06.Branch Prediction

Two possible techniques: static branch predictions and dynamic branch predictions
In general when you have a branch you have a stall to fetch the instruction in the correct program counter. You want to predict the result of a branch instruction the soon as possible.
The performance of a branch prediction technique depends on:

- Accuracy measured in terms of percentage of incorrect predictions given by the predictor(You want it to be as close to zero as possible).
- Cost of an incorrect prediction measured in terms of time lost to execute useless instructions (misprediction penalty) given by the processor architecture: the cost increases for deeply pipelined processors(The longer the pipeline the costly is an error)
- Branch frequency given by the application: the importance of accurate branch prediction is higher in programs with higher branch frequency(With few branches you lose some cycles on millions of cycle, if you have many branches an incorrect prediction lead to restart computation several time)

Branch Prediction Techniques

There are many methods to deal with the performance loss due to branch hazards:

- Static Branch Prediction Techniques: The actions for a branch are fixed for each branch during the entire execution. The actions are fixed at compile time.
- Dynamic Branch Prediction Techniques: The decision causing the branch prediction can change during the program execution.
  In both cases, care must be taken not to change the processor state until the branch is definitely known.
  The execution has to be always correct, you don't have to change the effects on the system when you are optimizing. For the branches you don't change anything in the instruction until it is known as you then can continue safely.

Static Branch Prediction

Static Branch Prediction is used in processors where the expectation is that the branch behavior is highly predictable at compile time. Static Branch Prediction can also be used to assist dynamic predictors. Something static is more efficient as they are predictable.
Possible prediction techniques:

- Branch Always Not Taken (Predicted-Not-Taken): the sequential instruction flow we have fetched can continue as if the branch condition was not satisfied. If the condition in stage ID will result not satisfied (the prediction is correct), we can preserve performance. If the condition in stage ID will result satisfied (the prediction is incorrect), the branch is taken: we need to flush the next instruction already fetched (the fetched instruction is turned into

a nop) and we restart the execution by fetching the instruction at the branch target address having a One□cycle penalty

- Branch Always Taken (Predicted-Taken). You can anticipate the fetch of the instruction immediately after the jump and then commit if the branch is actually taken. The predicted-taken scheme makes sense for pipelines where the branch target is known before the branch outcome. If you don't known the target address you have to wait for it. In MIPS you donìt known the branch target address earlier than the branch outcome so this approach is useless.
- Backward Taken Forward Not Taken (BTFNT): the prediction is based on the branch direction(Backward-going branches are predicted as taken, Forward-going branches are predicted as not taken). Useful for loop iteration as you can save a lot of cycle if the loop is long enough.
- Profile-Driven Prediction: based on profiling information collected from earlier runs. The method can use compiler hints. You can assume that the data have a certain distribution.
- Delayed Branch: The compiler statically schedules an independent instruction in the branch delay slot. The instruction in the branch delay slot is executed whether or not the branch is taken. If we assume a branch delay of one-cycle (as for MIPS) we have only one-delay slot, but if you have a deeper pipeline you can have also something longer than 1 cycle that has to execute these independent instructions.
  The last three are very static assumption based on the direction of the jump.

Dynamic Branch Prediction

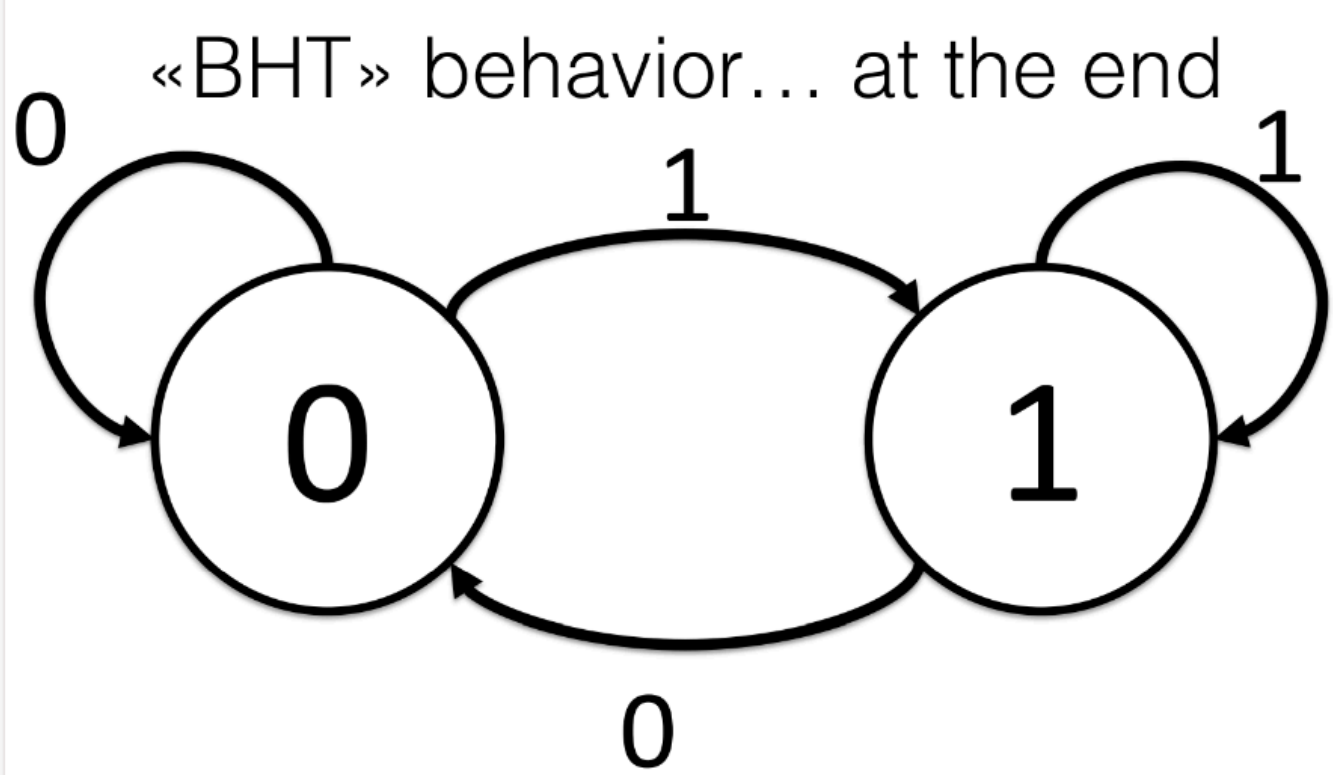
Basic Idea: To use the past branch behavior to predict the future.
We use hardware to dynamically predict the outcome of a branch: the prediction will depend on the behavior of the branch at run time and will change if the branch changes its behavior during execution.
We start with a simple branch prediction scheme and then examine approaches that increase the branch prediction accuracy.
Dynamic branch prediction is based on two interacting mechanisms:

- Branch Outcome Predictor (BOP): to predict the direction of a branch (i.e. taken or not taken).
- Branch Target Predictor (BTP): to predict the branch target address in case of taken branch
  Branch Target Buffer (Branch Target Predictor) is a cache storing the predicted branch target address for the next instruction after a branch
  We access the BTB in the IF stage using the instruction address of the fetched instruction (a possible branch) to index the cache

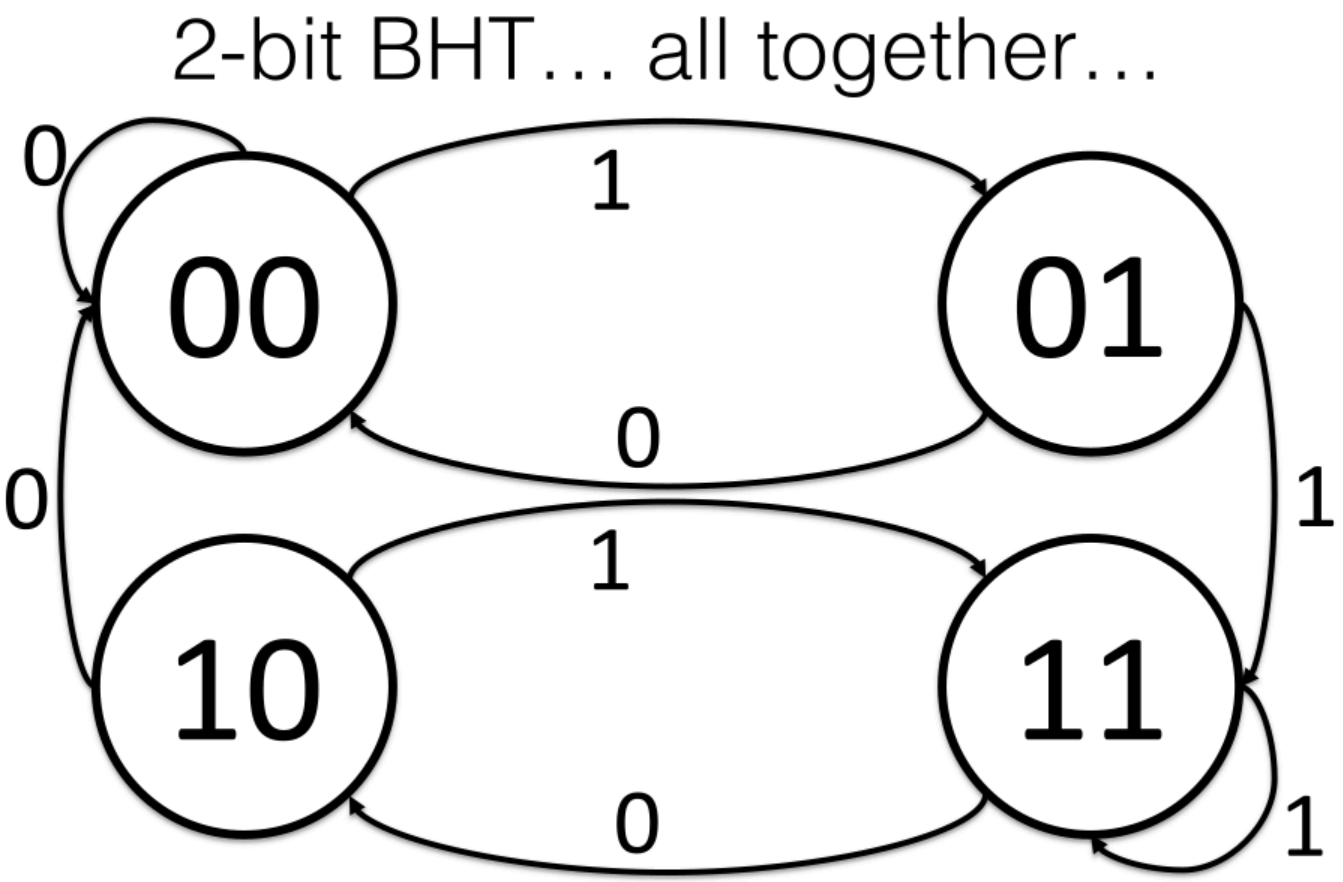
Typical entry of the BTB:

Exact Address of a Branch	Predicted target address

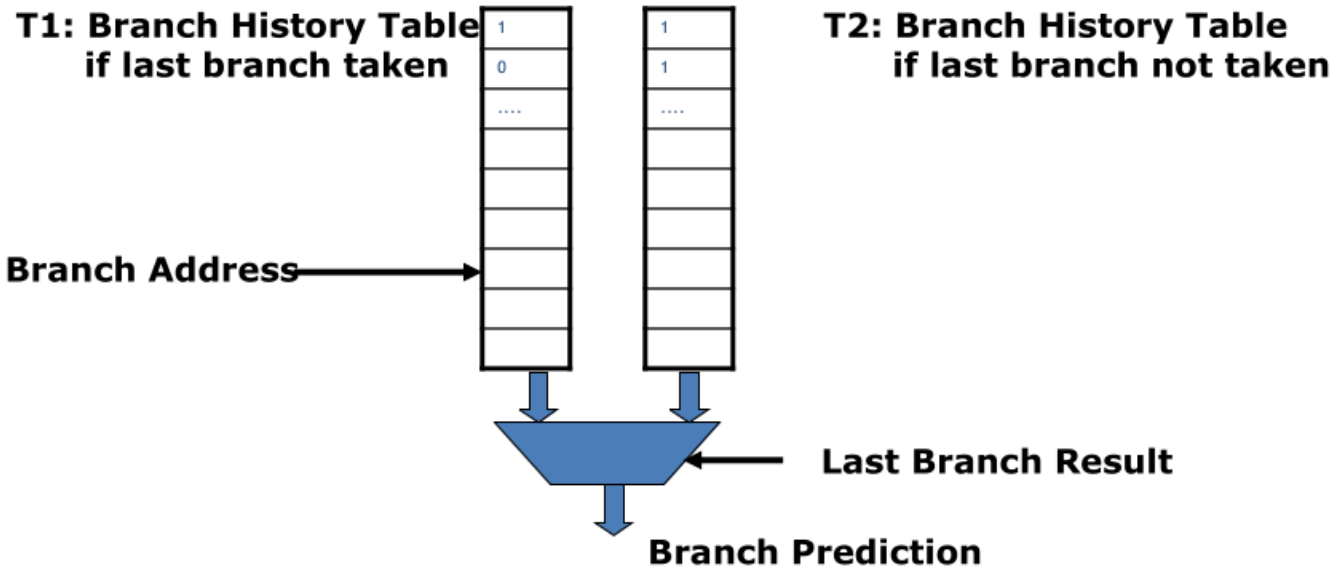
The predicted target address is expressed as PC- relative. You can somehow predict the outcome of the next operations.
Branch History Table: Table containing 1 bit for each entry that says whether the branch was recently taken or not. Table indexed by the lower portion of the address of the branch instruction.
Prediction: hint that it is assumed to be correct, fetching begins in the predicted direction. If the hint turns out to be wrong, the prediction bit is inverted and stored back. The pipeline is flushed, and the correct sequence is executed.
The table has no tags (every access is a hit) and the prediction bit could have been put there by another branch with the same low-order address bits: but it does not matter.
The prediction is just a hint!



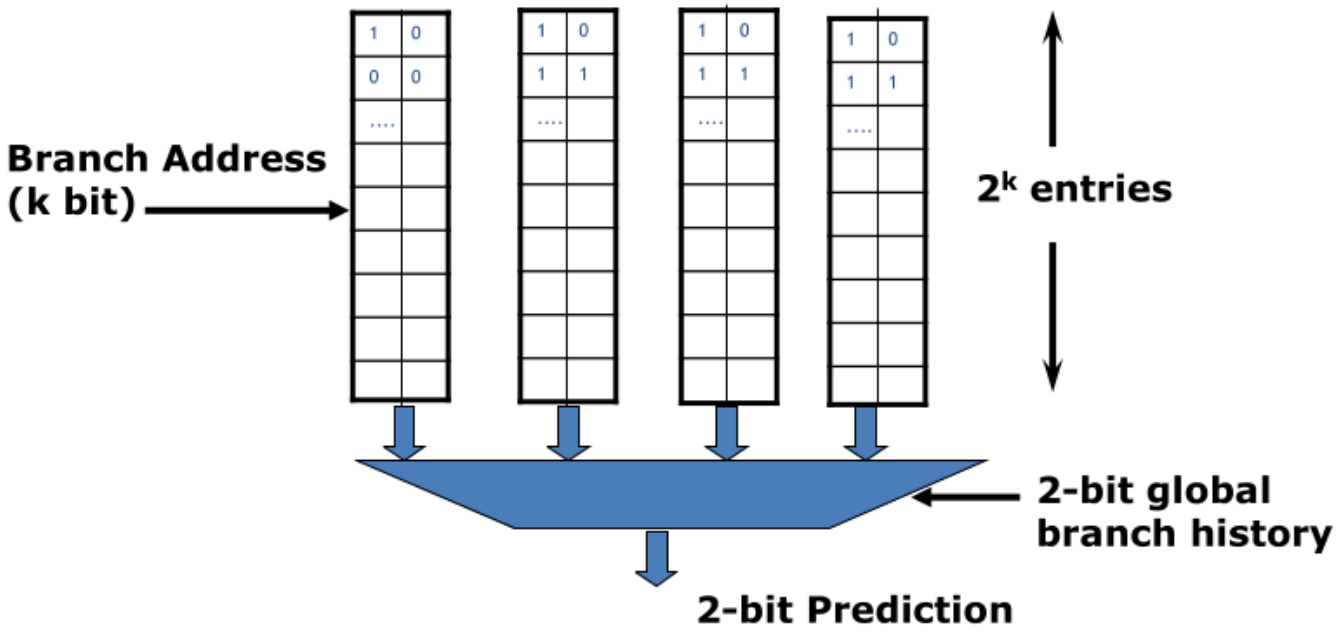
Accuracy of the Branch History Table: A misprediction occurs when the prediction is incorrect for that branch, or the same index has been referenced by two different branches, and the previous history refers to the other branch.
To solve this problem it is enough to increase the number of rows in the BHT or to use a hashing function(such as in GShare)



The BHT model can be generalised on n-bit saturating counter for each entry in the prediction buffer.
BHT predictors use only the recent behavior of a single branch to predict the future behavior of that branch.
Basic Idea: the behavior of recent branches are correlated, that is the recent behavior of other branches rather than just the current branch (we are trying to predict) can influence the prediction of the current branch.
Branch predictors that use the behavior of other branches to make a prediction are called Correlating Predictors or 2-level Predictors.



In general (m, n) correlating predictor records last m branches to choose from 2^m BHTs, each of which is a n-bit predictor.
The branch prediction buffer can be indexed by using a concatenation of low-order bits from the branch address with m-bit global history (i.e. global history of the most recent m branches).



Accuracy of Correlating Predictors

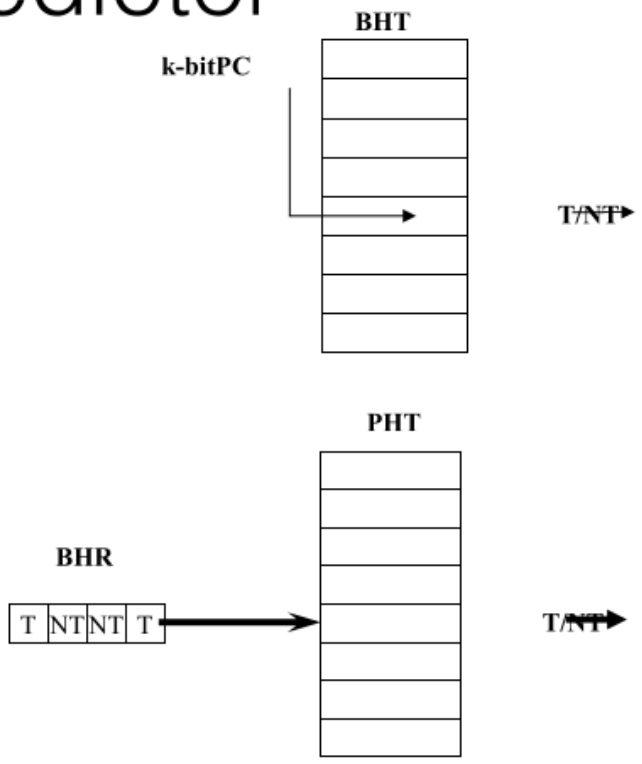
A 2-bit predictor with no global history is simply a (0, 2) predictor.
By comparing the performance of a 2-bit simple predictor with 4K entries and a (2,2) correlating predictor with 1K entries.
The (2,2) predictor not only outperforms the simply 2-bit predictor with the same number of total bits (4K total bits), it often outperforms a 2- bit predictor with an unlimited number of entries. (This means there is a correlation inside the program).

Two-Level Adaptive Branch Predictors

The first level history is recorded in one (or more) k-bit shift register called Branch History Register (BHR), which records the outcomes of the k most recent branches. It is more an history of the previous k entry
The second level history is recorded in one (or more) tables called Pattern History Table (PHT) of two-bit saturating counters. It is based on 2-bit count
The BHR is used to index the PHT to select which 2-bit counter to use.
Once the two-bit counter is selected, the prediction is made using the same method as in the two-bit counter scheme.
We use the value inside the shift register to index the pattern of the history table. That bit value is used as a 2-bit counter. We assume we have a pattern.

GA Predictor

- BHT: Local predictor
  - Indexed by the low-order bits of PC (branch address)
- GAs: Local and global predictor
  - 2-level predictor: PHT Indexed by the content of BHR (global history)



GShare Predictor

- **GShare:** Local XOR global information
  - Indexed by the exclusive OR of the low-order bits of PC (branch address) and the content of BHR (global history)