Course on: “Advanced Computer Architectures”

Branch Prediction Techniques

Prof. Cristina Silvano
Politecnico di Milano
email: cristina.silvano@polimi.it

Outline

- The Problem of Control Hazards in the Processor Pipeline
- Branch Prediction Techniques
  - Static Branch Prediction
  - Dynamic Branch Prediction

The Problem of Control Hazards

Conditional Branch Instructions

- A branch is **taken** if the condition is satisfied: the **branch target address** is stored in the Program Counter (PC) instead of the address of the next instruction in the sequential instruction stream (PC + 4).

- Examples of conditional branches for MIPS processor: **beq** (*branch on equal*) and **bne** (*branch on not equal*)
  - `beq $s1, $s2, L1   # go to L1 if ($s1 == $s2)`
  - `bne $s1, $s2, L1   # go to L1 if ($s1 != $s2)`

Execution of conditional branches for 5-stage MIPS pipeline

`beq $x,$y,L1`

Instr. Fetch & PC Increm.	Register Read `$x` e `$y`	ALU Op. **`($x-$y)`** **`& (PC+4+offset)`**	Write of PC	

- Instruction fetch and PC increment
- Registers read **`($x`** and **`$y)`** from Register File.
- ALU operation to compare registers ($x and $y) to derive **Branch Outcome** (branch taken or branch not taken).
- Computation of **Branch Target Address** **(PC+4+offset):** the value (PC+4) is added to the least significant 16 bit of the instruction after sign extension
- The result of registers comparison from ALU (Branch Outcome) is used to decide the value to be stored in the PC: **(PC+4)** or **(PC+4+offset).**

Execution of conditional branches for 5-stage MIPS pipeline

IF Instruction Fetch	ID Instruction Decode	EX Execution	ME Memory Access	WB Write Back

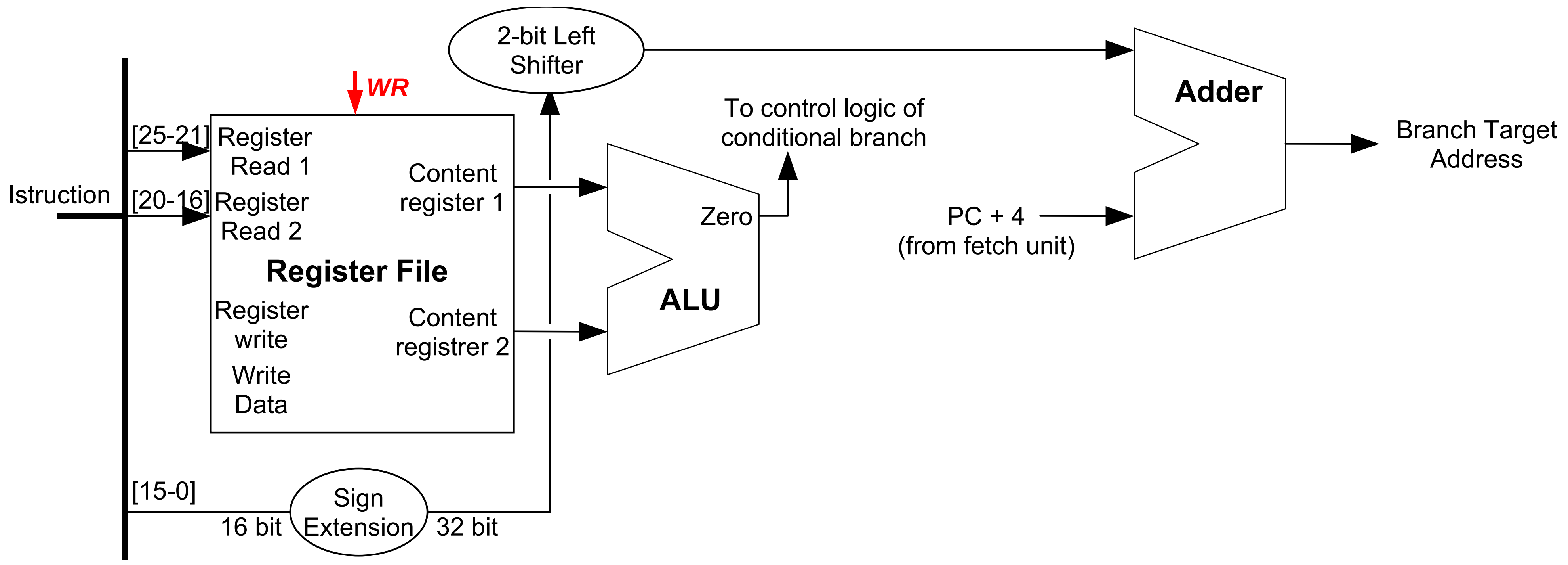
`beq $x,$y,L1`

Instr. Fetch & PC Increm.	Register Read **`$x`** e **`$y`**	ALU Op. **`($x-$y)`** **`& (PC+4+offset)`**	Write of PC	

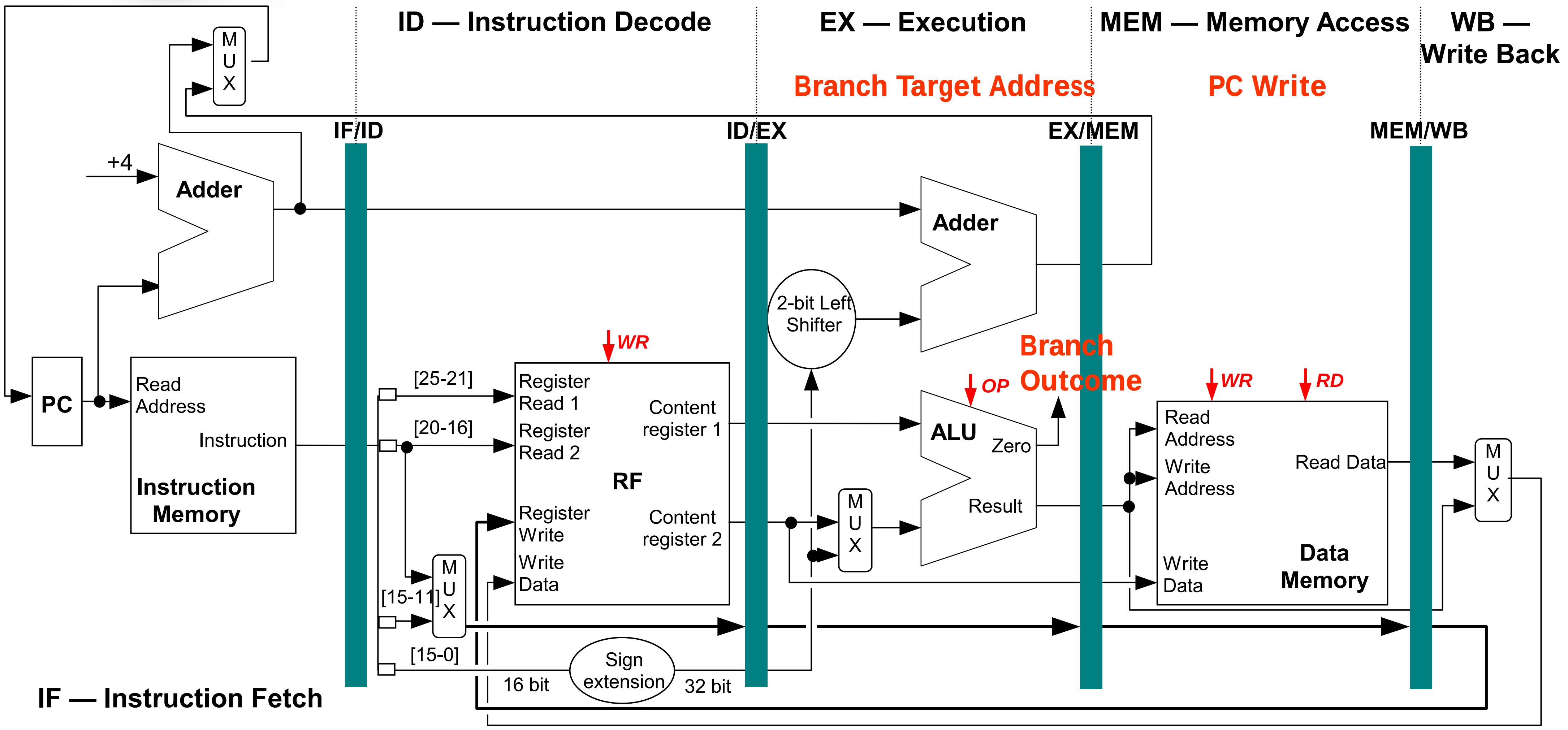
- **Branch Outcome** and **Branch Target Address** are ready at the end of the EX stage (3th stage)
- Conditional branches are solved when **PC** is updated at the end of the ME stage (4th stage)

Execution of conditional branches for MIPS

- Processor resources to execute conditional branches:



Implementation of the 5-stage MIPS Pipeline



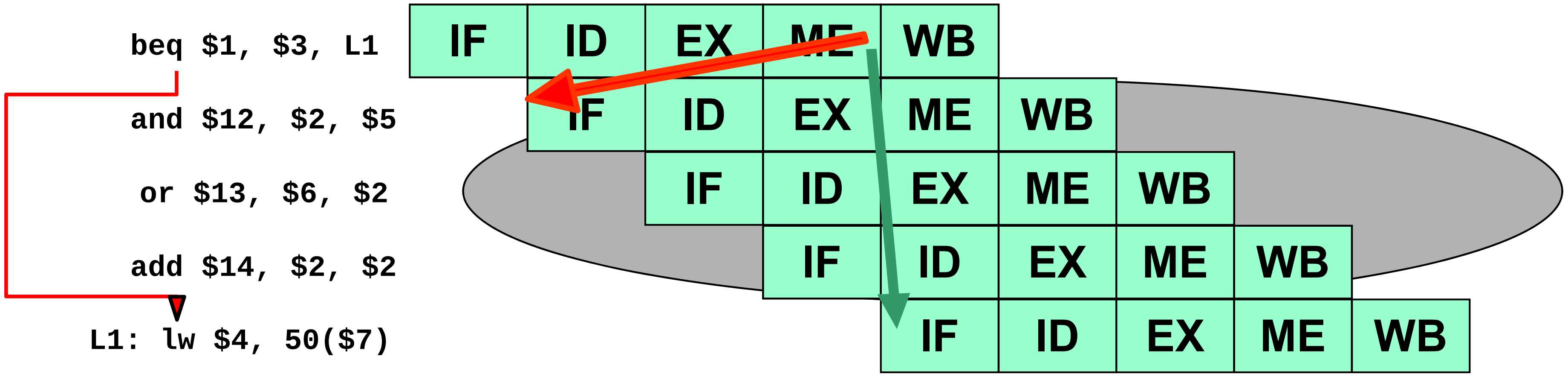
The Problem of Control Hazards

- **Control hazards:** Attempt to make a decision on the next instruction to fetch before the branch condition is evaluated.
- Control hazards arise from the pipelining of conditional branches and other instructions changing the PC.
- Control hazards reduce the performance from the ideal speedup gained by the pipelining since they can make it necessary to **stall** the pipeline.

Branch Hazards

- To feed the pipeline we need to fetch a new instruction at each clock cycle, but the branch decision (to change or not change the PC) is taken during the MEM stage.
- This delay to determine the correct instruction to fetch is called **Control Hazard or Conditional Branch Hazard**
- If a branch changes the PC to its target address, it is a **taken branch**
- If a branch falls through, it is **not taken** **or** **untaken.**

Branch Hazards: Example



- The branch instruction may or may not change the PC in MEM stage, but the next 3 instructions are fetched and their execution is started.
- If the branch is **not taken,** the pipeline execution is fine
- If the branch is **taken,** it is necessary to *flush* the next 3 instructions in the pipeline before they are writing their results, then we need to fetch the `lw` instruction at the branch target address **(`L1`)**

Branch Hazards: Solutions

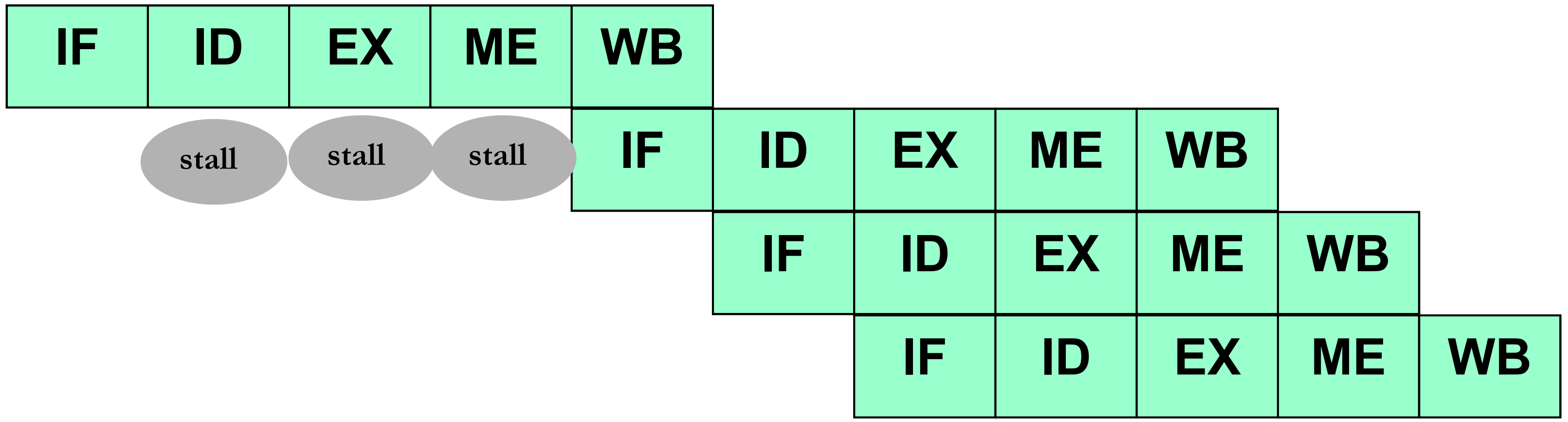
- If the branch is *not taken*, introducing three cycles penalty is not justified $\Rightarrow$ throughput reduction.
- **Solution:** We can assume the *branch not taken*, and **flush** the next 3 instructions in the pipeline only if the branch will be taken. (We cannot assume the *branch taken* because we don't know the branch target address)
- This solution introduces the idea of *branch prediction*
- *But, let's assume to be conservative…*

Branch Hazards: Conservative assumption

- **Conservative assumption:** To stall the pipeline until the branch decision is taken **(stalling until resolution),** then fetch the correct instruction flow.
  - Without forwarding : We need to stall for **3** clock cycles
  - With forwarding: We need to stall for **2** clock cycles

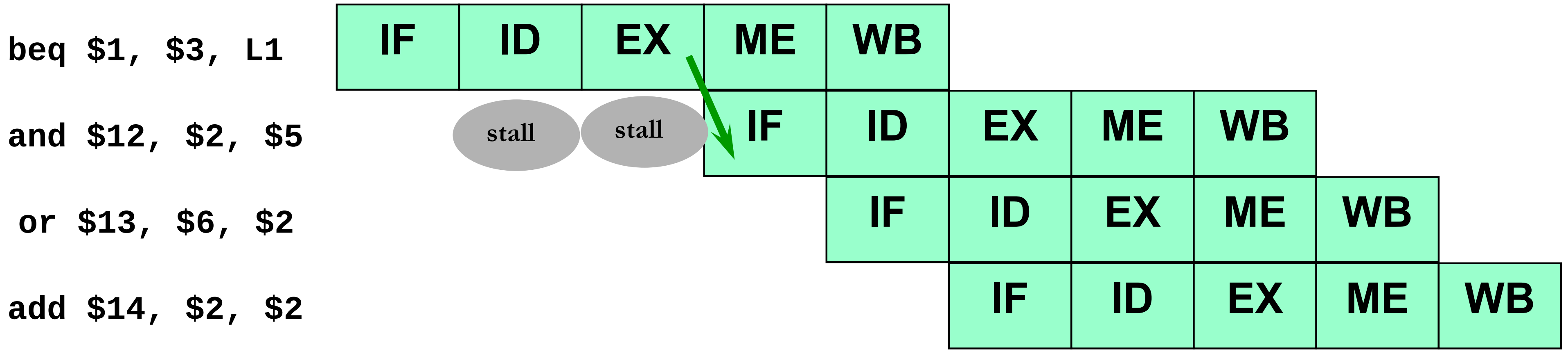
Branch Stalls without Forwarding

```
beq $1, $3, L1
and $12, $2, $5
 or $13, $6, $2
add $14, $2, $2
```



- **Conservative assumption:** Stalling until resolution at the end of the ME stage.
- Each branch costs **three stalls** to fetch the correct instruction flow: (PC+4) or Branch Target Address

Branch Stalls with Forwarding



- **Conservative assumption:** Stalling until resolution at the end of the EX stage (when the BO and BTA are known)
- Each branch costs **two stalls** to fetch the correct instruction flow: (PC+4) or Branch Target Address

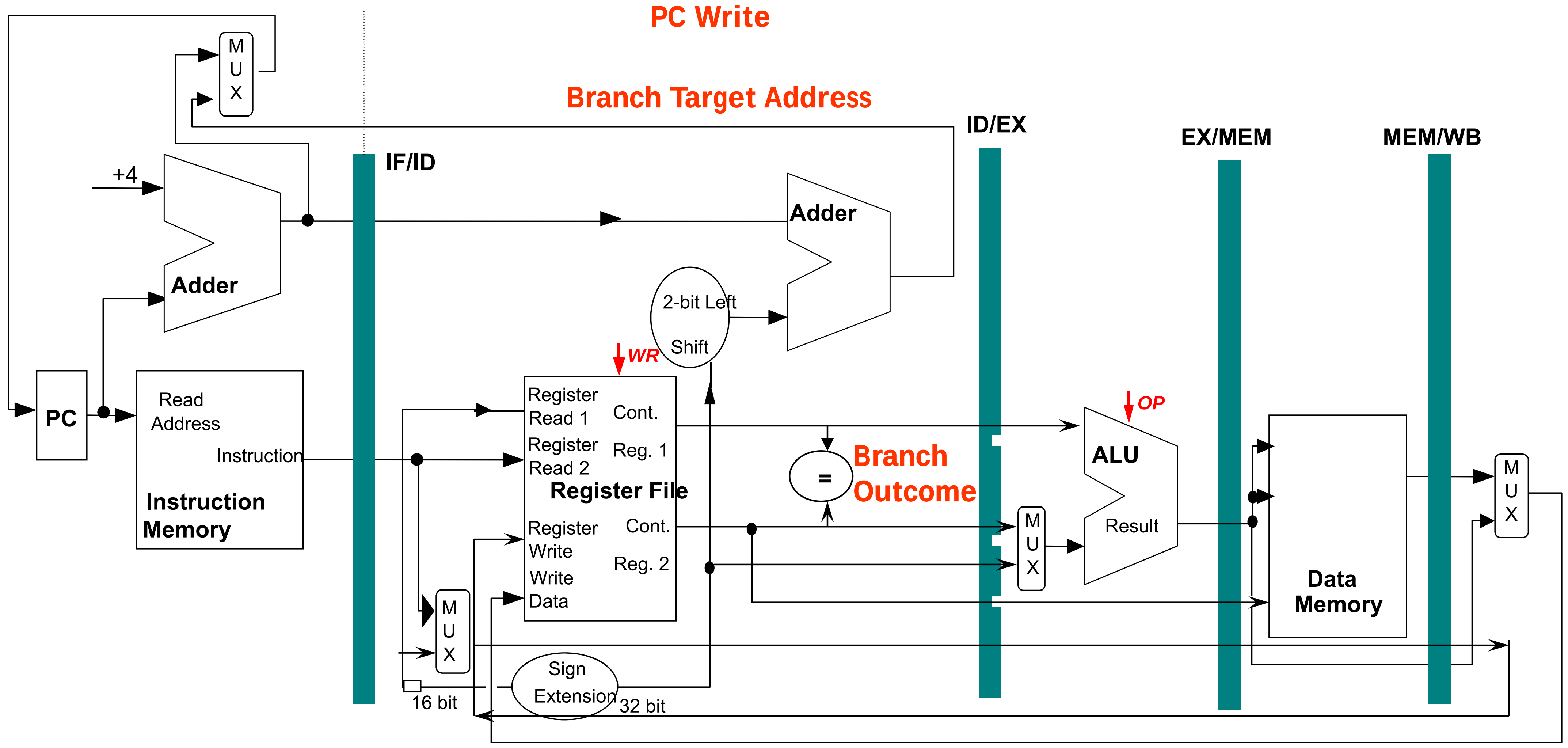
Early Evaluation of the PC

- To improve performance in case of branch hazards, we need to add more hardware resources to:
  1. Compare registers to derive the **Branch Outcome**
  2. Compute the **Branch Target Address**
  3. Update the PC register

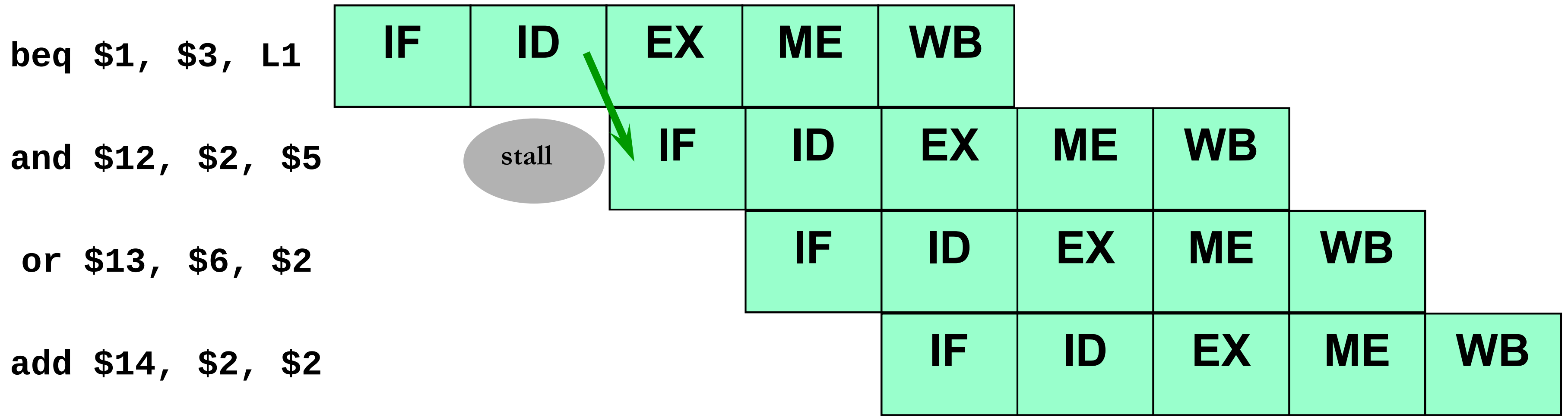
  as soon as possible in the pipeline.

- MIPS processor anticipated the comparison of registers, computation of BTA and update of PC *during ID stage.*

MIPS Processor: Early Evaluation of the PC



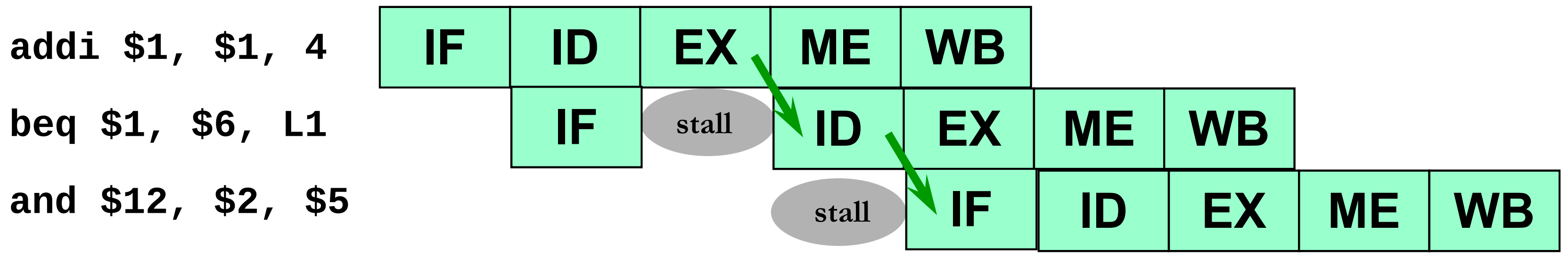
MIPS Processor: Early Evaluation of the PC



- Conservative assumption: Stalling until resolution at the end of the ID stage (when the BO and BTA are known)
- Each branch costs **one stall** to fetch the correct instruction flow: (PC+4) or Branch Target Address

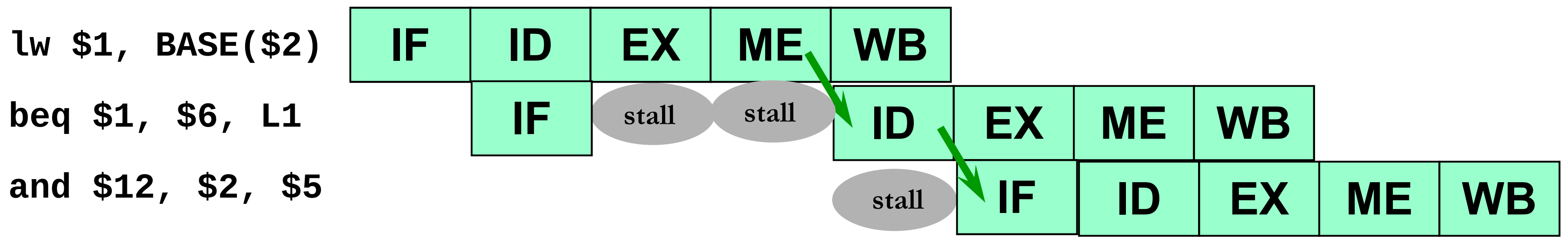
MIPS Processor: Early Evaluation of the PC

- **Consequence** of early evaluation of the branch decision in ID stage:
  - In case of **add** instruction followed by a **branch** testing the result ⇒ we need to introduce **one stall** **before ID stage of branch** to enable the forwarding (EX-ID) of the result from EX stage of previous instruction. As usual we need one stall **after** the branch for branch resolution.



MIPS Processor: Early Evaluation of the PC

- **Consequence** of early evaluation of the branch decision in ID stage:
  - In case of **`load`** instruction followed by a **`branch`** testing the result ⇒ we need to introduce **two stalls before ID stage of branch** to enable the forwarding (ME-ID) of the result from EX stage of previous instruction. As usual we need one stall **after** the branch for branch resolution.



MIPS Processor: Early Evaluation of the PC

- With the branch decision made during ID stage, there is a reduction of the cost associated with each branch **(branch penalty):**
  - We need only **one-clock-cycle stall after** each branch
  - Or a **flush** of only **one** instruction following the branch

- One-cycle-delay for every branch still yields a performance loss of 10% to 30% depending on the branch frequency

- Pipeline Stall Cycles per Instruction due to Branches =
  Branch Frequency x Branch Penalty

- We will examine some branch prediction techniques to deal with this performance loss.

Branch Prediction Techniques

Branch Prediction Techniques

- **Main goal:** try to predict as early as possible the outcome of a branch instruction.

- The **performance** of a branch prediction technique depends on:
  - **Accuracy** measured in terms of **percentage of incorrect predictions** given by the predictor.
  - **Cost** of a 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
  - **Branch frequency** given by the application: the importance of accurate branch prediction is higher in programs with higher branch frequency.

Branch Prediction Techniques

- There are **two types** of methods to deal with the performance loss due to branch hazards:
  - **Static Branch Prediction Techniques:** The actions (taken/untaken) for a branch prediction are **fixed at compile time** for each branch during the entire execution.
  - **Dynamic Branch Prediction Techniques:** The actions (taken/untaken) for a branch prediction **can change at runtime** during the program execution.
- In both cases, we need to do not change the processor state and registers until the Branch Outcome is definitely known.

Static Branch Prediction Techniques

Static Branch Prediction Techniques

- **Static Branch Prediction** is used when the expectation is that the branch behavior of the target application is highly predictable at compile time.
- **Static Branch Prediction** can also be used to assist dynamic predictors.

Static Branch Prediction Techniques

1) Branch Always Not Taken (Predicted-Not-Taken)

2) Branch Always Taken (Predicted-Taken)

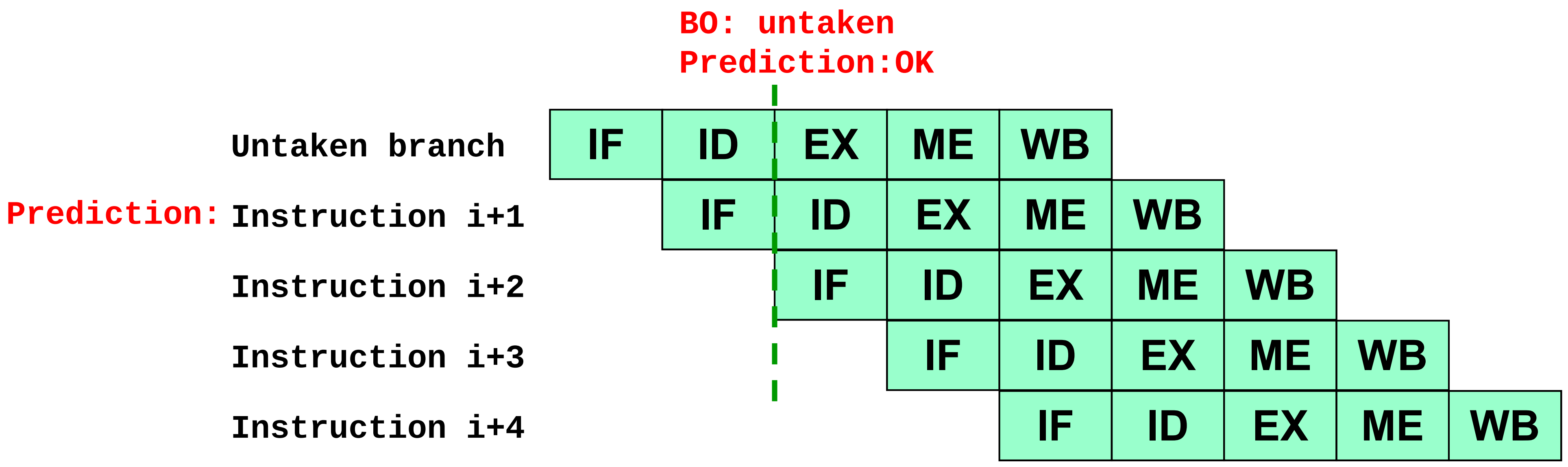
3) Backward Taken Forward Not Taken (BTFNT)

4) Profile-Driven Prediction

5) Delayed Branch

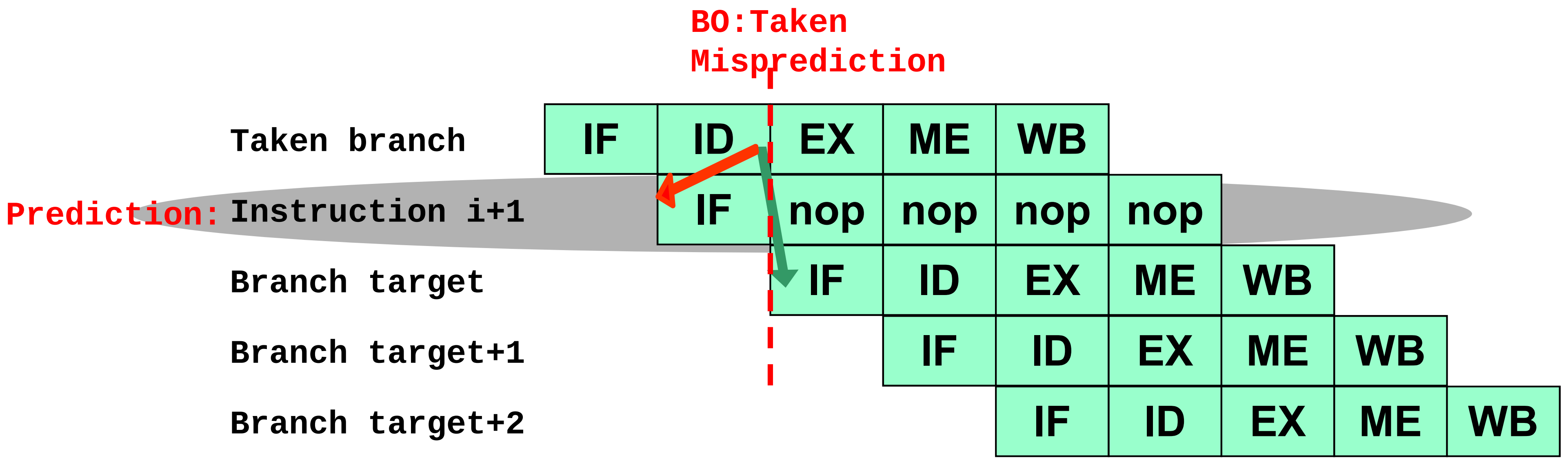
1) Branch Always Not Taken

- We assume the **branch will not be taken,** thus the sequential instruction flow we have fetched can continue as if the branch condition was not satisfied.
- If the BO at the end of ID stage will result **not taken (the prediction is correct),** we can preserve performance.



1) Branch Always Not Taken

- If the BO at the end of ID stage will result **taken (the prediction is incorrect):**
  - We need to **flush** the next instruction already fetched (the next instruction is turned into a **nop**) and we restart the execution by fetching the instruction at the Branch Target Address
    $\Rightarrow$ **One-cycle performance penalty**



Taken branch	IF	ID	EX	ME	WB				
Instruction i+1		IF	nop	nop	nop	nop			
Branch target			IF	ID	EX	ME	WB		
Branch target+1				IF	ID	EX	ME	WB	
Branch target+2					IF	ID	EX	ME	WB

2) Branch Always Taken

- An alternative scheme is to consider every branch as **taken:** as soon as the branch is decoded and the **Branch Target Address** is computed, we assume the branch to be taken and we begin fetching and executing at the target address.
- The predicted-taken scheme makes sense for pipelines where the branch target address is known **before** the branch outcome.
- In MIPS pipeline, we don't know the branch target address earlier than the branch outcome, so there is no advantage in the application of this technique.
  - We should anticipate the computation of BTA at the IF stage (before the ID stage) or we need a **Branch Target Buffer,** a cache to store the predicted value of the BTA for the next instruction after each branch.

3) Backward Taken Forward Not Taken (BTFNT)

- The prediction is based on the branch direction:
  - Backward-going branches are predicted as *taken*
    - Example: the branches at the end of loops go back at the beginning of the next loop iteration
      $\Rightarrow$ we assume the backward-going branches are always taken.
  - Forward-going branches are predicted as *not taken*
    - Example: the branches going forward to an ELSE label of an IF-ELSE clause $\Rightarrow$ if we assume the conditions associated to the ELSE as less probable, it is better to consider the forwarding branches always not taken

4) Profile-Driven Prediction

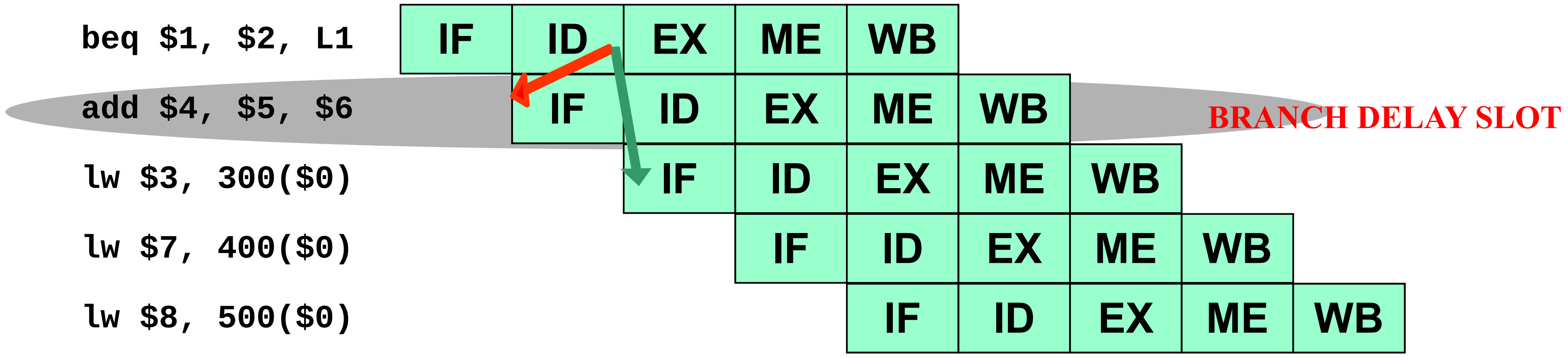
- Let us assume we can profile the behavior of a target application program by executing several runs with different data sets
- The branch prediction is based on profiling information collected from earlier runs.
- The profile-driven prediction method can use compiler hints associated to each branch.

5) Delayed Branch Technique

- Scheduling technique: 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) $\Rightarrow$ we have only **one-delay slot** to fill in
- It is possible to have for some deeply pipeline processors a branch delay longer than one-cycle

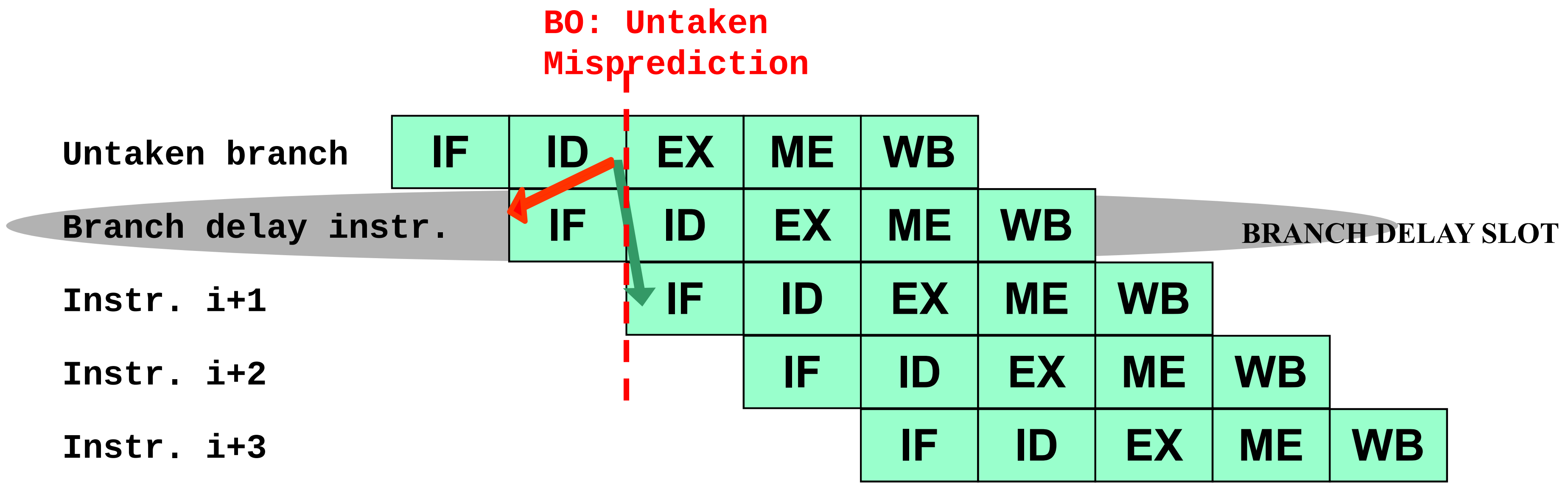
5) Delayed Branch Technique

- The MIPS compiler always schedules a branch independent instruction after the branch.
- Example: A previous **add** instruction with no effects on the branch is scheduled in the **Branch Delay Slot**



5) Delayed Branch Technique

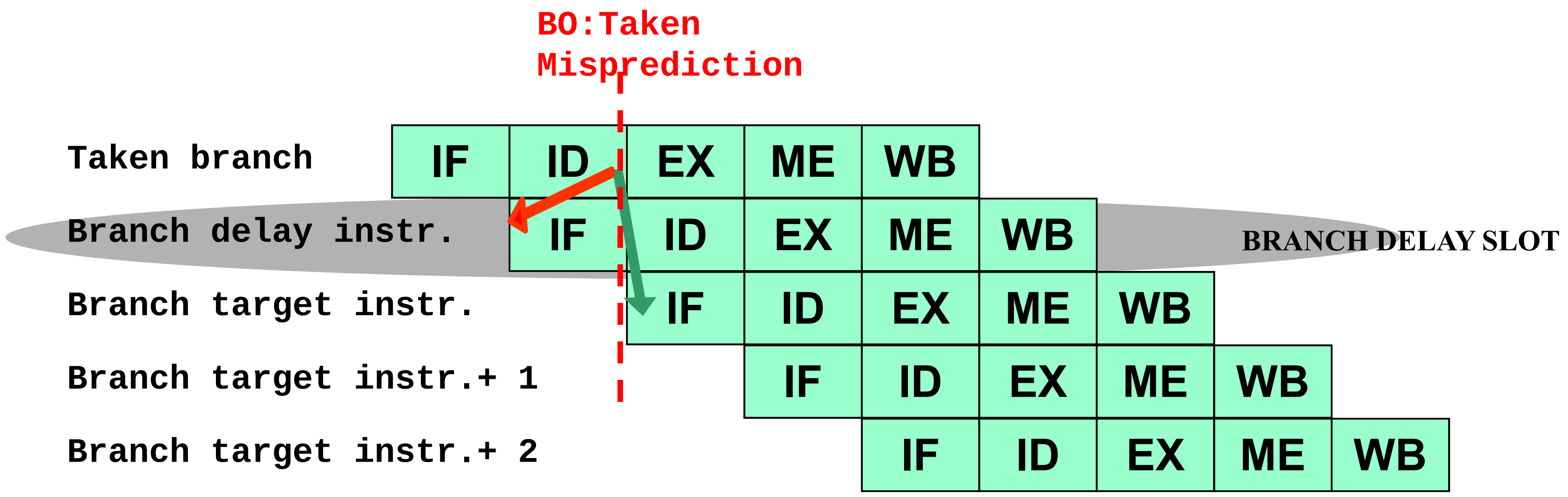
- The behavior of the delayed branch is the same whether or not the branch is taken.
  - If the branch is **untaken** $\Rightarrow$ execution continues with the instruction after the branch



Untaken branch	IF	ID	EX	ME	WB					
Branch delay instr.		IF	ID	EX	ME	WB				
Instr. i+1			IF	ID	EX	ME	WB			
Instr. i+2				IF	ID	EX	ME	WB		
Instr. i+3					IF	ID	EX	ME	WB	

5) Delayed Branch Technique

- If the branch is **taken** ⇒ execution continues at the branch target



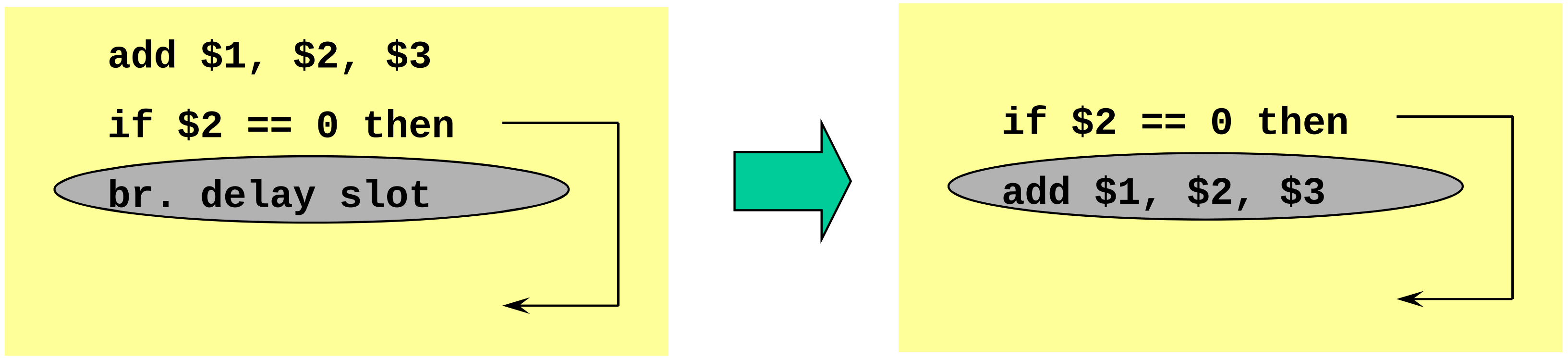
5) Delayed Branch Technique

- The job of the compiler is to make the instruction placed in the branch delay slot valid and useful.
- There are four ways in which the branch delay slot can be scheduled:

1. **From before**
2. **From target**
3. **From fall-through**
4. **From after**

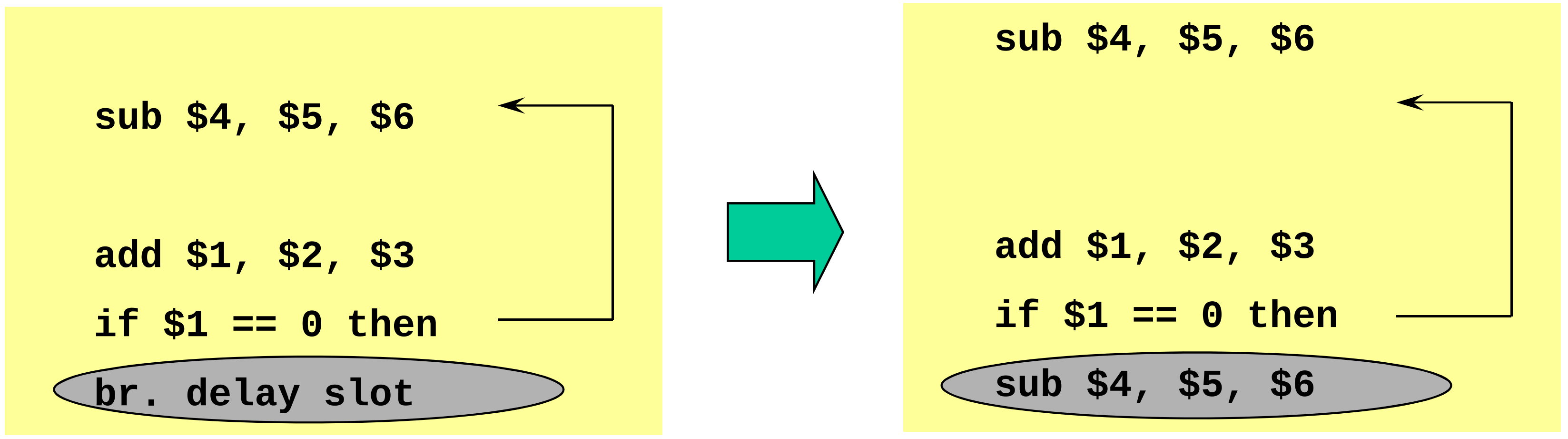
5) Delayed Branch Technique: From Before

- The branch delay slot is scheduled with an independent instruction from before the branch
- The instruction in the branch delay slot is **always executed** (whether the branch is taken or untaken).
- Then execution will continue based on the Branch Outcome in the right direction and the **add** instruction in the delay slot will never be flushed.



5) Delayed Branch Technique: From Target

- The use of **$1** in the branch condition prevents **add** instruction (whose destination is **$1**) from being moved after the branch.
- The branch delay slot is scheduled from the target of the branch (usually the target instruction **sub** needs to be copied because it can be reached by another path).
- This strategy is preferred when the branch is **taken** with high probability, such as loop branches **(backward branches).**
- If the branch is **untaken (misprediction),** the **sub** instruction in the delay slot needs to be flushed.

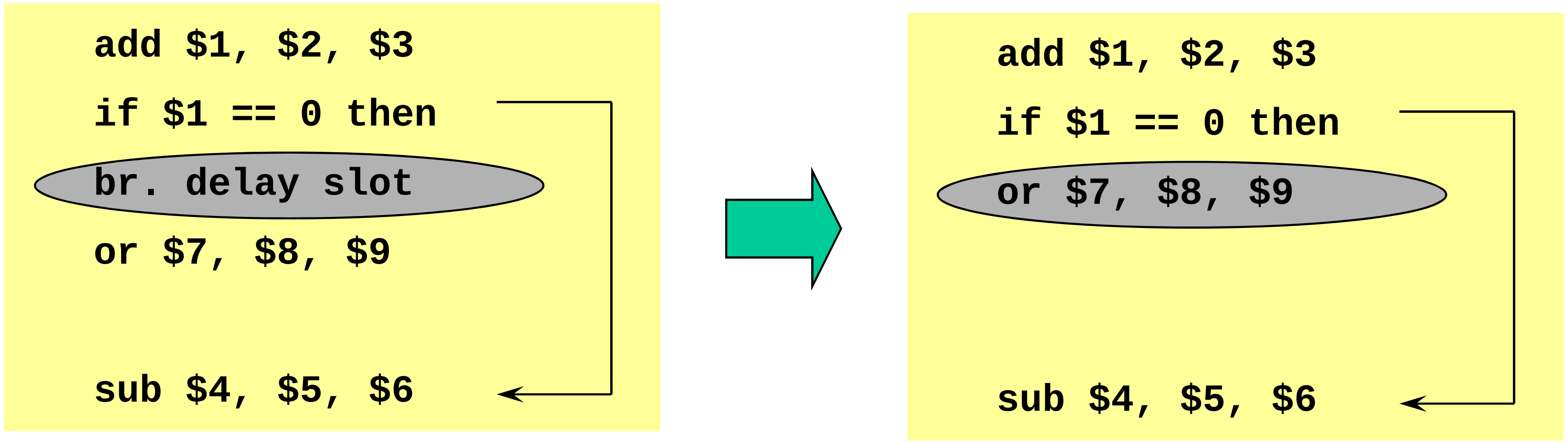


```
sub $4, $5, $6

add $1, $2, $3
if $1 == 0 then
br. delay slot
```

```
sub $4, $5, $6

add $1, $2, $3
if $1 == 0 then
sub $4, $5, $6
```

5) Delayed Branch Technique: From Fall-Through

- The use of **$1** in the branch condition prevents add instruction (whose destination is **$1**) from being moved after the branch.
- The branch delay slot is scheduled from the **not taken** fall-through path.
- This strategy is preferred when the branch is **not taken** with high probability, such as **forward branches.**
- If the branch is **taken (misprediction),** the **or** instruction in the delay slot needs to be flushed.



```
add $1, $2, $3
if $1 == 0 then
br. delay slot
or $7, $8, $9

sub $4, $5, $6
```

```
add $1, $2, $3
if $1 == 0 then
or $7, $8, $9


sub $4, $5, $6
```

5) Delayed Branch Technique

- To make the optimization legal for the target and fall-through cases, it must flushed or it must be OK to execute the moved instruction when the branch goes in the unexpected direction.
- By OK we mean that the instruction in the branch delay slot is executed but the work is wasted (the program will still execute correctly).
- For example, if the destination register is an unused temporary register when the branch goes in the unexpected direction.

5) Delayed Branch Technique

- In general, compilers are able to fill in about the 50% of delayed branch slots with valid and useful instructions, the remaining slots are filled with **nops**.
- In deeply pipelined processors, the delayed branch is longer that one cycle: many slots must be filled for every branch.
  - Since it is more difficult for the compiler to fill in all the slots with useful instructions $\Rightarrow$ almost all processors with delayed branch technique have a single delay slot

5) Delayed Branch Technique

- The main limitations on delayed branch scheduling arise from:
  - The restrictions on the instructions that can be scheduled in the delay slot.
  - The ability of the compiler to statically predict the outcome of the branch.

5) Delayed Branch Technique

- To improve the ability of the compiler to fill the branch delay slot ⇒ most processors have introduced a **canceling or nullifying branch:** the instruction includes the direction that the branch was predicted.
  - When the branch behaves as predicted ⇒ the instruction in the branch delay slot is executed normally.
  - When the branch is incorrectly predicted ⇒ the instruction in the branch delay slot is turned to a **nop** (flushed)
- In this way, the compiler need not be as conservative when filling the delay slot.

5) Delayed Branch Technique

- MIPS architecture has the **branch-likely** instruction, that behaves as cancel-if-not-taken branch:
  - The instruction in the branch delay slot is executed whether the branch is taken.
  - The instruction in the branch delay slot is **not executed (it is turned to a nop)** whether the branch is untaken.
- Useful approach for backward branches (such as loop branches).

Dynamic Branch Prediction Techniques

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 Schemes

- Dynamic branch prediction is based on two interacting hardware blocks:

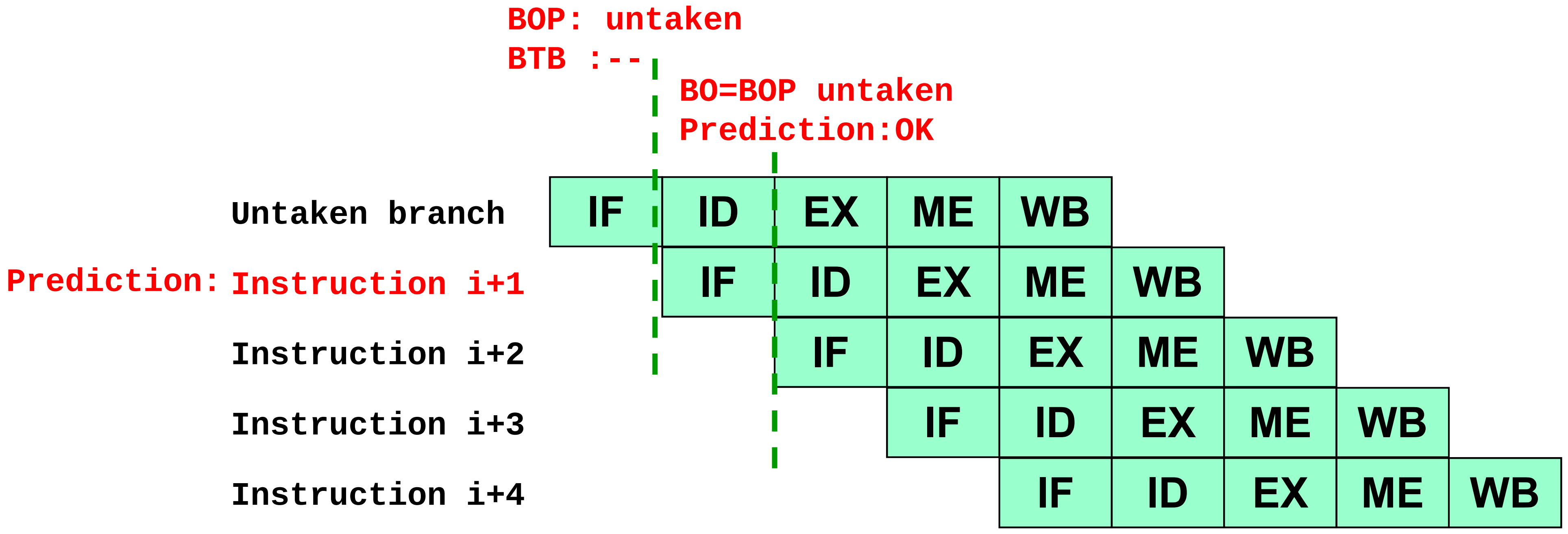
- **Branch Outcome Predictor (BOP):**
  - To predict the direction of a branch (i.e. taken or not taken).
- **Branch Target Predictor or Branch Target Buffer (BTB):**
  - To predict the branch target address in case of taken branch.

Dynamic Branch Prediction Schemes

- The **Branch Outcome Predictor (BOP)** and **the Branch Target Buffer (BTB)** are used by the **Instruction Fetch Unit** to predict the next instruction to read in the Instruction Cache:
  - If branch is predicted by BOP in IF stage as not taken $\Rightarrow$ PC is incremented.
  - If branch is predicted by BOP in IF stage as taken $\Rightarrow$ BTB gives the Branch Target Address (BTA)

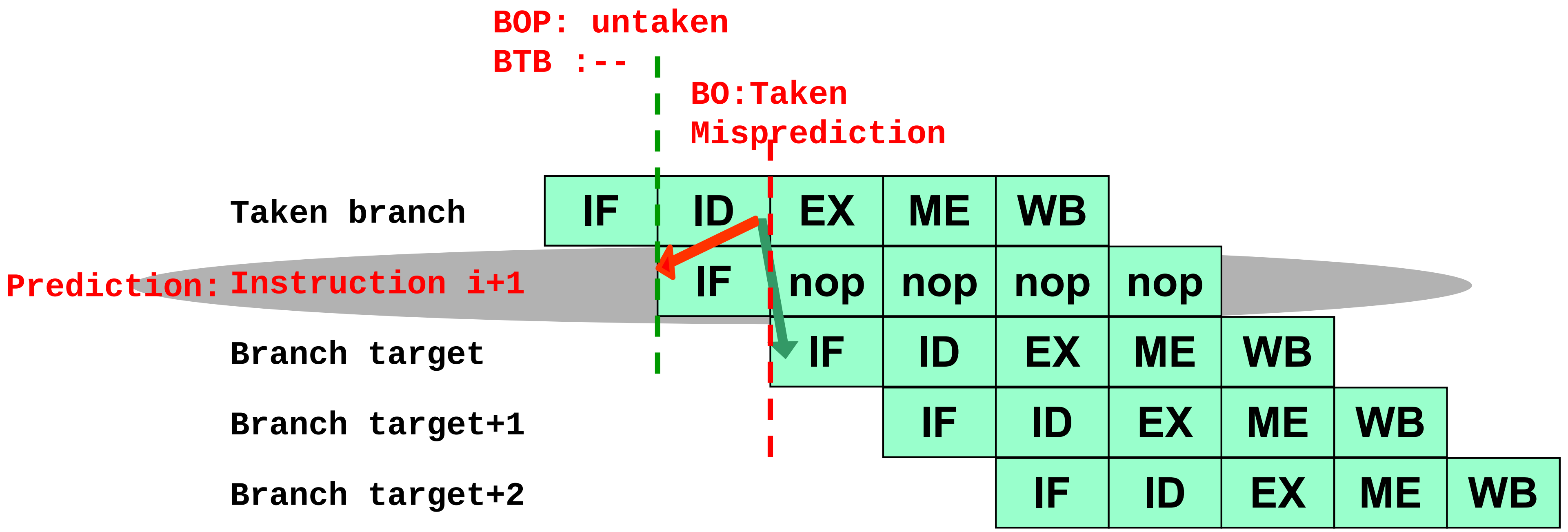
Dynamic Branch Prediction Schemes

- If branch is predicted by BOP in IF stage as **not taken** $\Rightarrow$ PC is incremented.
- If the BO at the end of ID stage will result as **not taken (the prediction is correct),** we can preserve performance.



Dynamic Branch Prediction Schemes

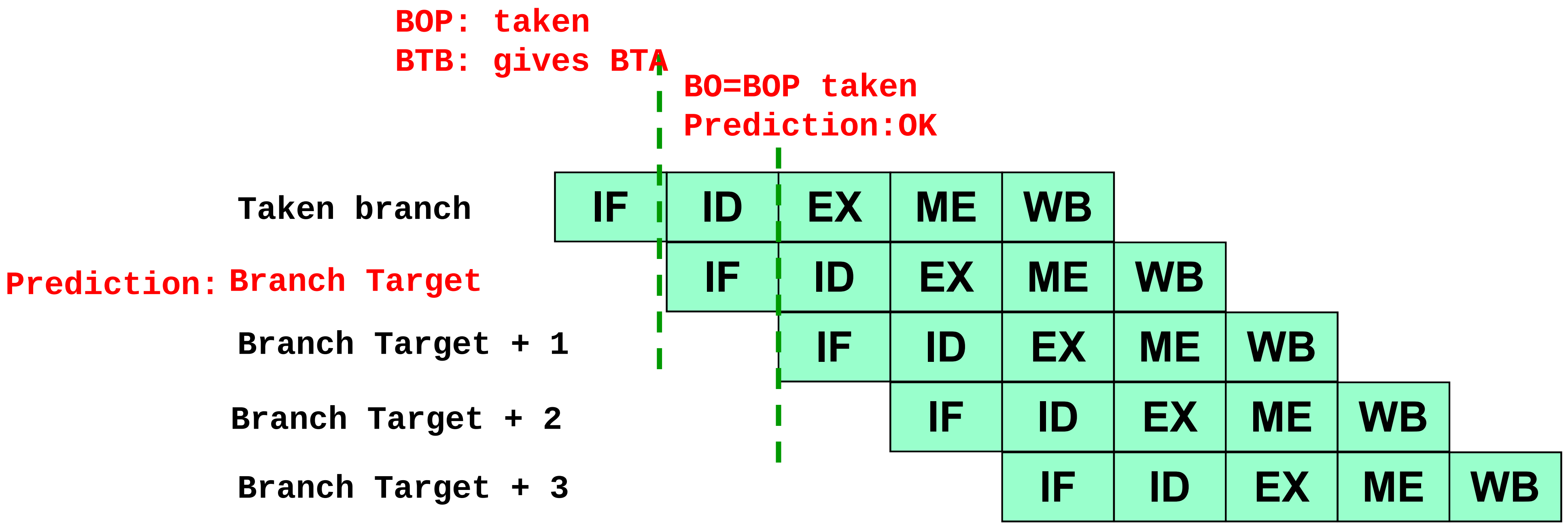
- If the BO at the end of ID stage will result **taken (misprediction):**
  - We need to **flush** the next instruction already fetched (the next instruction is turned into a **nop**) and we restart the execution by fetching at the Branch Target Address ⇒ **One-cycle penalty**



Taken branch	IF	ID	EX	ME	WB				
Instruction i+1		IF	nop	nop	nop	nop			
Branch target			IF	ID	EX	ME	WB		
Branch target+1				IF	ID	EX	ME	WB	
Branch target+2					IF	ID	EX	ME	WB

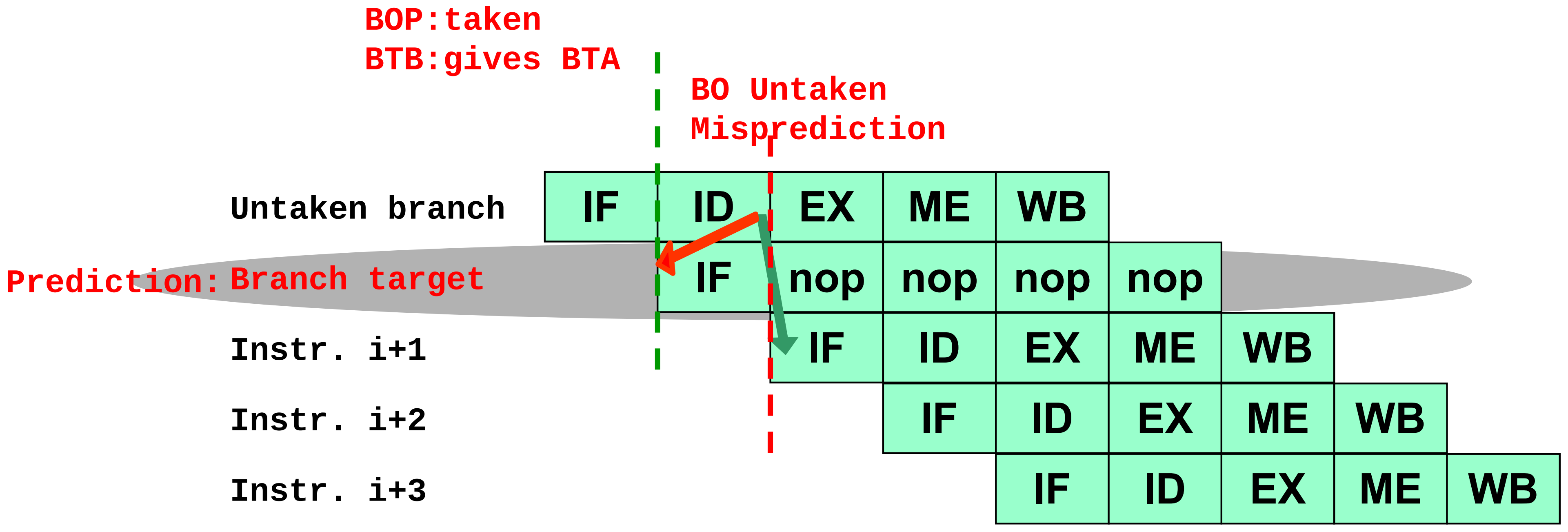
Dynamic Branch Prediction Schemes

- If branch is predicted by BOP in IF stage as **taken**
  $\Rightarrow$ BTB gives the target address
- If the BO at the end of ID stage will result as **taken (the prediction is correct),** we can preserve performance.



Dynamic Branch Prediction Schemes

- If the BO at the end of ID stage will result **untaken (misprediction):**
  - We need to **flush** the next instruction already fetched (the next instruction is turned into a **nop**) and we restart the execution by fetching at the Branch Target Address ⇒ **One-cycle penalty**



Dynamic Branch Prediction Techniques

1) Branch History Table

2) Correlating Branch Predictors

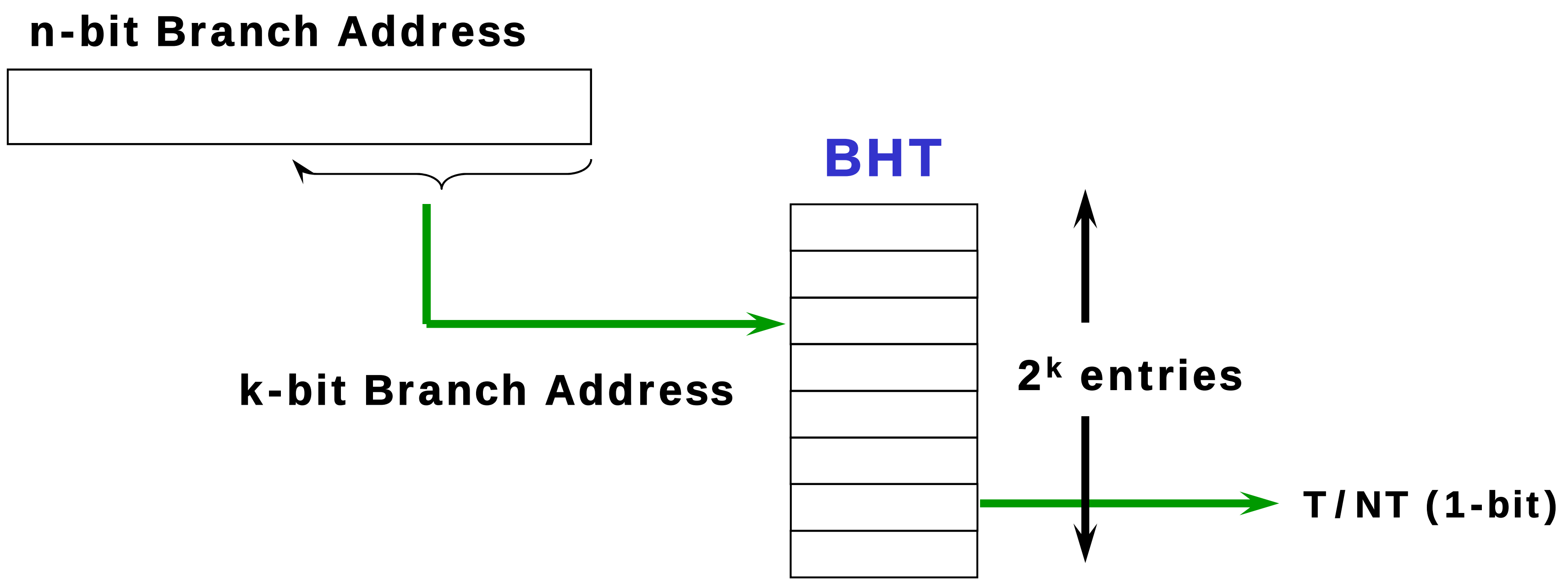
3) Two-level Adaptive Branch Predictors

4) Branch Target Buffer

1) Branch History Table

- **Branch History Table (or Branch Prediction Buffer):**
  - Table containing **1 bit** for each entry that says whether the branch was recently taken or not.
  - Table indexed by the lower portion k-bit of the address of the branch instruction (to keep the size of the table limited)
  - For locality reasons, we would expect that the most significant bits of the branch address are **not** changed

1) Branch History Table



1) Branch History Table

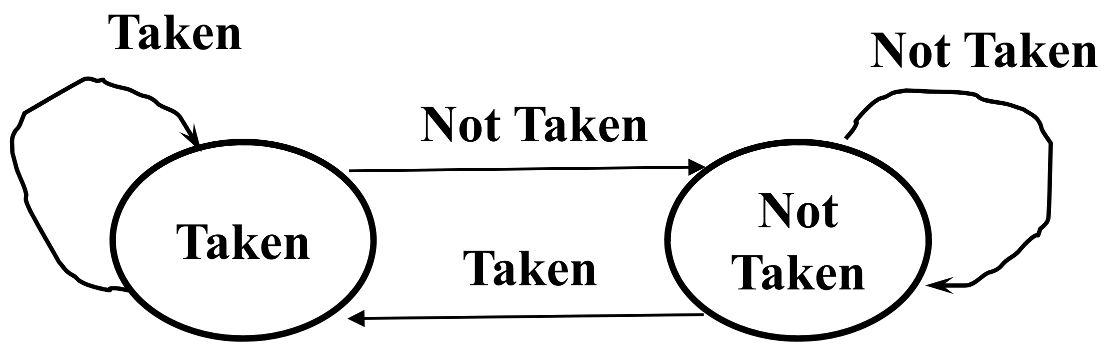
- Prediction: **hint** that it is assumed to be correct, and fetching begins in the predicted direction.
  - If the hint turns out to be wrong, the prediction bit is inverted and stored back. The pipeline is flushed and the correct sequence is executed with one cycle penalty.
- The table has **no tags** (every access is a hit) and the prediction bit could has been put there by another branch with the same low-order address bits: but it doesn't matter. *The prediction is just a hint!*

1) 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 (This can occur because there is no tag check)
    - To reduce this problem it is enough to increase the number of rows in the BHT (that is to increase k) or to use a hashing function (such as in GShare).

1) FSM for 1-bit Branch History Table



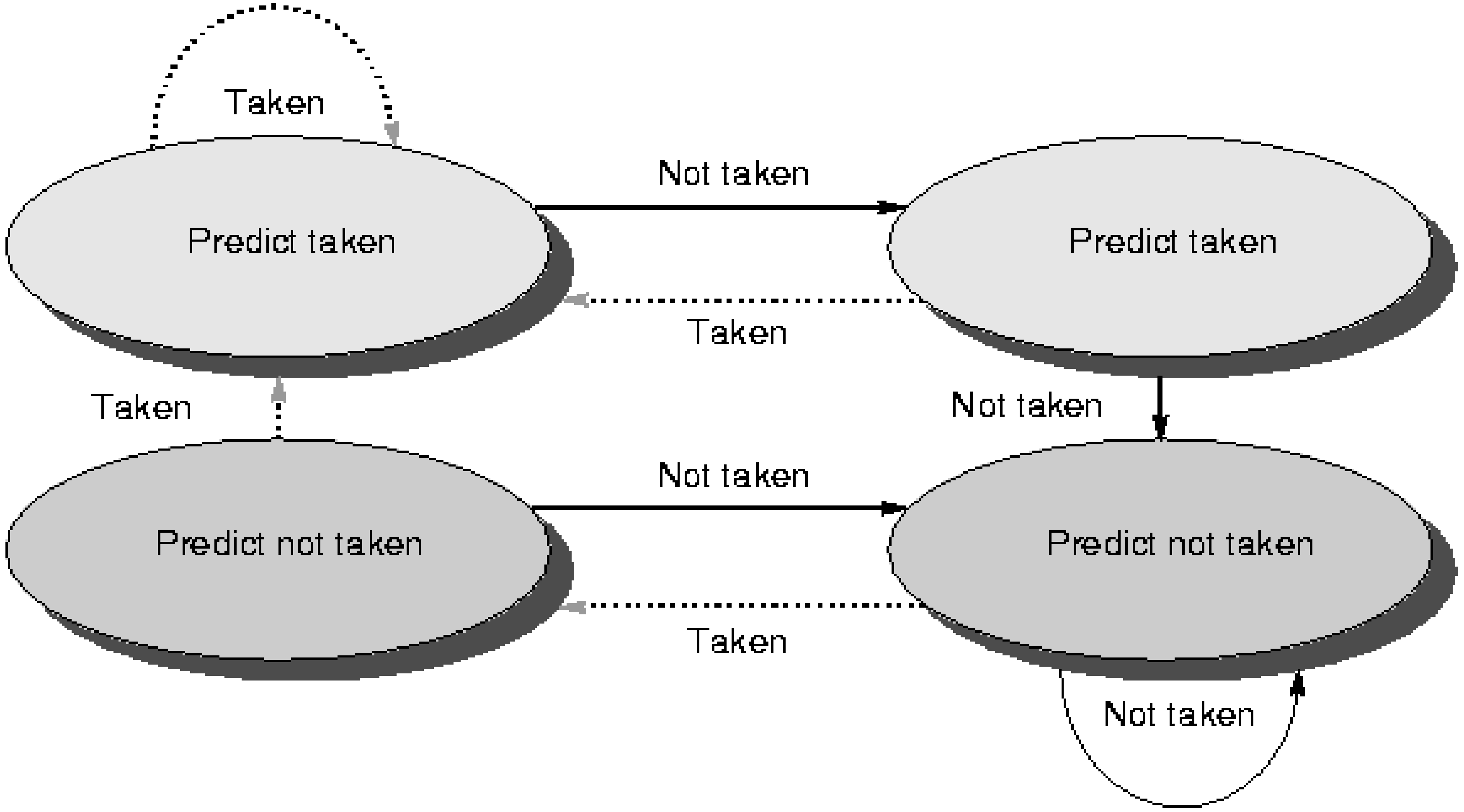
1) 1-bit Branch History Table

- Shortcoming of the 1-bit BHT:
  - In a loop branch, even if a branch is almost always taken and then not taken once, the 1-bit BHT will mispredict twice (rather than once) when it is not taken.
- That scheme causes **two** wrong predictions:
  - At the last loop iteration, since the prediction bit will say taken, while we need to exit from the loop.
  - When we re-enter the loop, at the end of the first loop iteration we need to take the branch to stay in the loop, while the prediction bit say to exit from the loop, since the prediction bit was flipped on previous execution of the last iteration of the loop.
- For example, if we consider a loop branch whose behavior is taken nine times and not taken once, the prediction accuracy is only 80% (due to 2 incorrect predictions and 8 correct ones).

1) 2-bit Branch History Table

- The prediction must miss twice before it is changed.
- In a loop branch, at the last loop iteration, we do not need to change the prediction.
- For each index in the table, the 2 bits are used to encode the four states of a finite state machine.

1) FSM for 2-bit Branch History Table



1) n-bit Branch History Table

- Generalization: n-bit saturating counter for each entry in the prediction buffer.
  - The counter can take on values between *0* and 2^n-1
  - When the counter is greater than or equal to one-half of its maximum value (2^n-1), the branch is predicted as taken.
  - Otherwise, it is predicted as untaken.
- As in the 2-bit scheme, the counter is incremented on a taken branch and decremented on an untaken branch.
- Studies on n-bit predictors have shown that 2-bit predictors behave almost as well.

1) Accuracy of 2-bit Branch History Table

- For IBM Power architecture executing SPEC89 benchmarks , a 4K-entry BHT with 2-bit per entry results in:
  - Prediction accuracy from 99% to 82% (i.e. misprediction rate from 1% to 18%)
  - Almost similar performance with respect to an infinite buffer with 2-bit per entry.

2) Correlating Branch Predictors

- The 2-bit BHT uses 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.
- We try to exploit the correlation existing among different branches: branches are partially based on the same conditions => they can generate information that can influence the behavior of other branches;

2) Example of Correlating Branches

```
    If(a==2) a = 0; bb1
L1: If(b==2) b = 0; bb2
L2: If(a!=b) {};    bb3
```

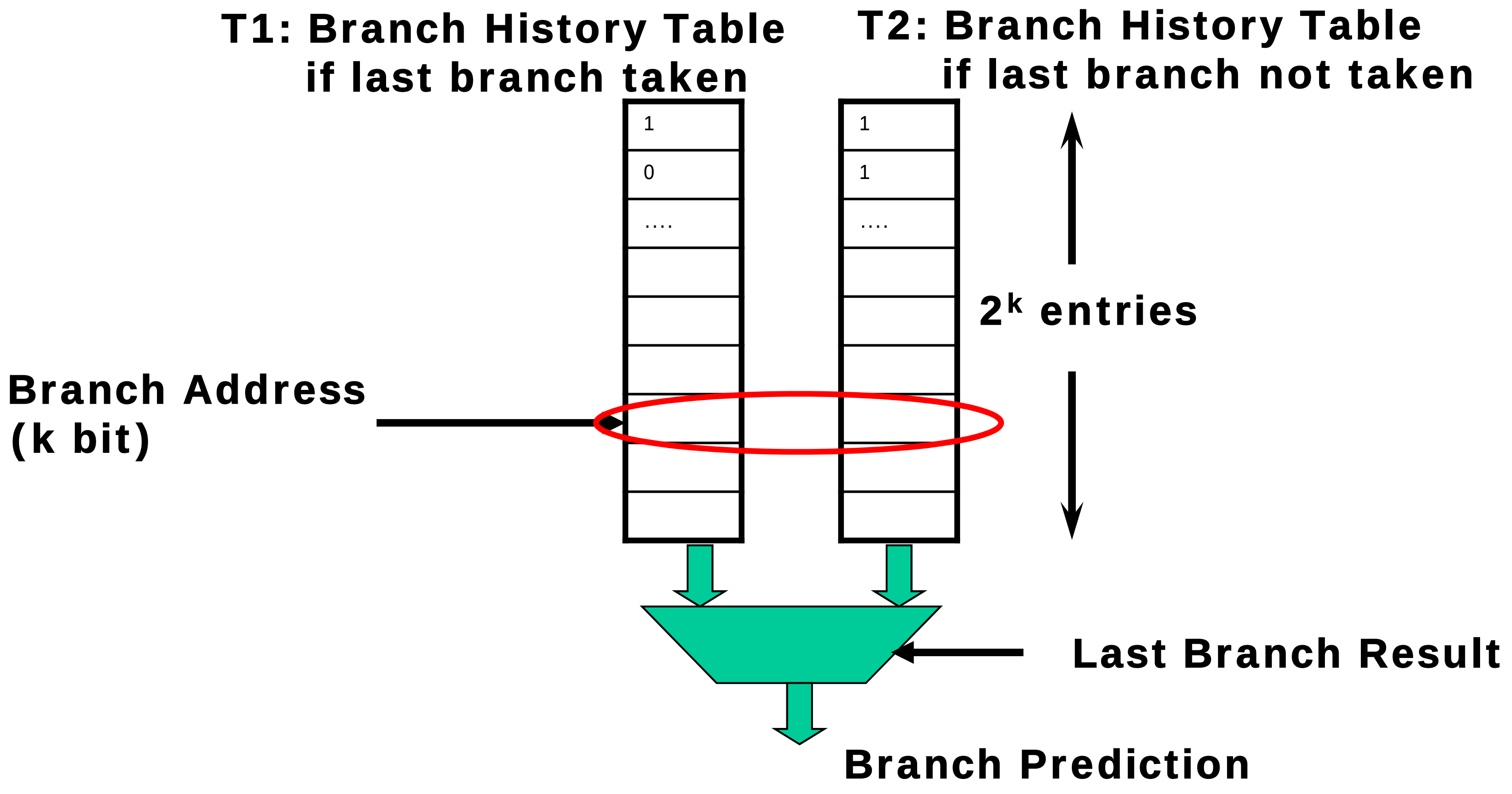
```
      subi r3,r1,2
      bnez r3,L1; bb1
      add  r1,r0,r0
L1:   subi r3,r2,2
      bnez r3,L2; bb2
      add  r2,r0,r0
L2:   sub  r3,r1,r2
      beqz r3,L3; bb3
L3:
```

Branch **bb3** is correlated to previous branches **bb1** and **bb2**.
If previous branches are both *not taken*,
then **bb3** will be *taken* **(a!=b)**

2) Correlating Branch Predictors

- Branch predictors that use the behavior of other branches to make a prediction are called **Correlating Predictors** or **2-level Predictors.**
- Example a **(1,1) Correlating Predictors** means a 1-bit predictor with 1-bit of correlation: **the behavior of last branch is used to choose among a pair of 1-bit branch predictors.**

2) Correlating Branch Predictors: Example


T1: Branch History Table if last branch taken
T2: Branch History Table if last branch not taken
1
0
....
1
1
....
2^k entries
Branch Address (k bit)
Last Branch Result
Branch Prediction

2) Correlating Branch Predictors

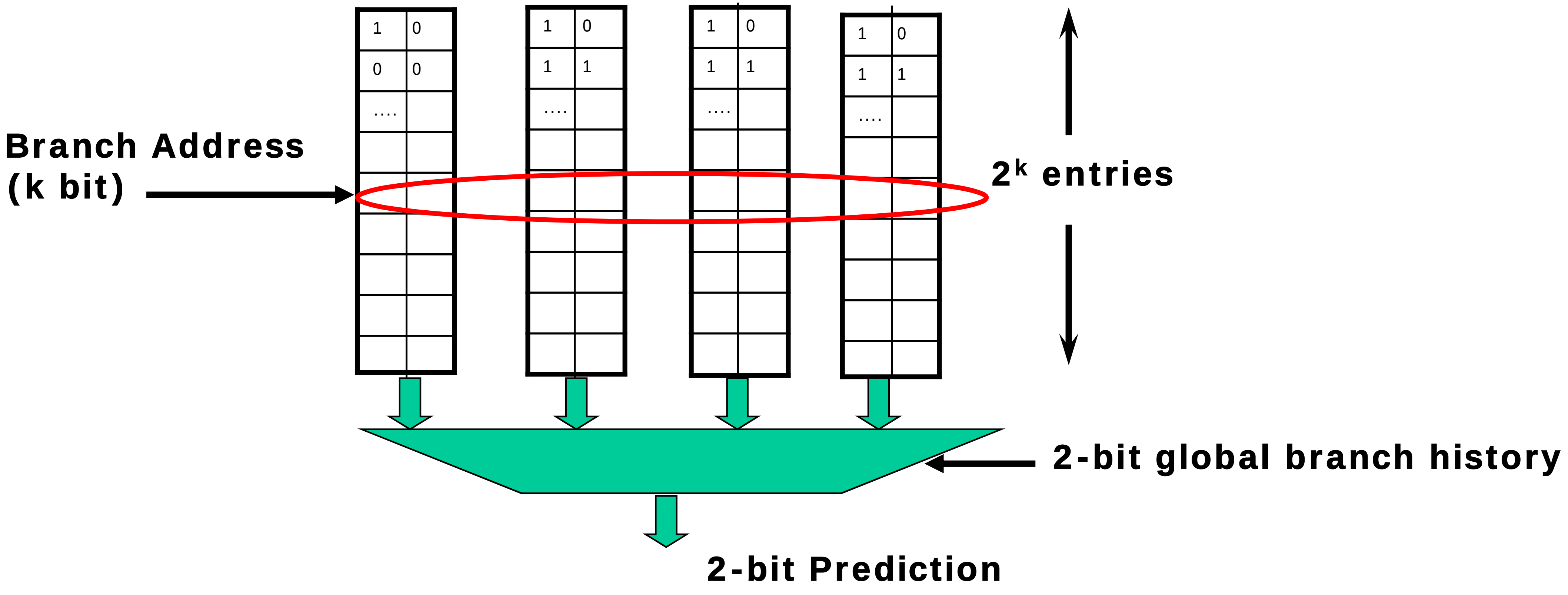
- Record if the most recently executed branches have been *taken* o *not taken*.
- The branch is predicted based on the previous executed branch by selecting the appropriate 1-bit BHT:
  - One prediction is used if the last branch executed was **taken**
  - Another prediction is used if the last branch executed was **not taken.**
- In general, the last branch executed is *not* the same instruction as the branch being predicted (although this can occur in simple loops with no other branches in the loops).

2) (m, n) Correlating Branch 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).

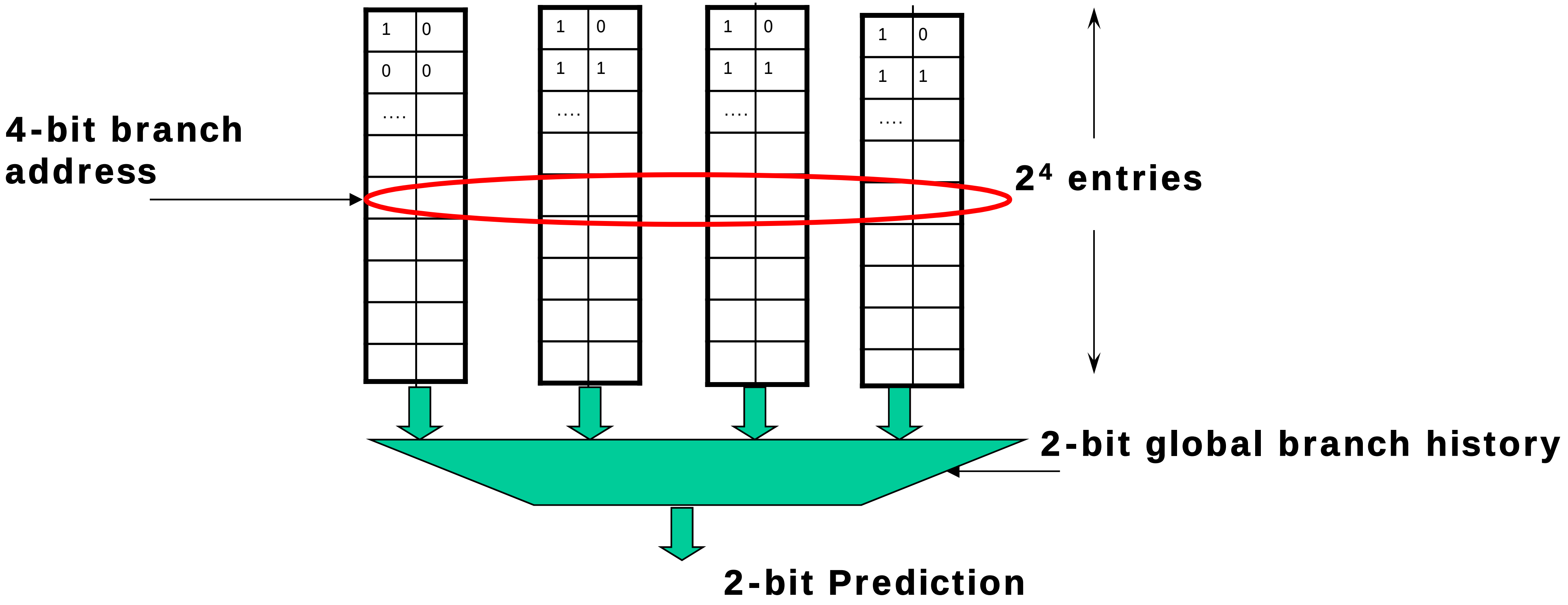
2) (2, 2) Correlating Branch Predictors

- A **(2, 2)** correlating predictor has 4 2-bit Branch History Tables.
    - It uses the 2-bit global history to choose among the 4 BHTs.



2) Example of (2, 2) Correlating Predictor

- Example: a **(2, 2)** correlating predictor with 64 total entries $\Rightarrow$ 6-bit index composed of: 2-bit global history and 4-bit low-order branch address bits



2) Example of (2, 2) Correlating Predictor

- Each BHT is composed of 16 entries of 2-bit each.
- The 4-bit branch address is used to choose four entries (a row).
- 2-bit global history is used to choose one of four entries in a row (one out of four BHTs)

2) Accuracy of Correlating Predictors

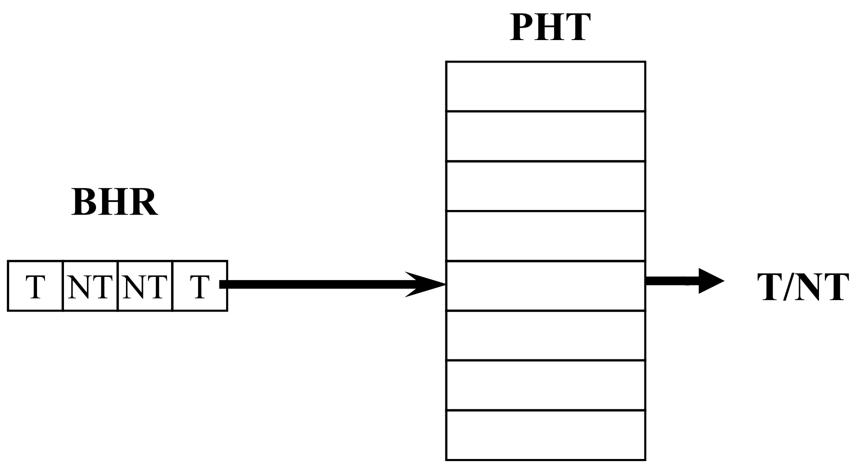
- A 2-bit BHT predictor with no global history is simply a (0, 2) predictor.
- By comparing the performance of a 2-bit simple predictor with 4K entries and a (2,2) correlating predictor with 1K entries.
- The (2,2) predictor not only outperforms the simply 2-bit predictor with the same number of total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

3) 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 (i.e. T, NT, NT, T) *(used as a global history)*
- The second level history is recorded in one (or more) tables called **Pattern History Table (PHT)** of two-bit saturating counters *(used as a local history)*
- 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.

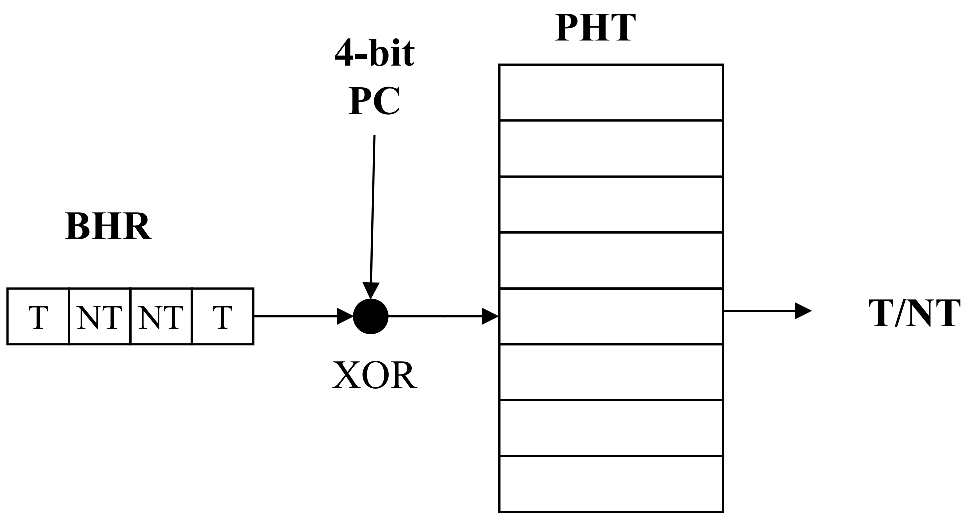
3) GA Predictor

- The global 2-level predictor uses the correlation between the current branch and the other branches in the global history to make the prediction
- **GAs:** Global and local predictor
  - 2-level predictor: PHT (local history) indexed by the content of BHR (global history)



3) GShare Predictor

- Variation of the GA predictor where want to correlate the BHR recording the outcomes of the most recent branches (global history) with the low-order bits of the branch address
- **GShare:** We make the XOR of 4-bit BHR (global history) with the low-order 4-bit of PC (branch address) to index the PHT (local history).



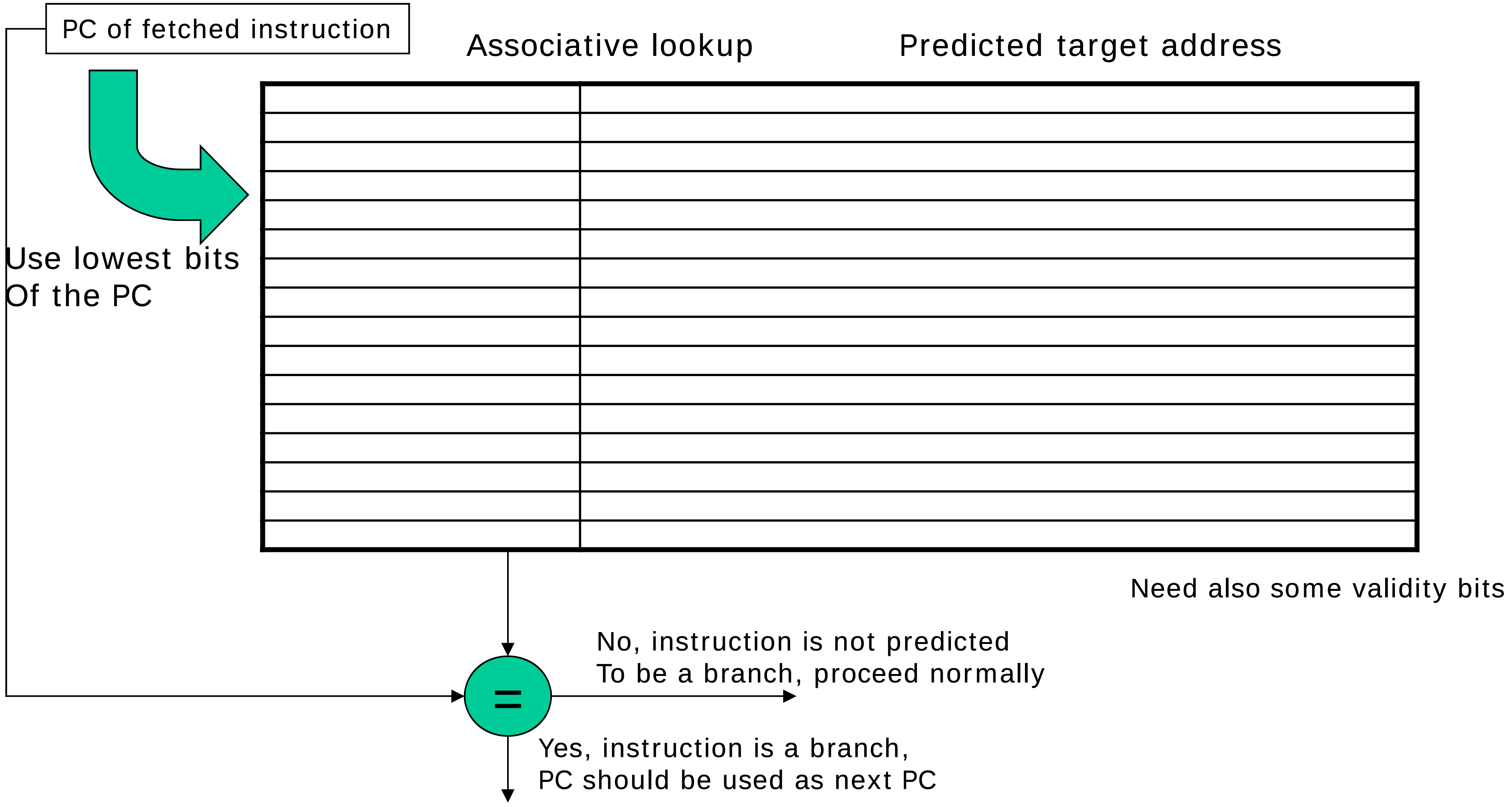
4) Branch Target Buffer

- **Branch Target Buffer (Branch Target Predictor)** is a cache storing the **predicted branch target address**
- We access the **BTB** in the **IF stage** by 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 BTA

- The predicted BTA is expressed as PC-relative

4) Structure of a Branch Target Buffer

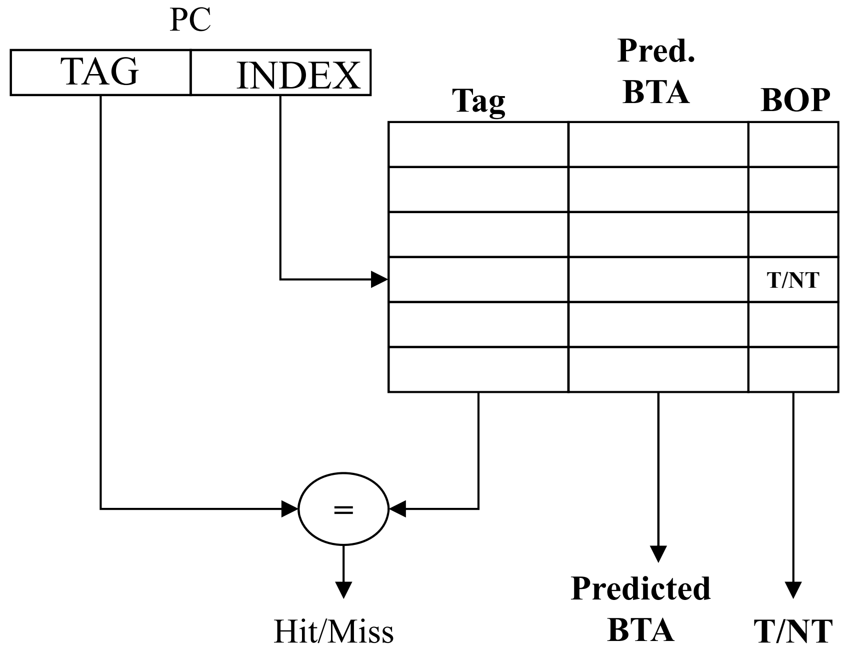


PC of fetched instruction
Associative lookup
Predicted target address
Use lowest bits
Of the PC
Need also some validity bits
No, instruction is not predicted
To be a branch, proceed normally
=
Yes, instruction is a branch,
PC should be used as next PC

4) Structure of a Branch Target Buffer

- In the BTB we need to store the predicted target address only for **taken** branches.

- BTB entry:
  - **Tag** + **Predicted BTA** (expressed as PC-relative for conditional branches) + Prediction state bits as in a Branch Outcome Predictor.



Speculation

- Without branch prediction, the amount of parallelism is quite limited, since it is limited to within a **basic block** – a straight-line code sequence with no branches in except to the entry and no branches out except at the exit.
- Branch prediction techniques can help to achieve significant amount of parallelism.
- We can further exploit ILP across multiple basic blocks overcoming control dependences by speculating on the outcome of branches and executing instructions as if our guesses were correct.
- With **speculation,** we fetch, issue and execute instructions as if out branch predictions were always correct, providing a mechanism to handle the situation where the speculation is incorrect.
- Speculation can be supported by the **compiler** or by the **hardware**.

References

- An introduction to the branch prediction problem can be found in **Chapter 3** of: J. Hennessy and D. Patterson, "Computer Architecture, a Quantitative Approach", Morgan Kaufmann, Third edition, May 2002.

- A survey of basic branch prediction techniques can be found in: D. J. Lalja, "Reducing the Branch Penalty in Pipelined Processors", Computer, pages 47-55, July 1988.

- A more detailed and advanced survey of the most used branch predictor architectures can be found in: M. Evers and T.-Y. Yeh, "Understanding Branches and Designing Branch Predictors for High-Performance Microprocessors", Proceedings of the IEEE, Vol. 89, No. 11, pages 1610-1620, November 2001.