# Branches

Branches account for 20% of all instructions, so mispredicting then can have a major impact on processor performance. Understanding and improving branch prediction will greatly enhance performance.

**Branches in a Pipeline**

If a branch is not taken the PC just advances to the next instruction.
If the branch is taken the immediate value is added to the PC to advance to the new destination.
It is not known until the end of the ALU stage whether the branch is taken or not.
It is better to make a prediction, then the processor is at least right some of the time.

**Branch Prediction Requirements**

Branch prediction must be based on the available information, which is the PC. Using only the PC, the processor must try and correctly guess if the instruction is a taken branch and if so, what is the new PC.

**Branch Prediction Accuracy**

CPI = 1 + (Mispredictions / Instruc) * (Penalty / Misprediction)
Mispredictions/Instruction is dependent on the predictor accuracy
Penalty/Misprediction is dependent on the size of the pipeline

The deeper the pipeline the more important it is to have an accurate branch predictor.

**Performance with Not-Taken Prediction**

Refuse-to-predict waits to determine if the instruction is a branch
- Every branch costs 3 cycles
- Non-branches cost 2 cycles

A not-taken prediction just fetches the next instruction.
- Every branch costs 1 or 3 cycles (1 the branch is not taken, 3 if it is taken)
- Non-branches cost 1 cycle

Every processor will perform some type of prediction

**Predict Not Taken**

The Not-taken predictor is the simplest predictor, just increment the PC.
20% of all instructions are branches, 60% of branches are taken.
So Predict Not-Taken is correct 88% of the time and incorrect 12% of the time

**Why Do We Need Better Branch Prediction?**

The larger the pipeline, the greater the penalty, especially when the branches are detected later in the pipeline and there are multiple instructions per cycle.

There is a lot of waste from a misprediction, especially if it is a deep and/or wide pipeline.

**Better Prediction - How?**
All that is known about the instruction is the PC. A better prediction would require more knowledge about the instruction. Since this is not possible another method of improving prediction is needed.
Better prediction can come from using the history of the instruction to make a better prediction.

**Branch Target Buffer (BTB)**
The Branch target buffer holds the target PC, it is indexed by PC number.
<u>BTB Steps</u>

1. At Fetch the processor has the PC of an instruction
2. Looks in the BTB for this PC number
3. The processor reads out the prediction for the next PC.
4. This predicted PC is then compared with the actual PC that is generated later in the pipeline.
5. If the BTB prediction and the actual PC are the same -- this is a correct prediction. If the two are not the same -- this is a misprediction, and the correct PC is stored in the BTB.

**Realistic BTB**
The BTB needs to have a 1 cycle latency, so it needs to be small.
The BTB needs to hold only the instructions that are likely to be executed soon.
Use the LSB for indexing to the BTB.

**Direction Prediction**
Use a Branch History Table (BHT) to first determine if the branch is taken or not taken.
<u>Steps for BHT</u>

1. Use the LSB of the PC to index the BHT.
2. The simplest BHT will have 1 bit.
    0 = branch not-taken.
    1 = branch taken.
3. If BHT = 0, just increment the PC
   If BHT = 1, look up the new PC in the BTB

Since the BHT stores just 1 bit, it can store the history of a lot of instructions.
The BTB is only accessed when the BHT says the instruction is a taken branch.

**Problems with 1 bit Predictors**

The 1 bit predictor is very accurate, it works well for large loops with many iterations. This is because these branches tend to be always (or almost always) taken or not-taken.

1-bit predictors do not do well when the branch is not so predictable.
Each anomalous behavior with result in two mispredictions, so if the behavior changes a lot there will be many mispredictions.

**2-Bit Predictors (2BP or 2BC)**
A 2-bit predictor can reduce the number of mispredictions when there is a change in the branch prediction.
The two bits in the 2BP play the following roles:

MSB - the prediction, taken or not-taken
LSB - the conviction bit, sure, or not sure

00 - not-taken, strong
01 - not-taken, weak
10 - taken, weak
11 - taken, strong

A single anomaly will cost 1 misprediction, a change in behavior will cost 2 mispredictions.

| Predictor Present State | Actual Branch Event | Predictor Next State |
|---|---|---|
| 00 | not taken | 00 |
| 00 | taken | 01 |
| 01 | not taken | 00 |
| 01 | taken | 10 |
| 10 | not taken | 01 |
| 10 | taken | 11 |
| 11 | not taken | 10 |
| 11 | taken | 11 |

**2-Bit Predictor Initialization**
If the 2BP starts in a weak state (01 or 10) , this will lead to zero mispredictions if correct and only one misprediction if wrong.
If the branch is alternating between taken, not-taken, starting in a weak state will lead to more mispredictions. This condition is slightly less likely to happen.
Most predictors just start in the 00 state because it is the easiest to initialize.

*Every predictor has a sequence that will result in every prediction being wrong.*

**1BP → 2BP**
More bits in a predictor leads to more cost, without greatly improving the predictor accuracy.

**History Based Predictors**
Using history of the branch, the alternating patterns predictions can be learned.

**1-Bit History with 2-Bit Counters**
Steps for BHT with history:

1. Look up the BHT index with the PC
2. The BHT will have a 1-bit history, a 2-bit counter for history = 0, and a 2-bit counter for when the history = 1.
3. Look at the history bit. If the history bit = 0,
   use the first counter, if history bit = 1 use the second counter

**2-Bit History Predictor**
The pattern NNT mispredicts ⅓ of the time when a 1-bit history predictor is used.
Using a 2-bit predictor will solve this problem.
If a 2-bit history predictor is used the BHT now has 2 bits for history, and four 2-bit counters.

**History Based Predictors with Shared Counters**
The cost of adding counters to each entry in the BHT can become prohibitive. Sharing counters between entries is a good alternative since most of the counters are not used.

An N-bit History Predictor will predict all patterns of length <= N+1
An N-bit History Predictor will cost $N+2*2^N$ per entry, with most of the counters wasted.

Use a Pattern History Table (PST) to reduce the cost of the predictor.
A pattern of 2 uses two counters, which means two entries in the BHT.
A pattern of 16 uses 16 counters (16 entries in the BHT).

**PShare**
PShare = Private history, shared counters - good for small loops and predictable short patterns.

GShare = Global history and shared counters - good for correlated branches.

**GShare or PShare**
Use both in a processor.

**Tournament Predictors**
Using two different predictors requires choosing one, but which one?
A tournament predictor can predict which predictor has the better prediction.

| **GShare** | **PShare** | **MetaPredictor** |
|---|---|---|
| Correct | Correct | no change |
| Correct | Incorrect | count down |
| Incorrect | Correct | count up |
| Incorrect | Incorrect | no change |

**Hierarchical Predictors**

While a tournament predictor combines to good predictors, a hierarchical predictor combines 1 good and one okay predictor.
In a tournament predictor both predictors are updated with every prediction. With a hierarchical predictor the good predictor is only updated when the okay predictor is incorrect.

**Return Address Stack Predictor**

If a branch is taken, the return address must be predicted. There are different types of branches:

Conditional branches: the BTB will work for predicting RAS
Unconditional branches: the BTB will work for predicting RAS
Function Returns: use a RAS predictor

RAS - a small hardware stack that is dedicated to storing the return addresses for functions.
When the RAS is full, use the wrap around the approach for replacement.
The return instruction needs to be identified before decoding. There are two ways to do this, using a predictor or using predecoding from the prefetch.

**How do we know its a RET**

Use a predictor or predecoding to determine when an instruction is a RET.