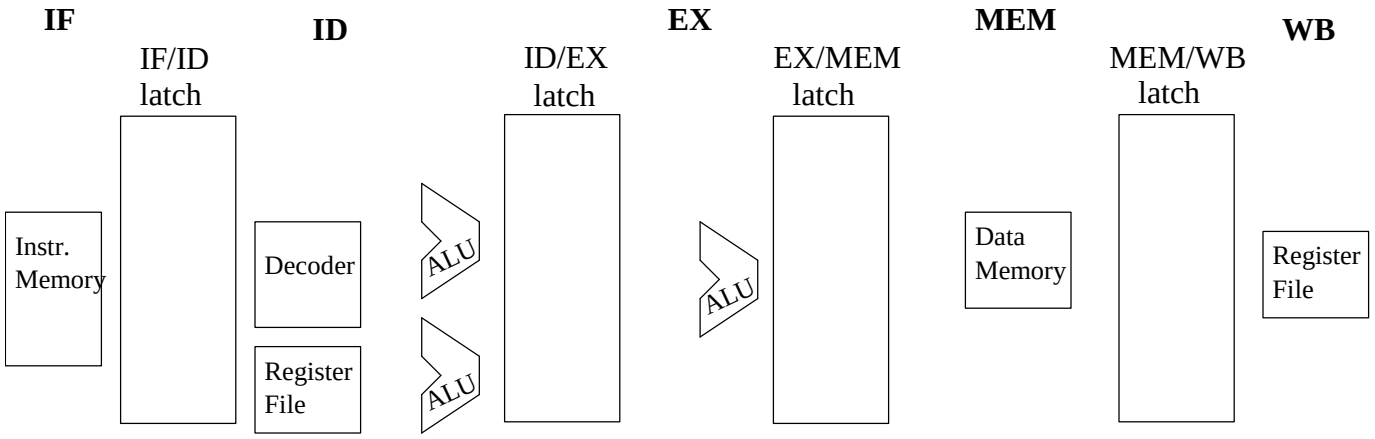
**Control Hazards** - branching causes problems since the pipeline can be filled with the wrong instructions.

| Stage | Abbreviation | Actions |
|---|---|---|
| Instruction Fetch | IF | Read next instruction into CPU and increment PC by 4 |
| Instruction Decode | ID | Determine opcode, read registers, **compare registers (if branch),** sign-extend immediate if needed, **compute target address of branch**, update PC if branch |
| Execution / Effective addr | EX | Calculate using operands prepared in ID<br>▪ memory ref: add base reg to offset to form effective address<br>▪ reg-reg ALU: ALU performs specified calculation<br>▪ reg-immediate ALU: ALU performs specified calculation |
| Memory access | MEM | ▪ load: read memory from effective address into pipeline register<br>▪ store: write reg value from ID stage to memory at effective address |
| Write-back | WB | ▪ ALU or load instruction: write result into register file |




Assembly-language Code Example -- Two possible "streams" of instruction

```
BEQ  R3, R8, ELSE
ADD  R4, R5, R6
SUB   R8, R5, R6
        .
        .
        .
ELSE: OR  R3, R3, R2
```

In what stage is the outcome of the comparison known?
ADD should not be executed if the branch is ***taken***

Assume the branch is taken:

| | Time → | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Instructions** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** |
| BEQ R3, R8, ELSE | | IF | **ID** | EX | MEM | WB | | | | | | |
| ADD R4, R5, R6 | | | IF | ID | EX | MEM | WB | ← changed to NOOP | | | | |
| ELSE: OR R3, R3, R2 | | | | IF | ID | EX | MEM | WB | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |

If the branch is ***taken***, then there is a ***branch penalty*** of 1 cycle, i.e., the instruction after a taken conditional branch (the ADD) must be thrown away by changing it to a NOOP (NO-OPeration) instruction. During clock cycle 3 the ID phase for the BEQ instruction was compare register R3 with R8, and computing the target address of branch (i.e., the location of the ELSE label).

If the branch is ***not taken*** and we continue to fetch instructions sequentially, then there is no branch penalty.

How could reduce the branch penalty in the pipeline above?

**Delayed Branching** (used in MIPS) - redefine the branch such that one (or two) instruction(s) after the branch will always be executed.

Compiler(and assembler) automatically rearranges code to fill the delayed-branch slot(s) with instructions that can always be executed. Instructions in the delayed-branch slot(s) do not need to be flushed after branching. If no instruction can be found to fill the delayed-branch slot(s), then a NOOP instruction is inserted.

| **Without Delayed Branching** | **With Delayed Branching** |
|---|---|
| SUB R5, R2, R1 | BEQ R3, R8, ELSE |
| BEQ R3, R8, ELSE | SUB R5, R2, R1 # delay slot alway done |
| ADD R4, R5, R6 | ADD R4, R5, R6 |
| . | . |
| . | . |
| ELSE: ADD R3, R3, R2 | ELSE: ADD R3, R3, R2 |

Due to data dependences, the instruction before the branch cannot always be moved into the branch-delay slot.

Other alternative to consider are:

| The Instruction at the Target of the Branch | |
|---|---|
| **Without Delayed Branching** | **With Delayed Branching** |
| LOOP: ADD R7, R8, R9<br>.<br>.<br>.<br>SUB R3, R2, R1<br>BEQ R3, R10, LOOP<br>MUL R4, R5, R6 | ADD R7, R8, R9<br>LOOP: .<br>.<br>.<br>SUB R3, R2, R1<br>BEQ R3, R10, LOOP<br>ADD R7, R8, R9 #delay slot<br>MUL R4, R5, R6 |

Can this technique always be used?

| The Instruction From the Fall-Through of the Branch | |
|---|---|
| **Without Delayed Branching** | **With Delayed Branching** |
| SUB R3, R2, R1<br>BEQ R3, R10, ELSE<br>ADD R8, R5, R6<br>.<br>.<br>ELSE: ADD R3, R3, R2 | SUB R3, R2, R1<br>BEQ R3, R10, ELSE<br>ADD R8, R5, R6 # delay slot<br>.<br>.<br>ELSE: ADD R3, R3, R2 |

Can this technique always be used?

**Note:** In the simple 5-stage pipeline of MIPS, delayed branching worked well. However, more modern processors have much longer pipelines and issue multiple instructions per clock cycle, so the branch delay becomes longer and a single delay slot is insufficient. Therefore, delayed branching is not used much today. Instead, more flexible dynamic branch prediction is used.

## Branch Prediction to reducing the branch penalty

Main idea: predict whether the branch will be taken and fetch accordingly

Fixed Techniques:

a) Predict never taken - continue to fetch sequentially. If the branch is not taken, then there is no wasted fetches.

b) Predict always taken - fetch from branch target as soon as possible
(From analyzing program behavior, > 50% of branches are taken.)

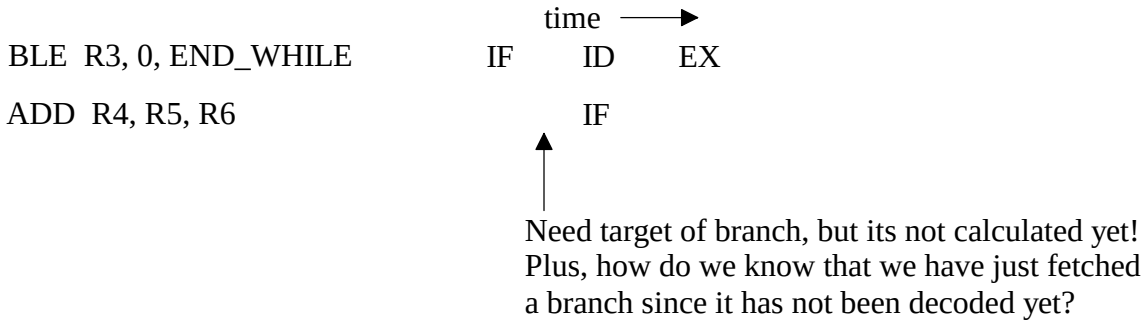
Static Techniques: Predict by opcode - compiler helps by having different opcodes based on likely outcome of the branch
Consider the HLL constructs:

| **HLL** | **AL** |
|---|---|
| | CMP x, #0 |
| While (x > 0) do | BR_LE_PREDICT_NOT_TAKEN END_WHILE |
| {loop body} | |
| end while | END_WHILE: |

Studies have found about a 75-82% successful prediction rate using this technique.

Dynamic Techniques: try to improve prediction by recording history of conditional branch
We need to store one or more history bits to reflect whether the most recent executions of the branch were taken or not.

Problem: How do we avoid always fetching the instruction after the branch?



Solution:
Branch Target Buffer (BTB) (I might use the terms “Branch Prediction Buffer”, or ”Branch History Table”)- small, fully-associative cache to store information about the most recently executed branch instructions. With a fully-associative cache, you supply a tag/key value to search for across the whole cache in parallel. In a BTB, the Branch instruction address acts as the tag since that’s what you know about an instruction at the beginning of the IF stage.

| Valid Bit | Branch Instr. Addr. | Target Address | Prediction Bits |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

Tag Field

During the IF stage, the Branch Target Buffer is checked to see if the instruction being fetched is a branch (if the addresses match) instruction.

If the instruction is a branch instruction and it is in the Branch Target Buffer, then the target address and prediction can be supplied by the BTB **by the end of the IF for the branch instruction**.

If the branch instruction is in the Branch Target Buffer, will the target address supplied correspond to the correct instruction to be execute next?

What if the instruction is a branch instruction and it is not in the Branch Target Buffer?

Should the Branch Target Buffer contain entries for unconditional as well as conditional branch instructions?

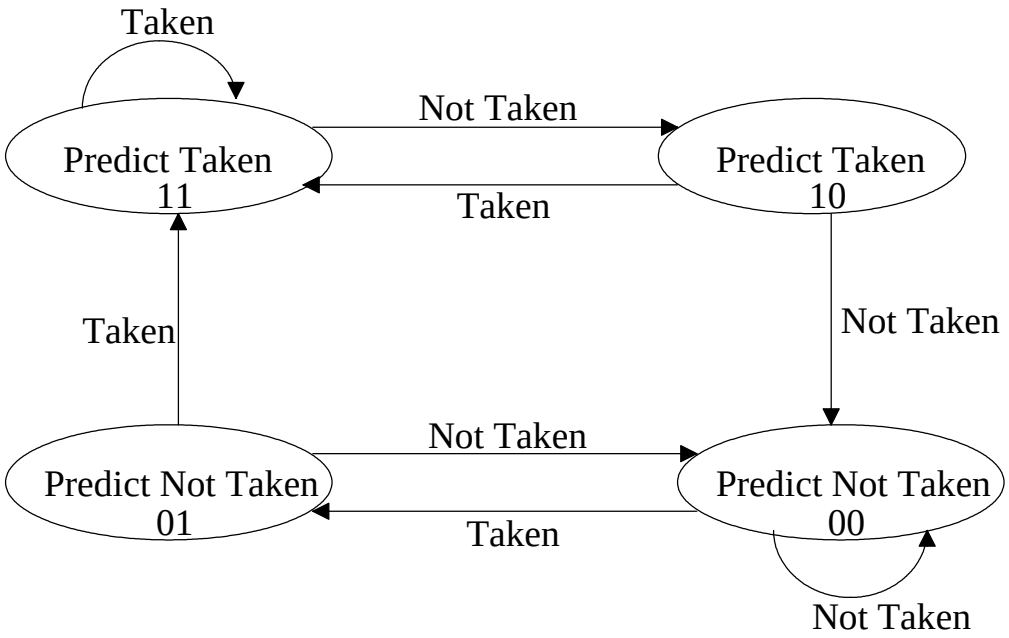
The table shows the advantage of using a Branch Target Buffer to improve accuracy of the branch prediction. It shows the impact of past ***n*** branches on prediction accuracy.

| | Type of mix | | |
|---|---|---|---|
| n | Compiler | Business | Scientific |
| 0 | 64.1 | 64.4 | 70.4 |
| 1 | 91.9 | 95.2 | 86.6 |
| 2 | 93.3 | 96.5 | 90.8 |
| 3 | 93.7 | 96.6 | 91.0 |
| 4 | 94.5 | 96.8 | 91.8 |
| 5 | 94.7 | 97.0 | 92.0 |

Notice:

1) the big jump in using the knowledge of just 1 past branch to predict the branch
2) notice the big jump in going from using 1 to 2 past branches to predict the branch for scientific applications. What types of data do scientific applications spend most of their time processing? What would be true about the code for processing this type of data?

Typically, two prediction bits are use so that two wrong predictions in a row are need to change the prediction -- see page 570.




How does this help for nested loops?

Consider the nested loops:

```
for (i = 1; i <= 100; i++) {
    for (j = 1; j <= 100; j++) {
       <do something>
    }
}
```

## Nested Loop Unconditional Branch Penalty With 1 History Bit

| | | PREDICTION | ACTUAL | NEXT PREDICTION | PENALTY |
|---|---|---|---|---|---|
| i = 1 | | miss in BHT | NOT TAKEN | NOT TAKEN | |
| | j = 1 | miss in BHT | NOT TAKEN | NOT TAKEN | |
| | j = 2 | NOT TAKEN | NOT TAKEN | NOT TAKEN | |
| | • | | | | |
| | • | | | | |
| | • | | | | |
| | j = 100 | NOT TAKEN | NOT TAKEN | NOT TAKEN | |
| | j = 101 | NOT TAKEN | TAKEN | TAKEN | 1 |
| i = 2 | | NOT TAKEN | NOT TAKEN | NOT TAKEN | |
| | j = 1 | TAKEN | NOT TAKEN | NOT TAKEN | 1 |
| | j = 2 | NOT TAKEN | NOT TAKEN | NOT TAKEN | |
| | • | | | | |
| | • | | | | |
| | • | | | | |
| | j = 100 | NOT TAKEN | NOT TAKEN | NOT TAKEN | |
| | j = 101 | NOT TAKEN | TAKEN | TAKEN | 1 |
| | • | | | | |
| | • | | | | |
| | • | | | | |

Penalties associated with conditional branches = $\underbrace{1 + 2 * 99}_{\text{inner loop}} + \underbrace{1}_{\text{outer loop}}$

Consider the nested loops:

```
for (i = 1; i <= 100; i++) {
    for (j = 1; j <= 100; j++) {
       <do something>
    }
}
```

## Nested Loop Unconditional Branch Penalty With 2 History Bit

| | | PREDICTION | ACTUAL | NEXT PREDICTION | PENALTY |
|---|---|---|---|---|---|
| i = 1 | | miss in BHT | NOT TAKEN | NOT TAKEN | |
| | j = 1 | miss in BHT | NOT TAKEN | NOT TAKEN | |
| | j = 2 | NOT TAKEN | NOT TAKEN | NOT TAKEN | |
| | • | | | | |
| | • | | | | |
| | • | | | | |
| | j = 100 | NOT TAKEN | NOT TAKEN | NOT TAKEN | |
| | j = 101 | NOT TAKEN | TAKEN | NOT TAKEN | 1 |
| i = 2 | | NOT TAKEN | NOT TAKEN | NOT TAKEN | |
| | j = 1 | NOT TAKEN | NOT TAKEN | NOT TAKEN | |
| | j = 2 | NOT TAKEN | NOT TAKEN | NOT TAKEN | |
| | • | | | | |
| | • | | | | |
| | • | | | | |
| | j = 100 | NOT TAKEN | NOT TAKEN | NOT TAKEN | |
| | j = 101 | NOT TAKEN | TAKEN | NOT TAKEN | 1 |
| | • | | | | |
| | • | | | | |
| | • | | | | |

Penalties associated with conditional branches = 1 * 100 + 1

inner loop

outer loop

**Correlating Predictor** - instead of just using local information about a branch (its past branch behavior) to make a prediction, more global information about the behavior of some number of recently executed branches can be used to improve the branch prediction.

**Tournament Predictor** - multiple predictions for each branch are recorded with a selection mechanism that chooses which predictor to use. A typical tournament predictor might contain two predictions for each branch:

1. one based on local information
2. one based on global branch behavior

By tracking performance of both predictors, the selector could choose the predictor which has been the most accurate.