分支预测器的设计

纸上得来终觉浅，绝知此事要躬行。——《冬夜读书示子聿》

为什么需要分支预测

- 为了处理指令之间的相关问题，在 EX 阶段跳转是主流的实现方式
- 分支指令在指令流中虽然不那么常见，但往往是在关键的地方出现。
- EX 阶段一个跳转就需要冲刷掉两条指令——着实可惜

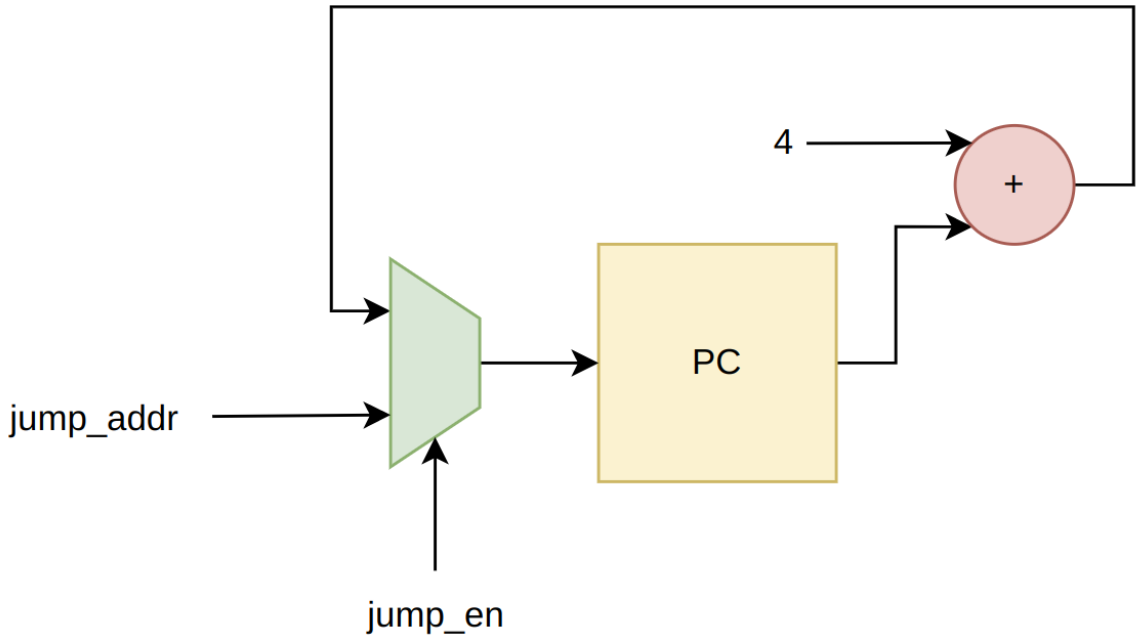
```
for(int i = 0; i < 10000000; i++);
```

对于一个只会因为控制冒险而冲刷两条指令的流水线，如果一个代码执行流程中有 15%的指令是确实跳转的分支指令，那么 CPI 就从 1 增加到 **0.85+0.15*3 = 1.3**

- 高性能处理器往往会使用超深流水线来减少延迟，一旦分支预测错误，代价可能是非常恐怖的

分支预测的设计基础

一个简单的分支预测器



- **PC+4**：预测方向（是跳转还是不跳）
- **PC+4**：预测地址（往哪里跳）

分支预测的时机

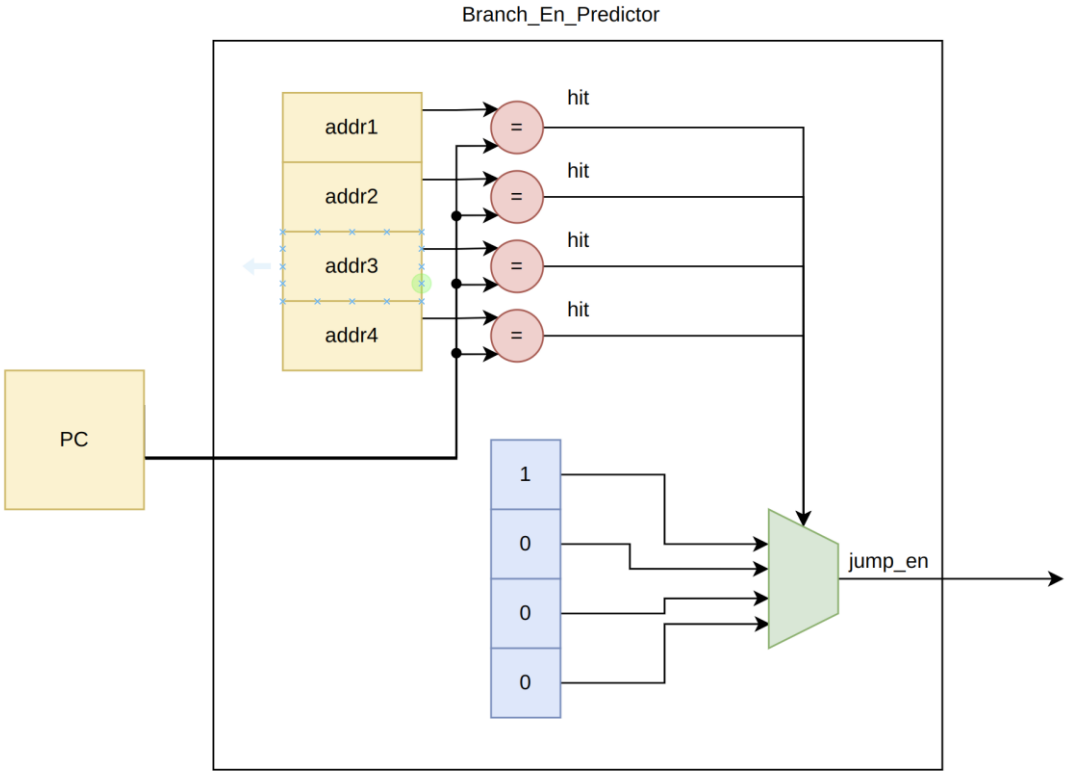
- 应该在取指阶段使用 PC 的值进行预测

预测分支方向

局部预测

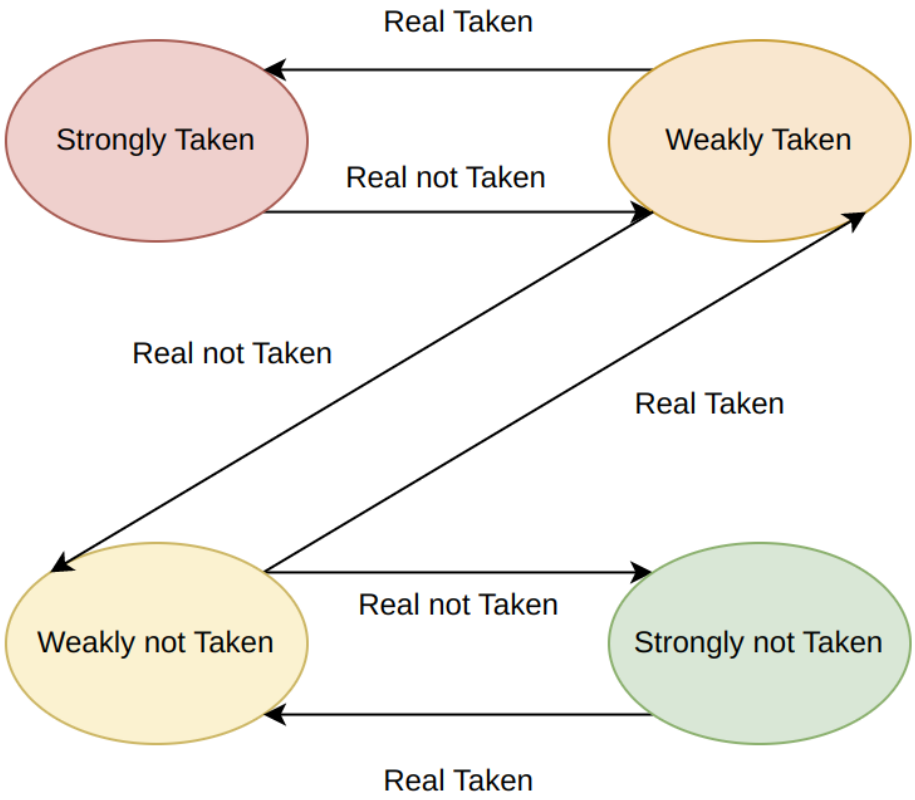
基于最后一次结果进行预测

- 使用高速缓存的思想：查询对应地址的数据是否在表中，如果在，那么直接进行预测
- 由于分支指令的分布是稀疏的，因此需要使用**哈希函数**对存储器的地址进行映射，或者使用**较高相连度的设计（全相连）**进行存储
- 参考设计 1：全相连
  - 这样的设计注定其不能对很多指令进行预测，因为全相连成本实在太高

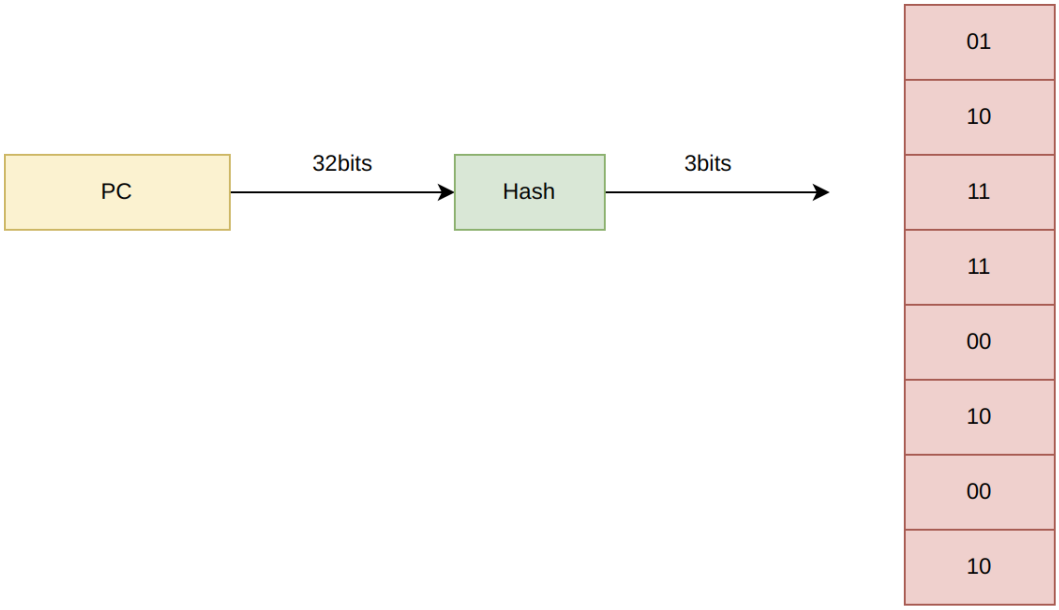


基于两位饱和计数器的预测

- 状态的设置：

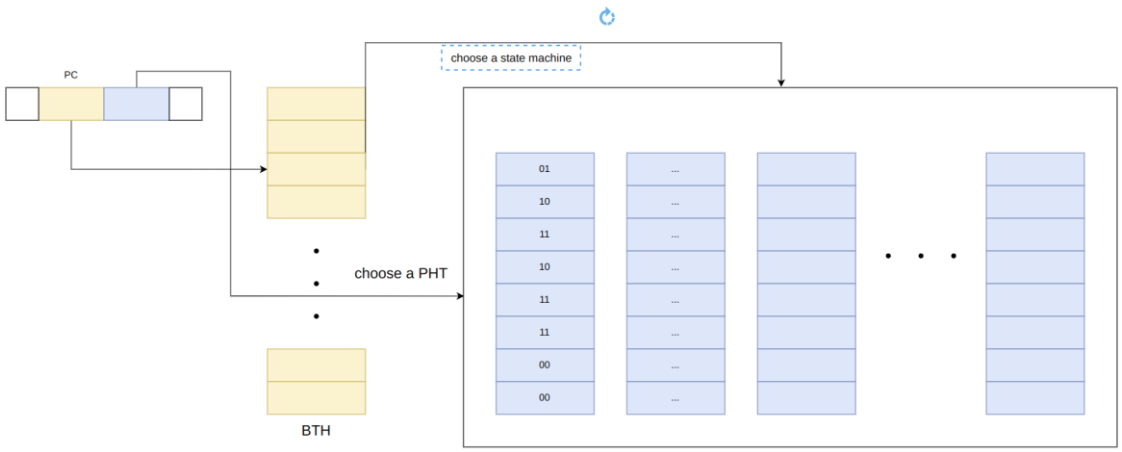


  - 在 两个 Taken 状态，每次预测其跳转，否则预测其不跳转
  - 根据 EXE 阶段的真实结果来更新状态
- 贪婪的思路：为每一条指令（也就是每一个 PC）都维护一个 2bit 计数器——显然资源消耗太大
- 只使用地址的 k 位来索引对应的状态：
  - 容易引起地址的冲突，这种冲突影响较大，可以通过**哈希函数**将 32 位地址映射到 k 位地址来缓解冲突



基于局部历史进行分支预测

- 假设一种情况：初始时是 01 状态，之后的跳转情况是 TNTNTNTNT，那么基于 2bit 的分支预测准确率接近于 0（这你会预测吗？）
- 历史信息记录表 BHT:记录一条跳转指令过去几次的跳转行为
- 用一条指令的历史信息来索引一个状态是更加合理的：
  - 使用指令地址来索引历史信息 BHT
  - 使用历史信息来索引跳转状态 PHT
  - 如果能为每一条指令都维护一个 BHT 和 PHT，那可以相当准确的进行预测，可惜……
- 可以维护一个 BHT 表和若干 PHT，用不同的地址部分进行索引：



- 卡跳转范围：普通跳转指令的跳转范围有限，我们可以维护较多的 BHT 表项，从而可以倾向于认为：对应到同一个 BHT 表项的跳转指令基本不冲突

Jal 和 Jalr 的跳转预测

- 和 branch 一起预测：简化了数据通路设计，但会对分支指令预测结果产生影响（一旦和 br 冲突那么会让准确率急剧下降）
- 单独预测：只要记录下了 Jal 的地址，就完全可以相信这个 PC 处会产生跳转（但也要小心自修改程序）

全局预测

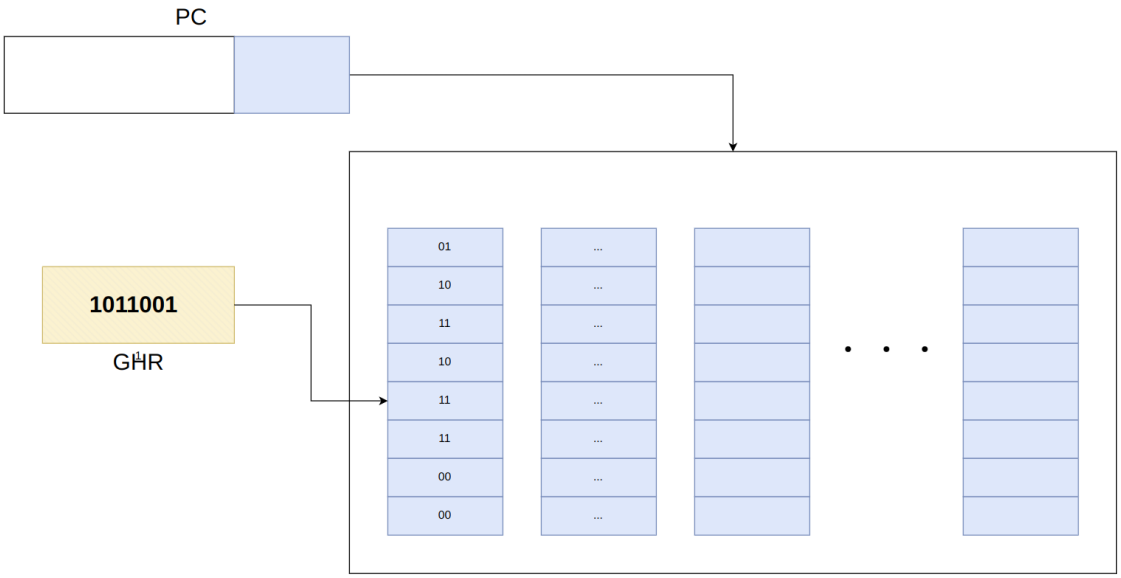
全局预测的例子

```
if(a == 0) temp = 1;
if(b == 0) temp = 2;
if(a != b) temp = 3;
```

假如前两个分支都没有跳转时，第三条分支指令一定会跳转，局部预测无法进行预测

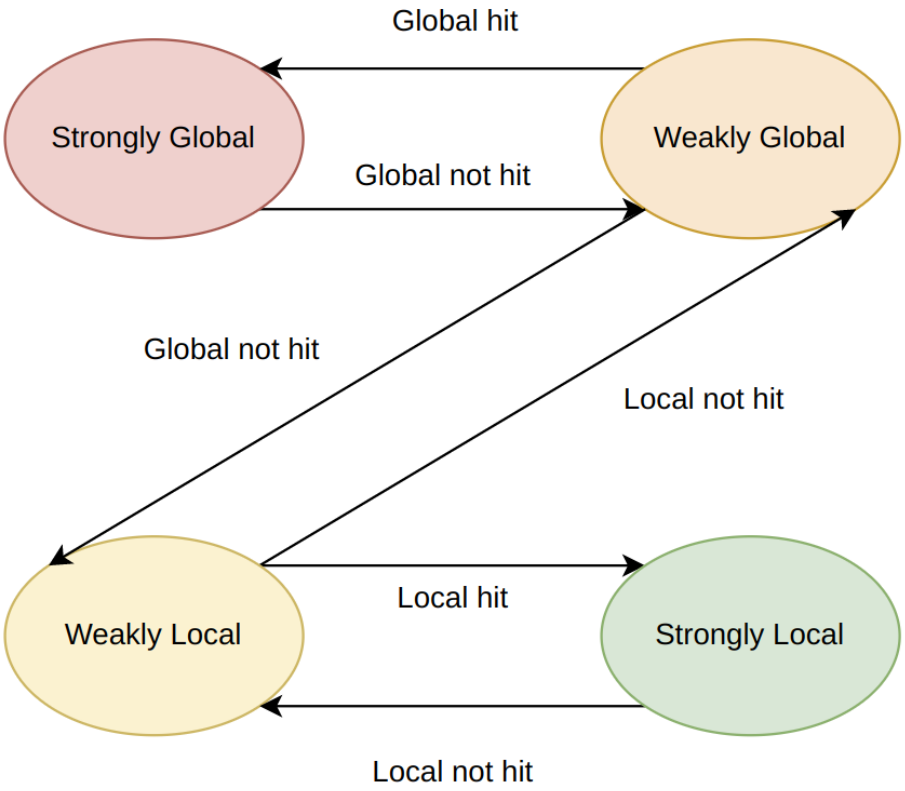
全局预测的实现

- 使用类似于局部预测的方法，使用全局历史寄存器（GHR）来保存最近几条分支指令的跳转情况，并使用 PHT 进行预测：



竞争预测

- 全局预测和局部预测都有其适用的地方，因此可以采用竞争预测来对预测正确性进行预测
- 竞争策略是多种多样的，这里介绍一种简单的竞争策略：

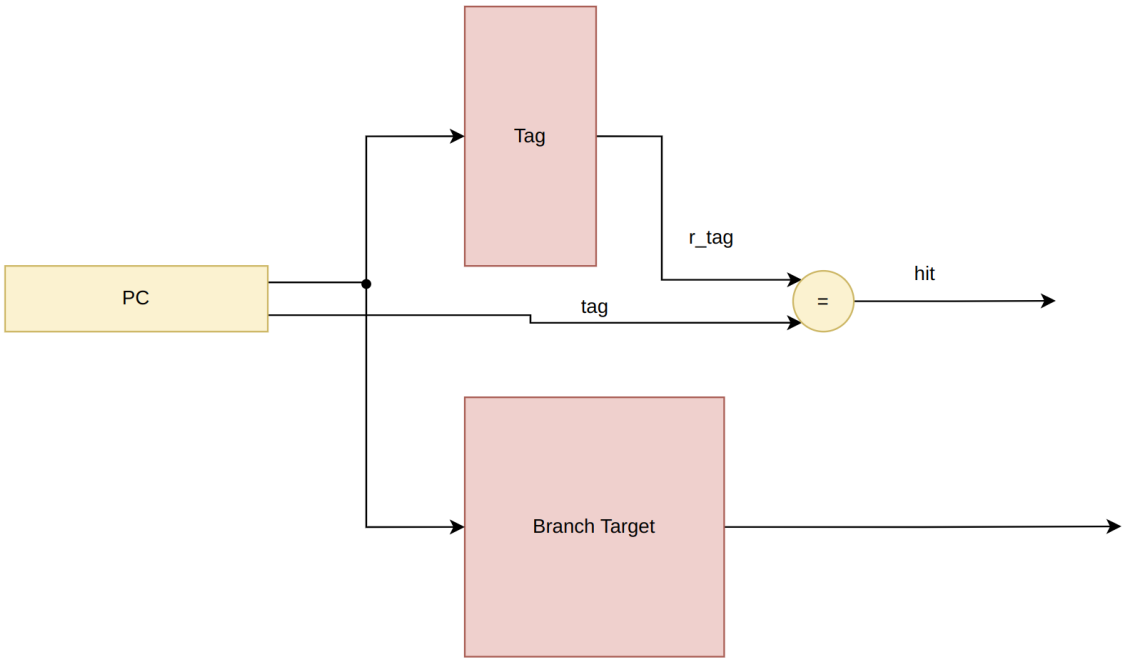


- 所有的分支预测策略必须要理论分析、实验验证，并指出你的优化针对哪些情况做了优化，这些特定情况是否有必要优化或是否为常见情况

预测分支地址

直接跳转地址预测

- 使用类似于 Cache 的方法保存跳转地址，这个表被称作 BTB（Branch Target Buffer）
  - BTB 使用 PC 的低位作为索引，高位作为 tag 判断是否命中
  - 若命中，则看方向预测器是否预测跳转，若是，则给出对应地址，否则给出 PC+4



- 由于分支指令的分布是稀疏的，且容易造成冲突，故可以选用合适的哈希函数对 PC 值进行哈希，同时也可以选择多路组相连的方法减少冲突

间接跳转（Jalr）地址预测

- Jalr 指令的跳转预测难度较大。一般编译器会将 Jalr 用作 return 指令，而 return 回去的位置往往难以预料：

```
    Quicksort(a, 0, i - 1);
    Quicksort(a, i + 1, n - 1);
```

- 一个简单的方法是不预测：不要使用直接跳转的方法，因为 Jalr 很容易干扰正常的分支预测
- 相信编译器：对于 99%的情况，哪怕是巨大如操作系统，Jal 的跳转范围也足够，因此编译器只会用 Jal 作函数调用，用 Jalr 作函数返回。因此我们可以用一个栈结构来保存 Jal 指令的下一条指令地址，在执行 Jalr 时弹出地址用作预测地址。同时在 BTB 中保存指令类型以方便预测

分支预测的纠正

- 之前的流水线中，只要是跳转就会冲刷流水线，但现在不同了，在执行单元，分支单元还需要知道几件事情：
  - 分支预测给出的预测跳转地址是什么
  - 分支预测器是否是对跳转地址进行预测
- 考虑自修改指令，如果一条 branch 指令被修改为算数指令，那么这样的跳转是错误的，ID 单元就可以纠正过来，并把 PC+4 进行复位
- EX 单元必须能纠正所有错误，只需要比较 NPC 和自己计算的地址是否相同。如果你的分支预测器比较好，可以在不跳转时给出 PC+4，那么就可以对每一个指令（哪怕不是分支指令）进行比对，这样更容易防止分支预测器的错误预测。