强调的是嵌套分支跳转的情况。如果你是在访存阶段才处理错误预测。若预测错误，此时译码阶段很有可能也存在一条分支指令。调试时请注意这种情况。**

PS:综上所述，在体系结构分支预测实验中，**建议：在取指阶段预测是否跳转；在译码阶段执行预测结果；在执行阶段判断是否预测正确；在提交阶段处理错误预测和更新PHT**

## 2.2 基于局部历史的分支预测

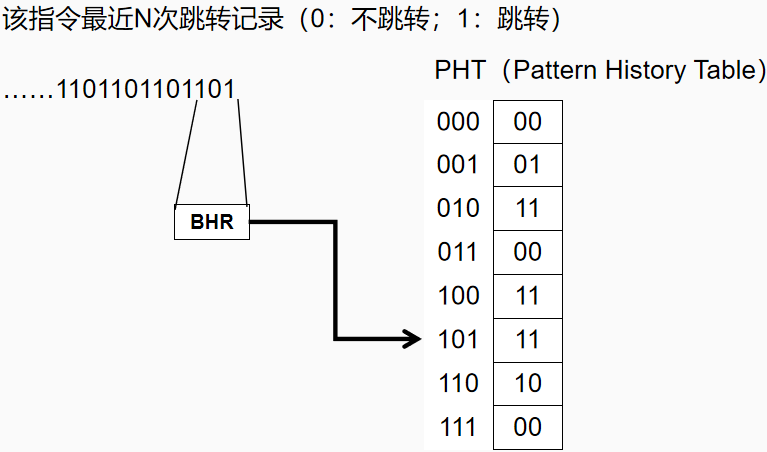
相比较于一位计数器的分支预测，基于两位饱和计数器的分支预测虽然已经有了很大的改善，但是仍然存在不足。若一条分支指令的历史跳转记录为“TNTNTNTNT……”，若采用基于两位饱和计数器的分支预测方法，假若现在处于“Weakly not Taken”状态，那么总是预测错误，命中率同样是0。又或者一条分支指令的历史跳转记录为“TTNTTNTTNTTN……”，最终状态会稳定在“Strongly Taken”中，每隔两个预测错误一次，准确约为60%。准确率同样不能令人满意。

我们不难发现，上述的跳转记录具有明显的规律。而饱和计数器有一种“投票表决”的味道，偏向于局部时间出现次数最多的哪一种情况，并没有很好的捕捉到这种规律。依据单条跳转指令的历史跳转记录**捕捉跳转规律**，从而预测该分支指令的方向，这种方法我们称之为**基于局部历史的分支预测**。

先考虑单条分支指令，我们使用一个寄存器**BHR(Branch History Register)**来**记录该指令的前面N次的历史跳转记录**。现在假设该分支指令的历史跳转记录为“TTNTTNTTNTTN……”，我们可以发现，最小循环周期为3，当前面三次跳转记录为“TTN”时，这次指令将会发生跳转，即“T”；当前面三次跳转记录为“TNT”时，这次指令将会发生跳转，即“T”；当前面三次跳转记录为“NTT”时，接下来将不会跳转，即“N”。这就是基于局部历史的分支预测的理论依据。

那么如何捕捉这种规律？我们可以针对每一种序列，训练一个两位饱和计数器，然后根据这个饱和计数器的状态预测跳转方向。形象的理解，假如你周期性的有三种心情“生气”、“高兴”和“抑郁”，恰好这儿有三张表，分别记录不同心情下的午餐。例如当你抑郁的时候，你就将你的午餐记录在抑郁对应的表上，记录的可能是“红烧茄子”或者“炝白菜”。假如你是一个很有规律的人，某一天，你又周期性的抑郁了，这时候店家一看你又网抑了，一查表，“哟，这家伙抑郁的时候最喜欢红烧茄子了”。不用你点餐，红烧茄子给你端上来。这就是基于局部历史的“菜单”预测。

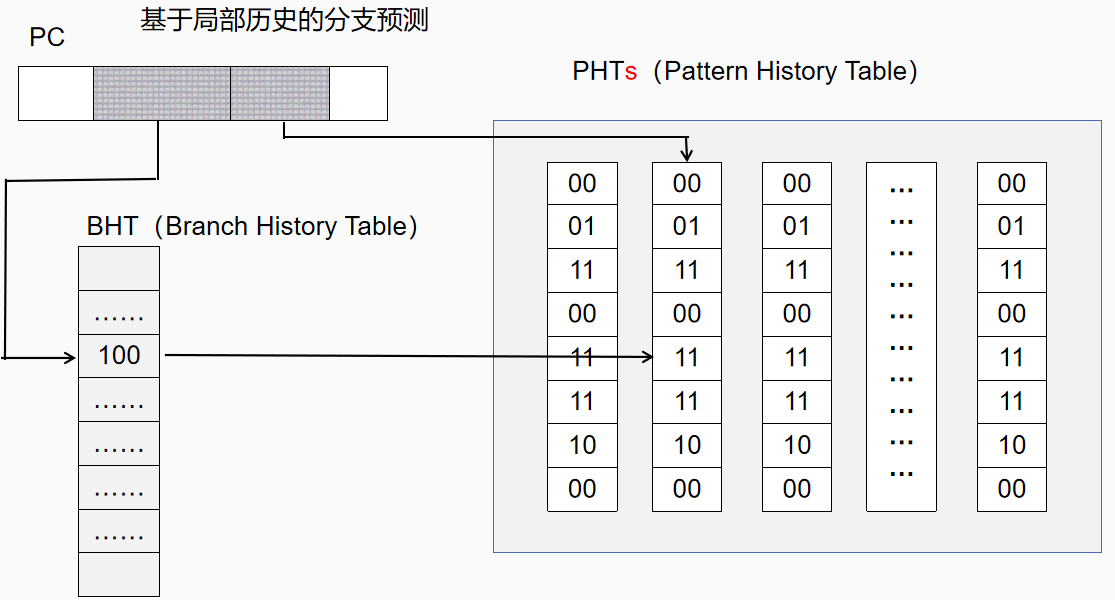
考虑单条分支跳转指令，基于局部历史的分支预测的工作机制示意图如下。



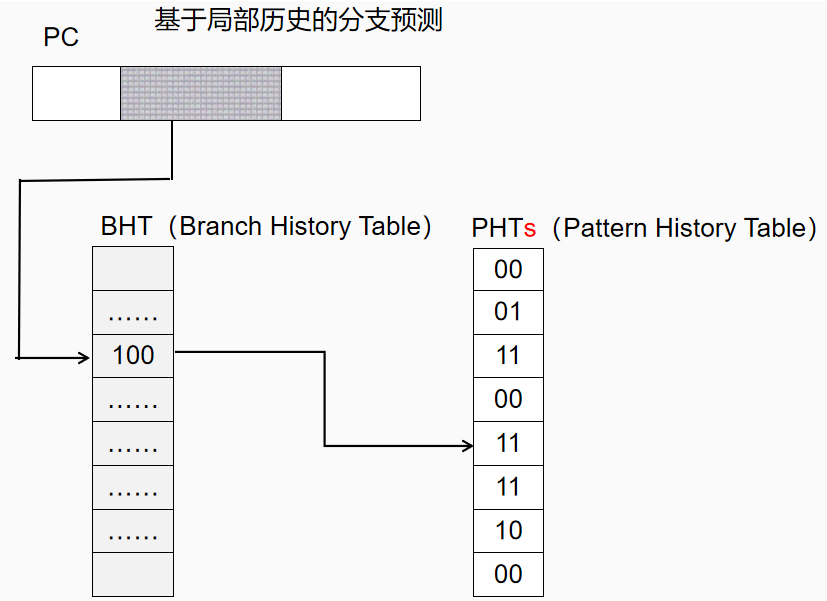
**图：基于局部历史预测单条指令的跳转方向**

上图中使用一个三位的BHR来保存最近三次分支跳转记录，然后使用BHR值来索引PHT，得到对应的饱和计数器，其中饱和计数器的状态为“11”，若代表“Strongly Taken”，则表示预测跳转。当然，BHR不一定是3位的，对于图中分支指令而言，历史跳转记录的最小循环周期为3，所以只要BHR的位数大于等于3，能够捕捉这种周期规律即可。当然，BHR并不是越大越好，一方面，一条分支指令需要对应一个BHR，过大的BHR会增加开销；另一方面，我们可以发现，对于3位的BHR，我们需要8个表项的PHT，但是我们仅用到了其中的3个表项，这种效率是极低的。除此之外，过宽的BHR寄存器也需要更多的训练时间。

从理论上讲，我们需要针对每一条分支指令，保留一个BHR（分支历史寄存器），并且训练一个PHT。但显而易见，这是极其奢侈且不切实际的做法。因为并每一条指令都是分支指令，所以我们**可以用PC的一部分来寻址BHT的表项**。同时，由图可知，并不是PHT的每一个表项都被使用，所以我们可以使用PC的一部分来寻址到哪一个PHT。并且，更为极端的一种情况，**所有分支指令可以共用一个PHT。如下图所示**。

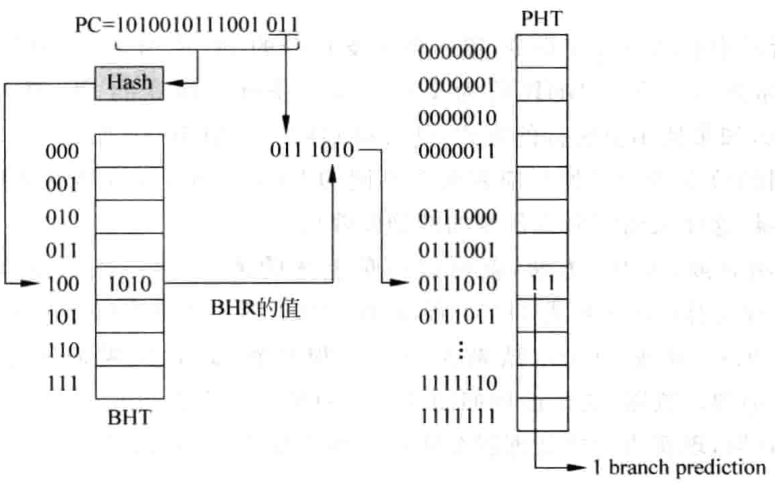


**图：使用PC的一部分进行索引**



**图：所有分支指令共用一个PHT**

同样，上述的操作可能会使多条分支指令映射到同一个BHT表项，或者映射到同一个PHT表项，产生别名问题。我们同样使用通过使用哈希来减少冲突。如下图所示。



**图：对PC值和BHR值进行处理，减少冲突**

**一个基于局部历史的分支预测的一个具体的实现：在预测阶段根据PC值索引BHT，得到对应的BHR，根据BHR的内容索引PHT，得到对应的饱和计数器，然后根据饱和计数器的内容得到预测方向；在译码阶段判断当前指令是否是分支指令，结合取指阶段得到的预测方向，判断是否跳转；在执行阶段判断预测结果是否正确；在访存阶段，如果预测正确，则根据访存阶段的指令地址索引到对应的BHR，将分支指令的跳转方向左移进得到的BHR中；如果预测错误，则清空流水线和将指令计数器置为正确的值，除此之外还要将分支指令正确的跳转方向左移进得到的BHR中。**

## 2.3 基于全局历史的分支预测

由于时间原因，且《超标量处理器设计》已经讲的相当详细，同时课上所讲主要内容也是基于全局历史的分支预测，此处不再赘述相关细节。不过由于《超标量处理器设计》相关概念跟PPT有所出入，建议先阅读“./对PPT上有关全局历史分支预测的一点解释.docx”。此处主要强调在实现基于全局历史的分支预测时，处理更新GHR时需要注意的地方。

我们同样可以形象化的理解基于全局历史的分支预测。在上一小节中点菜场景中，如果店家不再根据你的心情来预测你会点什么菜单，而是觉得你是一个没有原则的人，然后你确实是一个没有原则的人，**会根据前面用户的点餐来决定吃什么。这就是基于全局历史的“菜单”预测。（也就是说，不考虑是哪个指令的跳不跳转，不区分是哪个指令）**

### 2.3.2基于全局历史的分支预测的更新

基于全局历史的分支预测跟基于局部历史的分支预测极其相似。此处主要强调基于全局历史的分支预测的更新。主要是对GHR的更新和对饱和计数器的更新。

#### 2.3.2.1 对GHR进行更新

相对于基于局部历史的分支预测，基于全局历史的分支预测依赖于最近执行的分支指令。因此，及时更新GHR是必要的。有三种更新方案。

1. 在预测的时候根据预测的结果更新GHR。
2. 在执行阶段更新GHR。
3. 可以在提交阶段更新GHR。

可能听起来比较荒谬，为什么会在预测阶段根据预测的结果来更新GHR？但其实这种方法反而是最合理的。

首先我们先考虑在提交阶段更新GHR。这是一个极其安全的办法。当一条指令进入到提交阶段时，其要完成的操作是正确的并且确定的，对通用寄存器的更改是正确的。但是考虑到超标量处理器有很深的流水级数，当分支指令进入到提交阶段的时候，已经有很多条指令进入到处理器中，这其中可能就包含分支指令，那么这些分支指令并没有受到提交阶段的分支指令的影响。这显然是不符合我们基于全局历史的思想的。即使在五级流水中，在提交阶段更新GHR也是有一定的影响的。考虑指令序列“F: xxx; D: beq; E: xxx; M: bne”，此时D阶段beq指令不受M阶段的bne的影响。

举个具体的例子来讨论在GHR的更新，基于我们实现的五级流水，采用的是基于全局历史的分支预测。

* 1. 在取指阶段进行预测;
  2. 在译码阶段进行跳转;
  3. 在执行阶段判断预测结果是否正确;
  4. 在提交阶段（访存阶段）处理错误的预测和**更新GHR**；

在第一个时钟周期，分支指令b1进入译码阶段，若根据预测结果决定跳转（举例而已，是否跳转并不影响）。且跳转的目标地址的指令也是一条分支指令b2。

在第二个时钟周期，分支指令b1进入执行阶段，分支指令b2进入取指阶段。此时**处理器还未将b1的执行结果更新到GHR中，那么b2就将根据“旧”的，或者错误的GHR做出预测**。上面讲述的只是针对单发射的、顺序的、较少流水级数的处理器的影响，比如五级流水。如果处理器是超标量处理器，是多发射的，乱序的，较深流水的，若在提交阶段更新GHR，那么基于全局历史的分支预测失去了本来的意义，变成了一条咸鱼！

现在我们考虑第二种方法，在执行阶段更新GHR。我们首先分析第二种方法（在执行阶段更新GHR）和第一种方法（在提交阶段更新GHR）各自的优点跟不足。

首先，第二种方法比第一种方法提前不少周期（具体周期数依赖于具体实现）。对于五级流水，在执行阶段更新GHR虽然只是比在提交阶段更新提前了一个周期，但是后续分支指令都会“受益”于最新的GHR（虽然GHR的内容有可能是错的）。对于超标量处理器，提前了不少周期，相对于第一种方法，可以使更多的后续分支指令依赖于较新的GHR。、

但是，第二种方法也存在一个必须关注的问题。在执行阶段更新GHR，此时写入到GHR中的值是预测的，可能是错误的。这种更新我们称之为“推测”的更新。这里我们需要明确一下“在执行阶段更新GHR”的具体语义：**我们是在译码阶段准备好写入GHR的数据，然后在译码阶段到执行阶段的时钟上升沿，更新GHR。**之所以要强调这个问题，是因为有些同学可能会产生一些疑问：在我们给出的例子中，既然我们是在执行阶段判断预测结果是否正确，又在执行阶段更新GHR，为什么这个时候更新的GHR可能是错误的？这是因为在具体实现中，我们是更细GHR的逻辑是时序逻辑，**是时钟边沿触发（上升沿触发）**。那么虽然在执行阶段已经判断出预测结果是否正确，但是由于更新操作是边沿触发，所以我们无法在执行阶段将正确的值写入到GHR中，这里在执行阶段更新GHR的值是译码阶段接收到的预测值。当然，我们可以通过将时钟取反，从而在执行阶段用正确的值更新GHR（如果没记错的话，这种操作在上学期的计租实验中使用过），**但是这可能会对主频产生影响，而且代码风格不规范，因此不做推荐。**

上面只是从五级流水的角度对两种方案进行了对比。如果考虑超标量处理器，第二种方法还有新的问题。在此不做赘述，详情请参考《超标量处理器》。不过第三种方法也有类似的问题，将在后面详细讲述。

在此，我们应该对在执行阶段更新GHR跟在提交阶段更新GHR有了一定的认识。

我们发现，在**预测阶段更新GHR**，虽然也存在更新错误的情况，但是**能够更加及时的更新GHR**。为了更好理解，我们接下来将分析第三种方法（在预测阶段更新GHR）。

考虑五级流水，对于比较少见的“F: beq; D: bne;”的指令序列（甚至可能根本没有，个人所见到的分支指令后通常跟的是nop指令或者延迟槽指令），在执行阶段更新的策略使得F阶段的beq不能受益于D阶段的bne指令。但是如果采用在预测阶段更新GHR就可以避免这种问题。不过考虑超标量处理器，由于取值操作可能花费多个周期完成，所以在预测阶段更新GHR的优势就体现出来了。这是这两种方法的优势进行对比，接下来将比较这两种方法的不足。

首先这两种方法共有的不足是都**可能将错误的值写入到GHR中**，其次两种方法都有可能**在不应该更新GHR的时候更新GHR**。

第一种情况是后续的分支指令位于错误的分支路径中，并且更新了GHR。考虑五级流水，对于指令序列“F:xxx; D:beq; E:xxx; M:bne;”。若采用第三种方法，在预测阶段更新GHR，假设bne预测错误，也就是F、D、E阶段的指令都是错误的指令。由于我们在预测阶段更新GHR，此时beq的值已经被写入GHR中。若我们采用第二种办法，在执行阶段更新GHR，则不会出现这种情况（不考虑连续的分支跳转指令）。不过对于超标量处理器，这两种情况都有可能出现。

另一种情况就是在实现了精确异常的情况下，在处理异常前更新了GHR，无论是简单的五级流水，还是超标量处理器，都有可能出现这种情况。以五级流水为例，在提交阶段处理器异常，此时应该清空前面三个阶段，但是无论是在预测阶段更新GHR，还是在执行阶段更新GHR，都已经修改了GHR，改变了寄存器的值，违背了精确异常。

所以总结一下，**在提交阶段更新GHR的操作是完全正确的，但是不能及时的更新GHR，不符合基于全局历史的分支预测的思想；在执行阶段更新GHR能够比较及时的更新GHR，但是可能会错误的更新GHR；在预测阶段能够及时的更新GHR，但同样可能更新错误。在预测阶段更新GHR的效率最高，但带来的问题也是最多的。**不过，具体情况需要具体分析，比如对于五级流水，在执行阶段更新的效率跟在预测阶段更新的效率相差无几。为了便于实验的进行，基于现有的五级流水，从教学意义的角度出发，综合考虑，我们要求在实现在预测实现更新。

上面讲到，在预测阶段更新跟在执行阶段更新会带来问题。总结一下，可以简单分为两类问题，其中一种是将错误的值写入到GHR中，这类错误是在未能判断分支预测是否正确的情况下，将值更新到GHR中。另外一种情况就是在本不应该写入到GHR中的时候，将值写入到GHR中，这是因为分支指令位于错误的分支路径中或者位于异常指令后面。既然采用了在预测阶段更新GHR，那么接下来讨论如何解决这种方法带来的问题。由于计组实验4还未实现异常模块，并且五级流水流水级数较短，因此暂不考虑解决造成更新错误的第二种情况。

解决上述问题的方法有两种：1.提交阶段修复法；2.CheckPoint修复法。由于我们暂不考虑出现问题的第二种情况，因此，暂不讲述CheckPoint方法如何解决第二种情况的。详情请参考《超标量处理器》。

首先场景复现，我们现在设计是基础的五级流水，采用的是基于全局历史的分支预测，**在预测阶段预测跟更新GHR，在译码阶段判断是否要进行跳转，在执行阶段判断预测结果是否正确，在提交阶段判断对错误预测进行处理**。假设我们预测正确，那么我们更新GHR的操作就是正确的，那么处理器继续执行指令即可。假设我们预测错误，那么更新GHR的操作就是错误的，需要对此进行处理，下面有两种方法。

1. 提交阶段修复法。我们可以在提交阶段放置一个GHR（被称作Retired GHR）。只有当分支指令进入到提交阶段时，我们才更新这个GHR，如果在提交阶段发现分支预测错误，那么此时预测阶段的GHR的内容是错误的，此时我们**就可以将Retired GHR内的值写入到预测阶段的GHR**中。
2. CheckPoint修复法。由上面的例子，细心的同学可以发现，我们其实还可以额外增加一个寄存器GHR2同样记录分支指令的跳转记录。不过不同于GHR的是，每次更新GHR的时候，我们都将相反的值从右侧移入到GHR2中。这样，当我们在执行阶段发现预测错误时，意味着GHR中的值是错误的，但此时GHR2中的值是正确，此时我们就可以将GHR2中的值写入到GHR中。虽然在五级流水中，这种方法跟第一种方法是一样的，但是在流水级数较深的、或者超标量处理器，可以尽早的纠正GHR的错误。详情请参考《超标量处理器》。但是，这种修复法还是存在一定问题，因为此时不一定会更新GHR的内容。所以，这里需要结合提交阶段修复法，在提交阶段放置一个Retired GHR，做最后的保障。

#### 2.3.2.2 对饱和计数器进行更新

上文讲述到，饱和计数器这种技术的思想类似于“投票表决”，因此，即使更新不及时，即在提交阶段更新也不会对分支预测准确度产生较大的影响。所以我们**可以在提交阶段对其更新。**当然，也可以追求完美，及时更新饱和计数器，但是这样做的效费比低。

## 2.4 竞争的分支预测

在实现了基于局部历史的分支预测跟基于全局历史的分支预测后，实现竞争的分支预测就显得比较容易了。因为感觉《超标量处理器设计》这一部分内容已经讲的很详细了，我暂时没有什么拓展内容，所以建议同学们直接参考“《超标量处理器设计》--分支预测--竞争的分支预测”这一部分内容。

# 3 地址预测

由于时间关系，加上这部分实验内容属于“学有余力”部分，所以暂时没有讲述相关内容。当然，如果完成前面所有内容的同学，可以尝试做一下这部分内容。如果有什么问题，可以参考《超标量处理器设计》（因为主要内容也是参考的这本书），或者直接与我进行交流。