Branch Prediction Techniques

Donatella Sciuto: sciuto@elet.polimi.it

Marco D. Santambrogio: marco.santambrogio@polimi.it

Outline

- Dealing with Branches in the Processor Pipeline
- The Problem of Branch Hazards
- Branch Prediction Techniques
  - Static Branch Prediction
  - Dynamic Branch Prediction
- Performance of Branch Schemes

Conditional Branch Instructions

- **Conditional Branch Instruction:** the branch is taken only 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.
- Examples of 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,offset`

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 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,offset
```

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)

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.

Performance of Branch Schemes

- What is the performance impact of conditional branches?

$$\text{Pipeline Speedup} = \frac{\text{Pipeline Depth}}{1 + \text{Pipe Stall Cycles per Instruction due to Branches}}$$

$$= \frac{\text{Pipeline Depth}}{1 + \text{Branch Frequency} \times \text{Branch Penalty}}$$

Branch Stall Impact

Branch Stall Impact

- If CPI = 1, 30% branch,
  Stall 3 cycles => new CPI = 1.9!

Branch Stall Impact

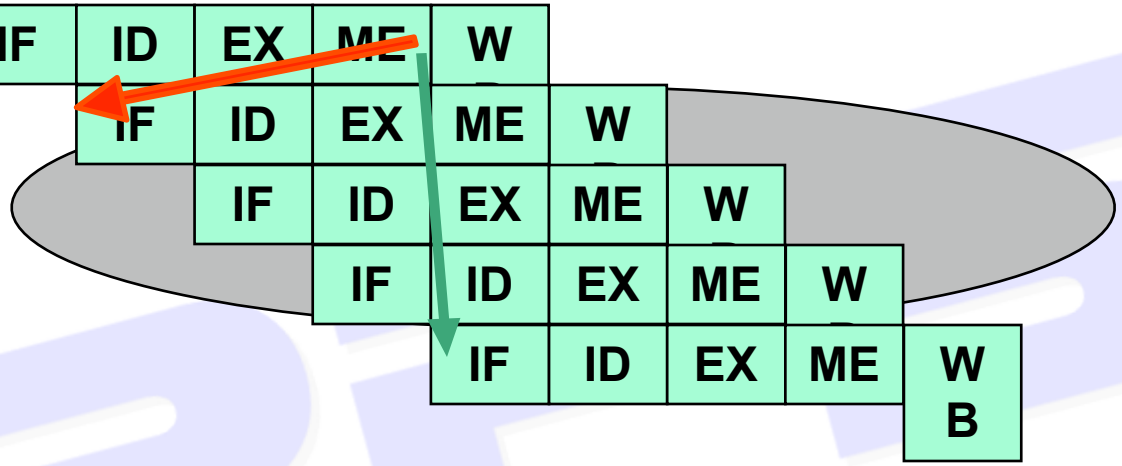
- If CPI = 1, 30% branch,
  Stall 3 cycles => new CPI = 1.9!
- Two part solution:
  - Determine branch taken or not sooner, AND
  - Compute taken branch address earlier
- MIPS branch tests if register = 0 or ≠ 0
- MIPS Solution:
  - Move Zero test to ID/RF stage
  - Adder to calculate new PC in ID/RF stage
  - 1 clock cycle penalty for branch versus 3

Branch Hazards

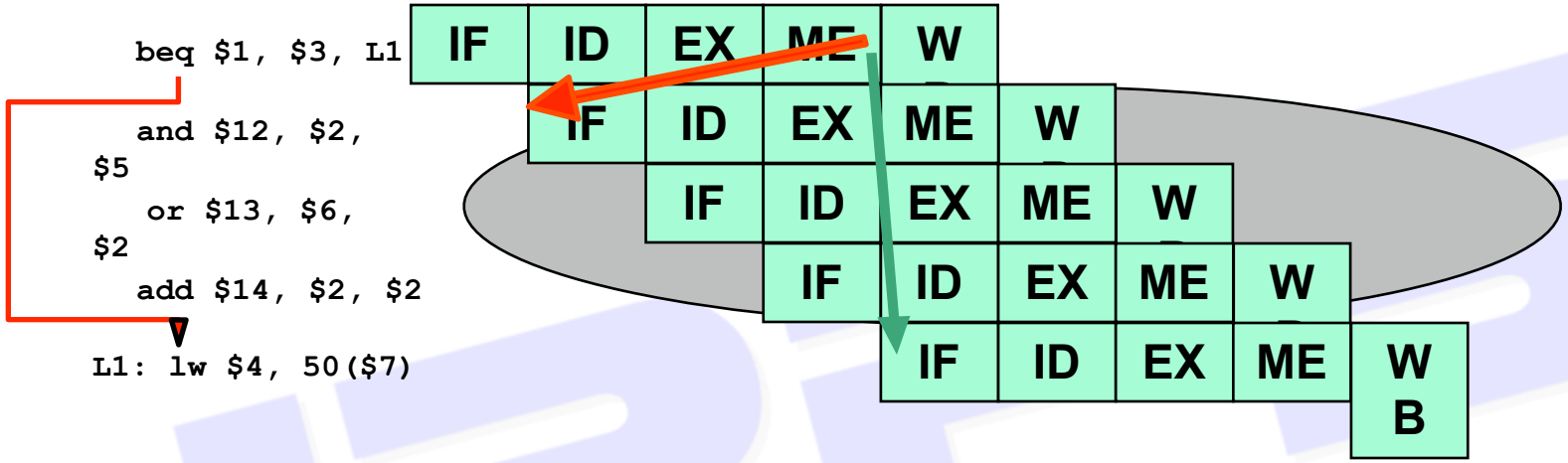
- To feed the pipeline we need to fetch a new instruction at each clock cycle, but the branch decision (to change or not change the PC) is taken during the MEM stage.
- This delay to determine the correct instruction to fetch is called **Control Hazard or Conditional Branch Hazard**
- If a branch changes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

```
   beq $1, $3, L1
   and $12, $2,
$5
    or $13, $6,
$2
   add $14, $2, $2
L1: lw $4, 50($7)
```

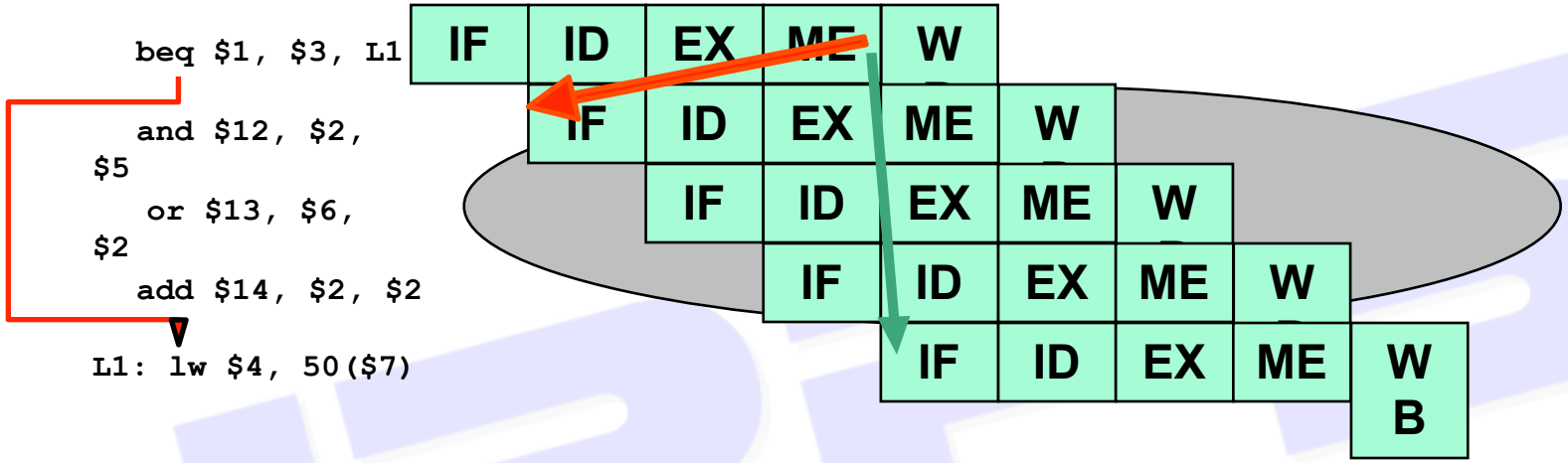


Branch Hazards: Example



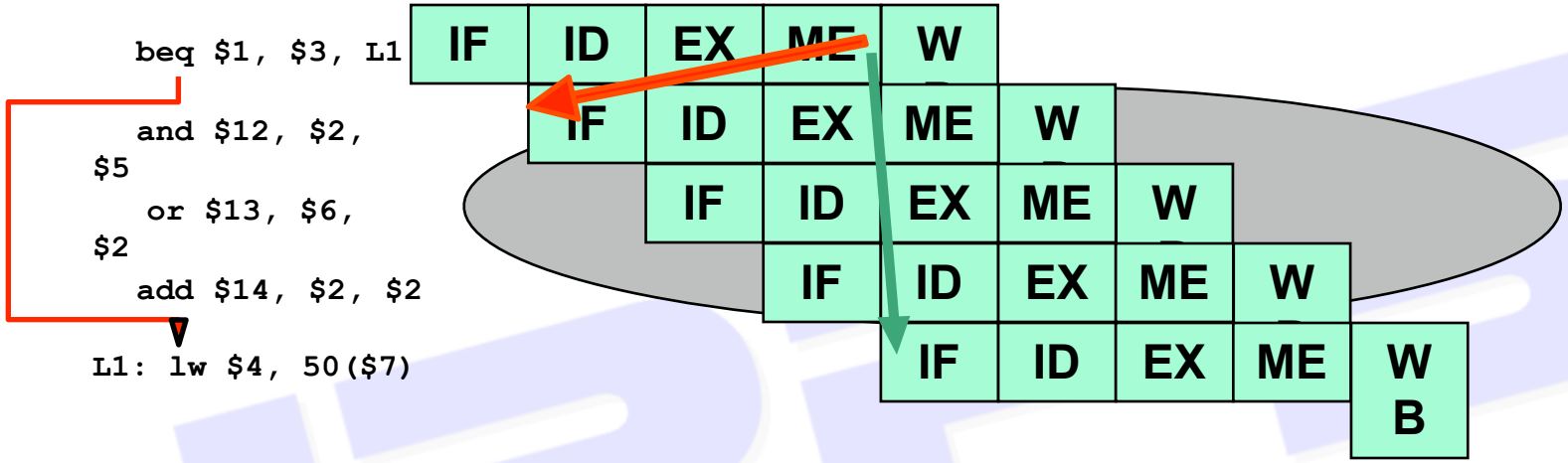
- The branch instruction may or may not change the PC in MEM stage, but the next 3 instructions are fetched and their execution is started.

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 OK

Branch Hazards: Example

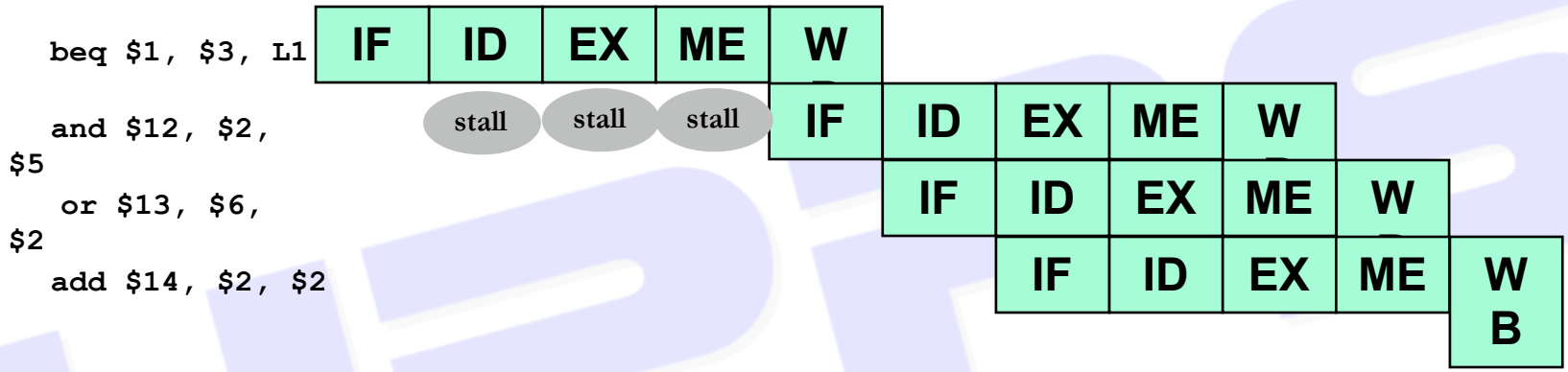


- The branch instruction may or may not change the PC in MEM stage, but the next 3 instructions are fetched and their execution is started.
- If the branch is **not taken,** the pipeline execution is OK
- If the branch is **taken,** it is necessary to ***flush*** the next 3 instructions in the pipeline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

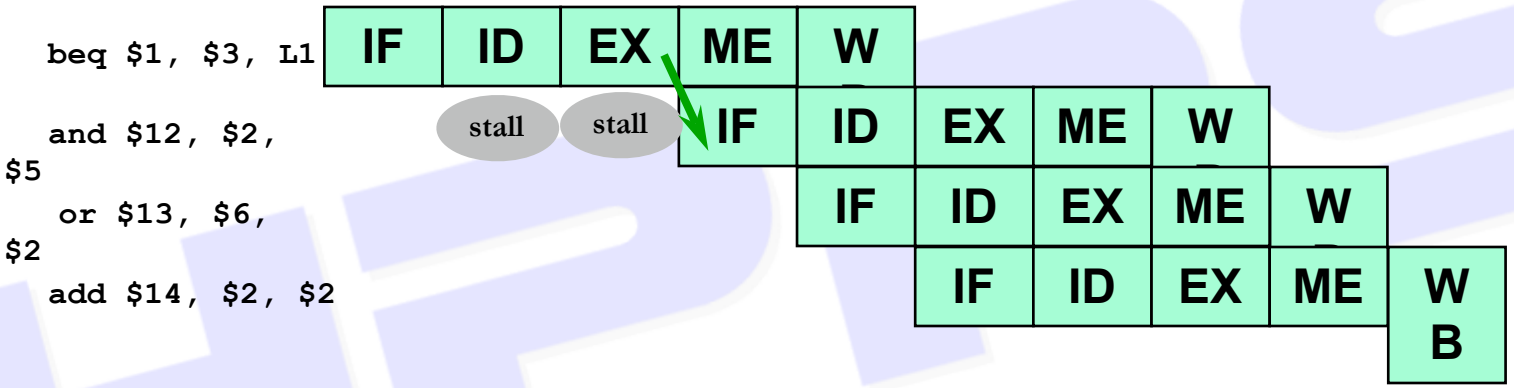
Branch Hazards: Solutions

- To stall the pipeline until the branch decision is taken **(stalling until resolution)** and then fetch the correct instruction flow.
  - Without forwarding : for **three** clock cycles
  - With forwarding: for **two** clock cycles
- If the branch is *not taken*, the three cycles penalty is not justified ⇒ throughput reduction.
- We can assume the ***branch not taken***, and **flush** the next 3 instructions in the pipeline only if the branch will be taken.

Branch Stalls without Forwarding



Branch Stalls with Forwarding



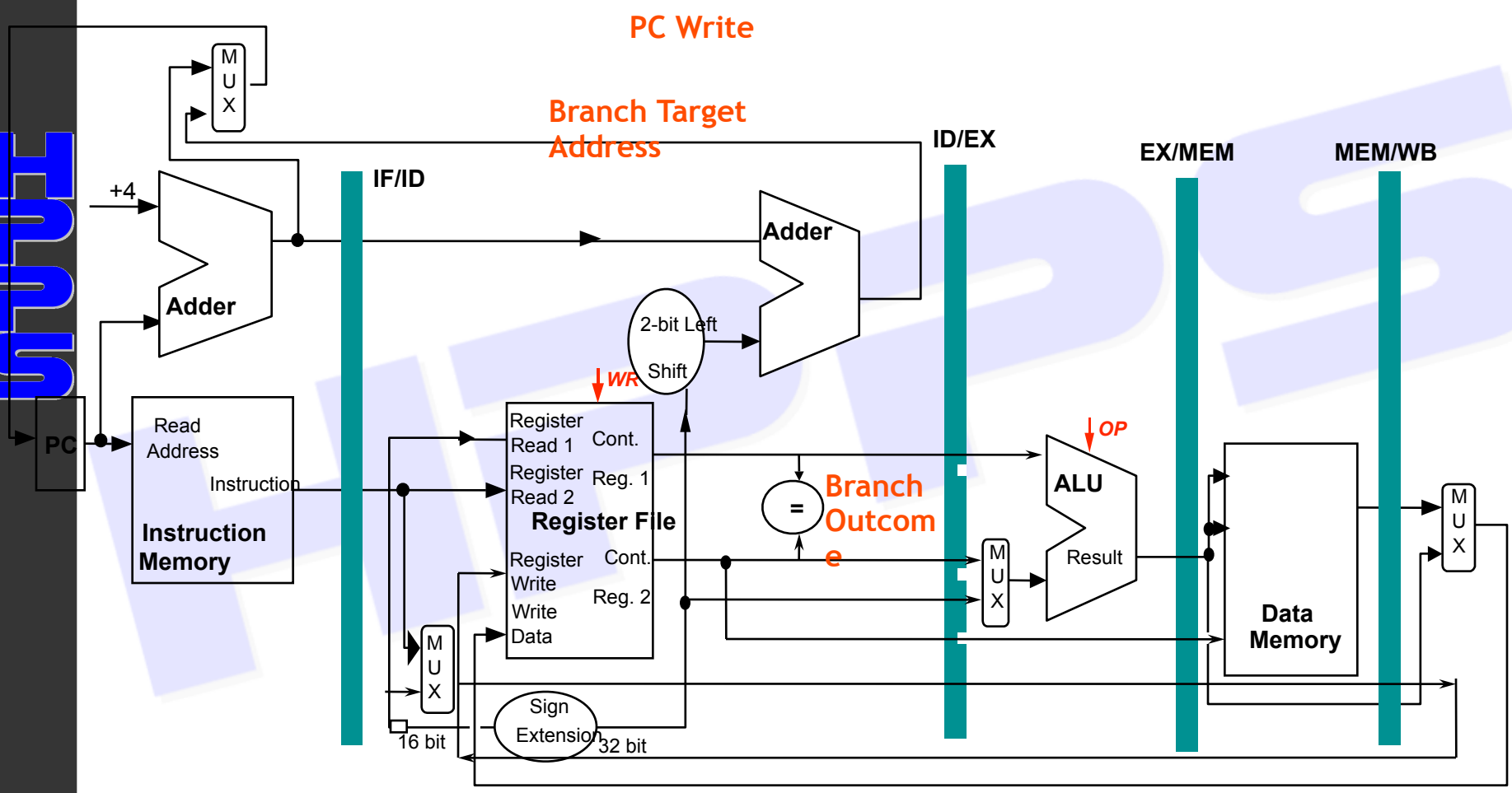
Early Evaluation of the PC

- To improve performance in case of branch hazards, we need to add hardware resources to:
  1. Compare registers to derive branch outcome
  2. Compute branch target address
  3. Update the PC register

  as soon as possible in the pipeline.

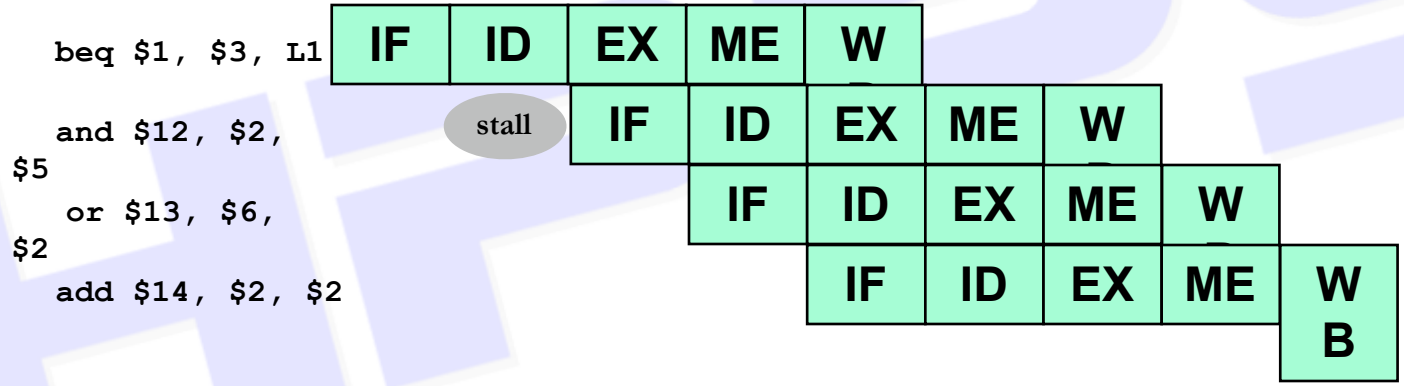
- MIPS processor compares registers, computes branch target address and updates PC **during ID stage.**

MIPS Processor: Early Evaluation of the PC



MIPS Processor: Early Evaluation of the PC

- Each branch costs **one stall** to fetch the correct instruction flow: (PC+4) or branch target address



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 techniques to deal with this performance loss.

Branch Prediction Techniques

Branch Prediction Techniques

- In general, the problem of the branch becomes more important for deeply pipelined processors because the cost of incorrect predictions increases (the branches are solved 4 stages or more after the ID stage)
- **Main goal of branch prediction techniques:** try to predict ASAP the result of a branch instruction.
- The performance of a branch prediction technique depends on:
  - **Accuracy** measured in terms of **percentage of incorrect predictions**.
  - **Cost** of a incorrect prediction measured in terms of time lost to execute useless instructions **(misprediction penalty).**
- We also need to consider **branch frequency:** the importance of accurate branch prediction is higher in programs with higher branch frequency.

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.

Static Branch Prediction Techniques

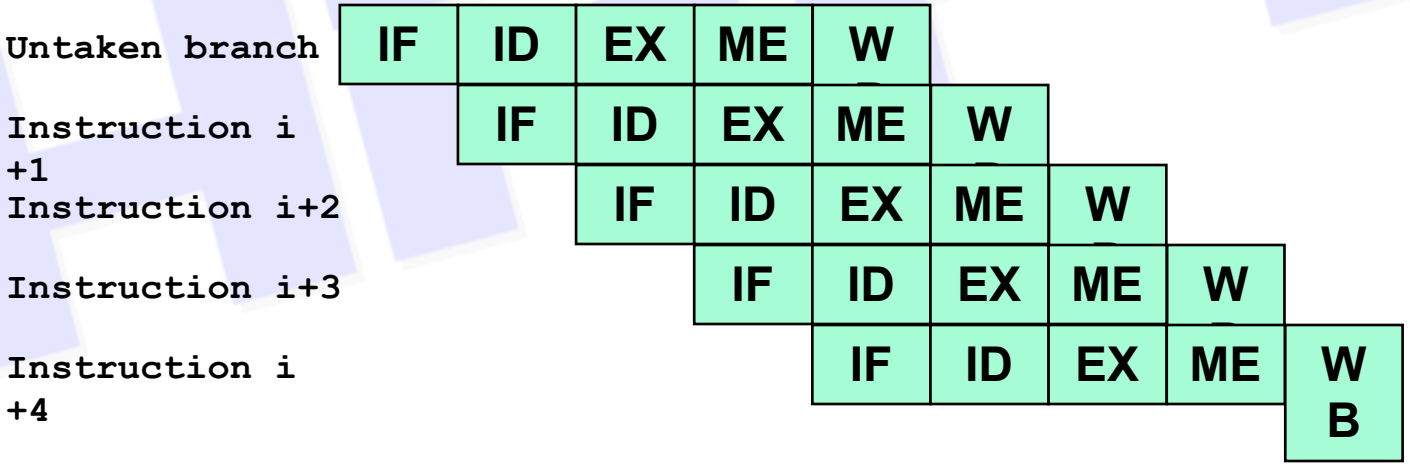
- Static Branch Prediction is used in processors where the expectation is that the branch behavior is highly predictable at compile time.
- Static Branch Prediction can also be used to assist dynamic predictors.

Static Branch Prediction Techniques

- Branch Always Not Taken (Predicted-Not-Taken)
- Branch Always Taken (Predicted-Taken)
- Backward Taken Forward Not Taken (BTFNT)
- Profile-Driven Prediction
- Delayed Branch

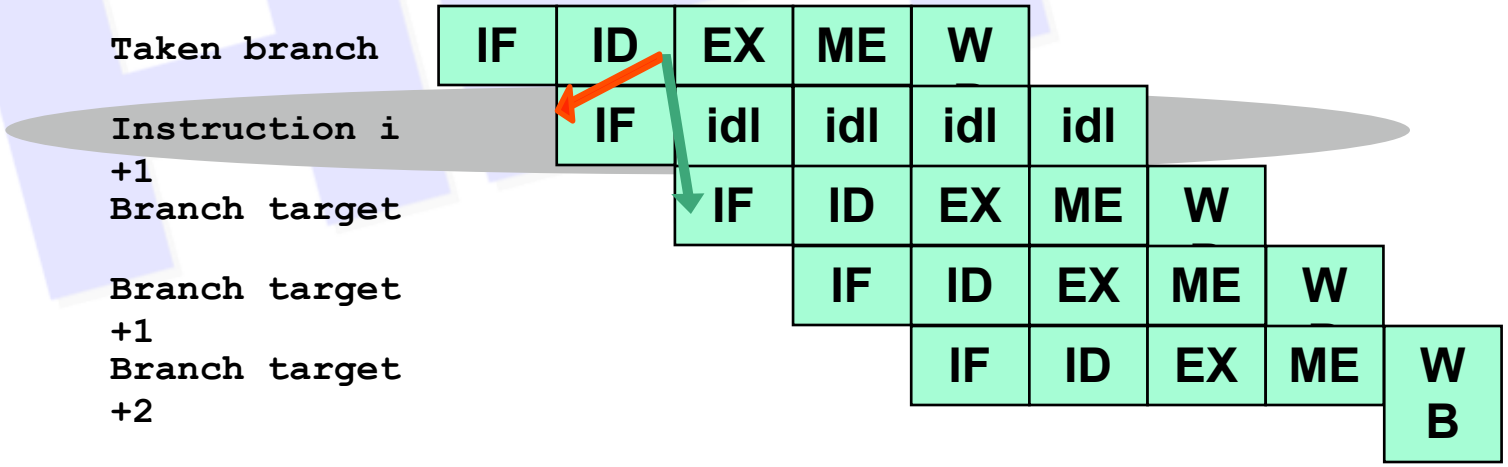
Branch Always Not Taken

- We assume the **branch will not taken,** thus 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.



Branch Always Not Taken

- 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 ⇒ **One-cycle penalty**



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.
- The predicted-taken scheme makes sense for pipelines where the branch target 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 this approach for this pipeline.

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
      ⇒ we assume the backward-going branches are always taken.
  - Forward-going branches are predicted as not taken

Profile-Driven Prediction

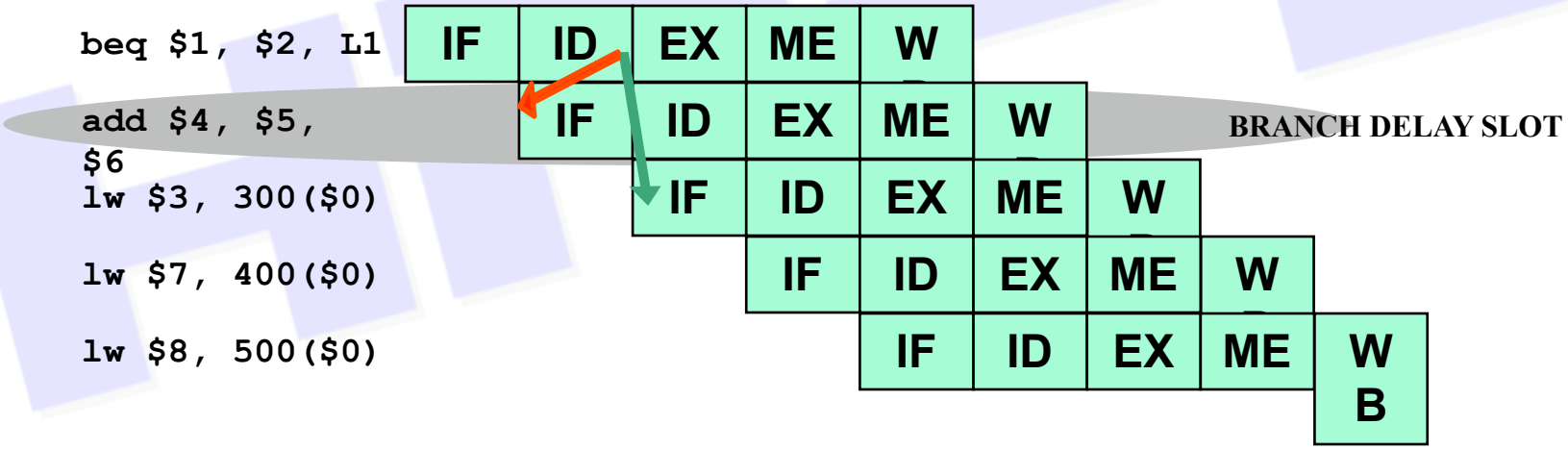
- The branch prediction is based on profiling information collected from earlier runs.
- The method can use compiler hints.

Delayed Branch 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) ⇒ we have only **one-delay slot**
  - Although it is possible to have for some deeply pipeline processors a branch delay longer than one-cycle ⇒ almost all processors with delayed branch have a single delay slot (since it is usually difficult for the compiler to fill in more than one delay slot).

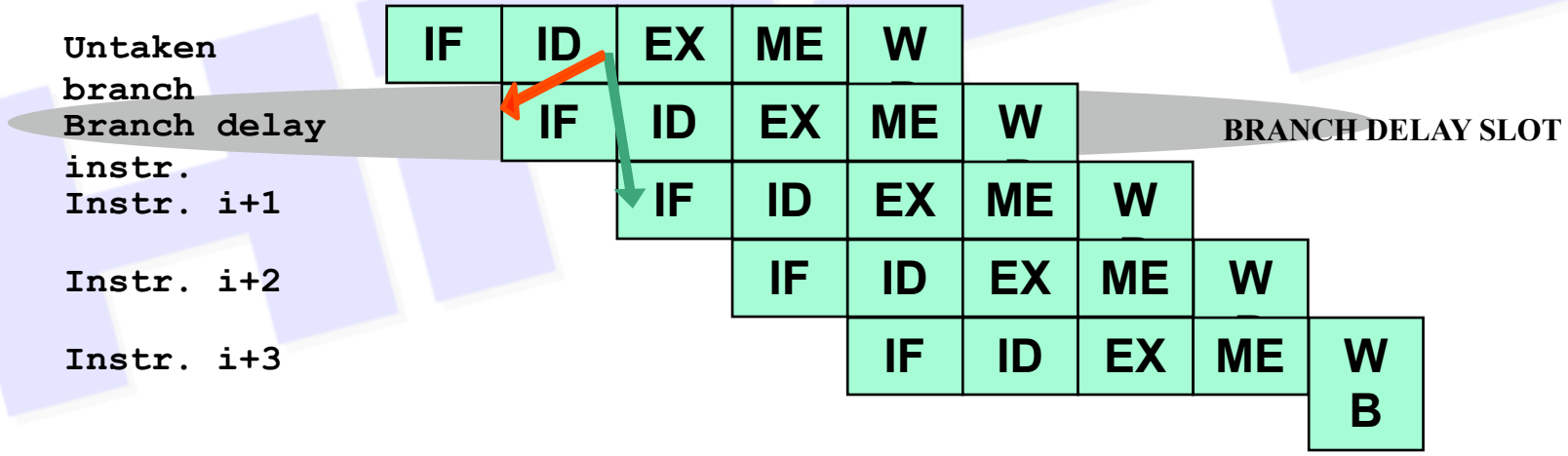
Delayed Branch Technique

- The MIPS compiler always schedules a branch independent instruction after the branch.
- Example: A previous **`add`** instruction with any effects on the branch is scheduled in the **Branch Delay Slot**



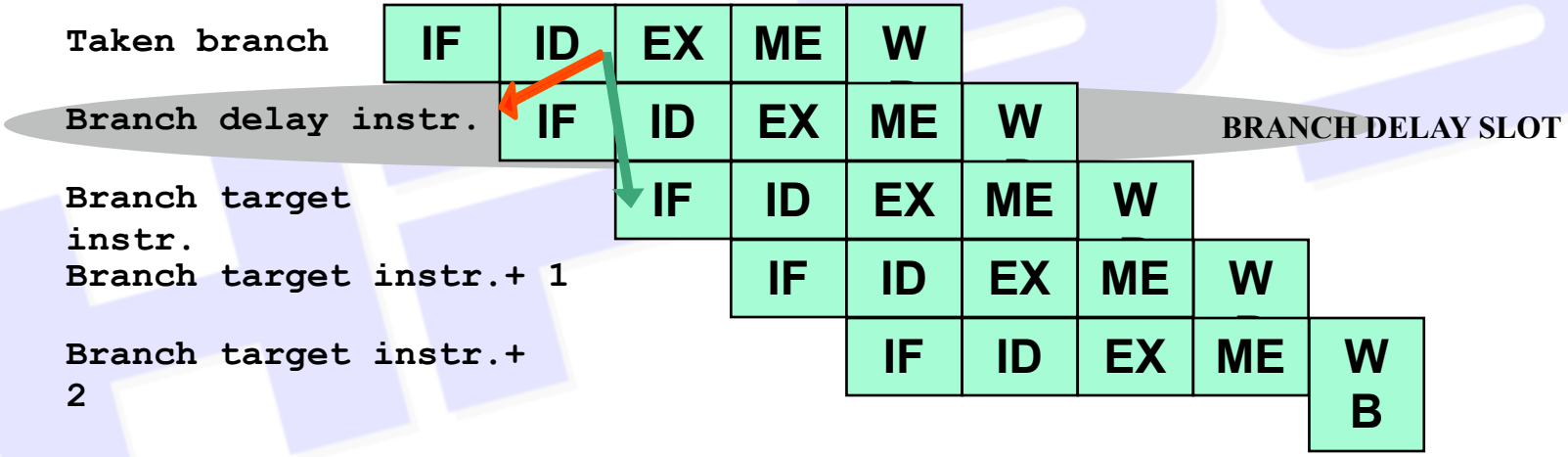
Delayed Branch Technique

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



Delayed Branch Technique

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

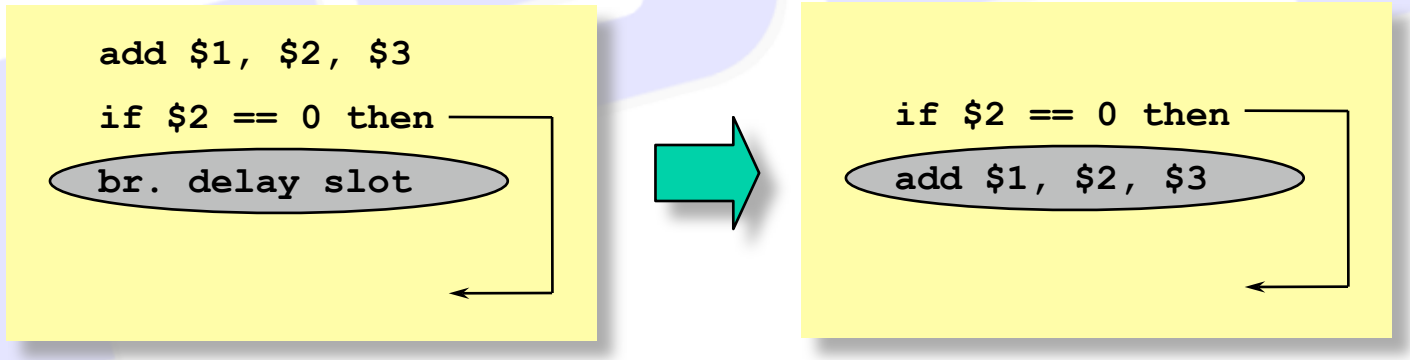


Delayed Branch Technique

- The job of the compiler is to make the instruction placed in the branch delay slot valid and useful.
- There are three ways in which the branch delay slot can be scheduled:
  1. **From before**
  2. **From target**
  3. **From fall-through**

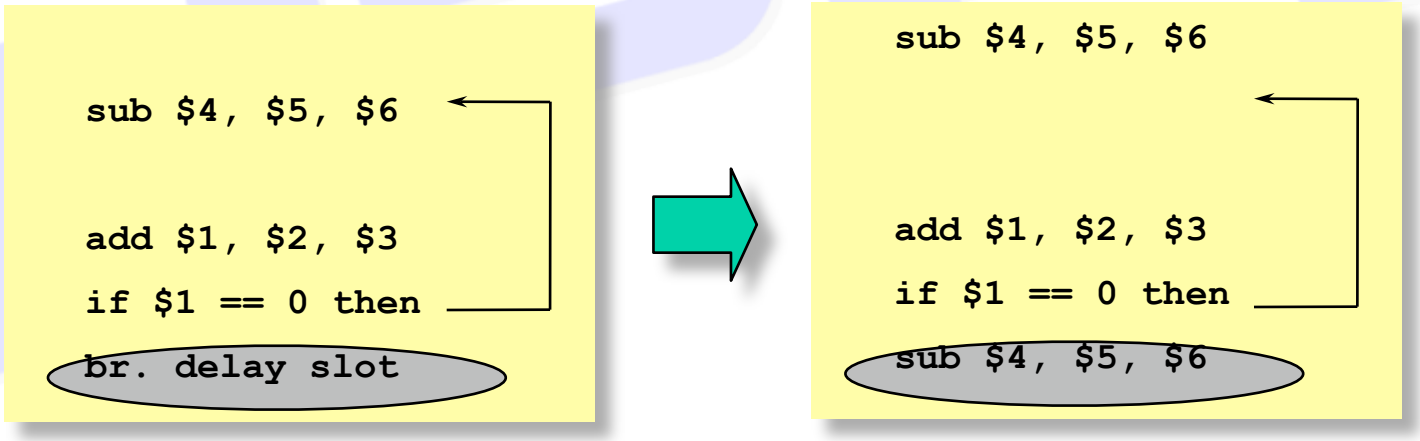
Delayed Branch Technique: From Before

- 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).



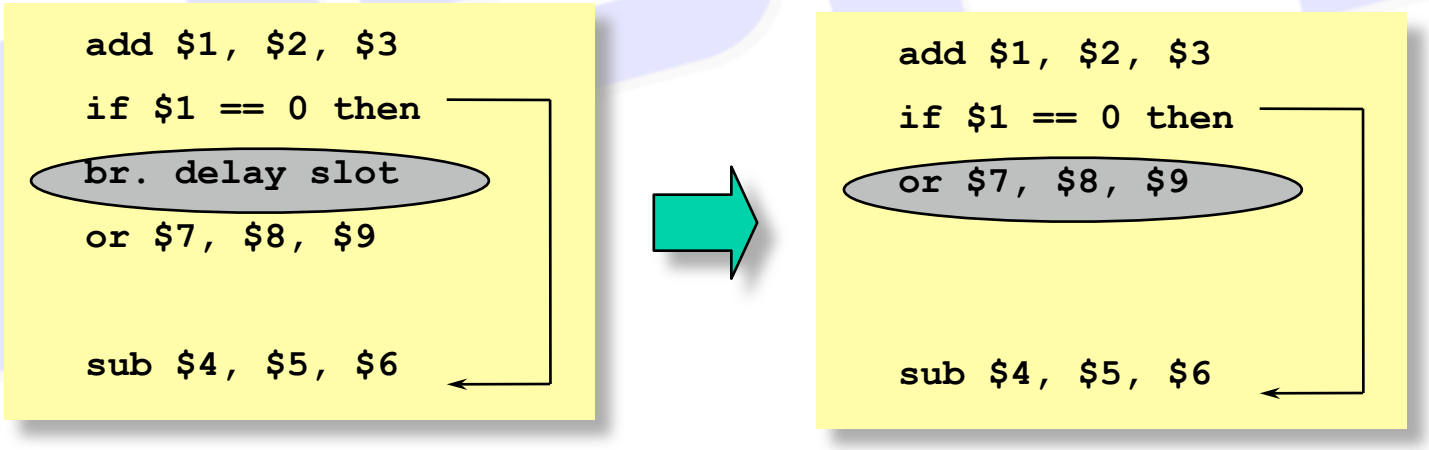
Delayed Branch Technique: From Target

- 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 will need 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).**



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.**



Delayed Branch Technique

- To make the optimization legal for the target an fall-through cases, 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.

Delayed Branch Technique

- In general, the compilers are able to fill about 50% of delayed branch slots with valid and useful instructions, the remaining slots are filled with **`nops`**.
- In deeply pipeline, the delayed branch is longer that one cycle: many slots must be filled for every branch, thus it is more difficult to fill all the slots with useful instructions.

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.

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 into a NOP
- In this way, the compiler need not be as conservative when filling the delay slot.

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 NOP)** whether the branch is untaken.
- Useful approach for backward branches (such as loop branches).
- The branch delay slot must be filled from target.

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.

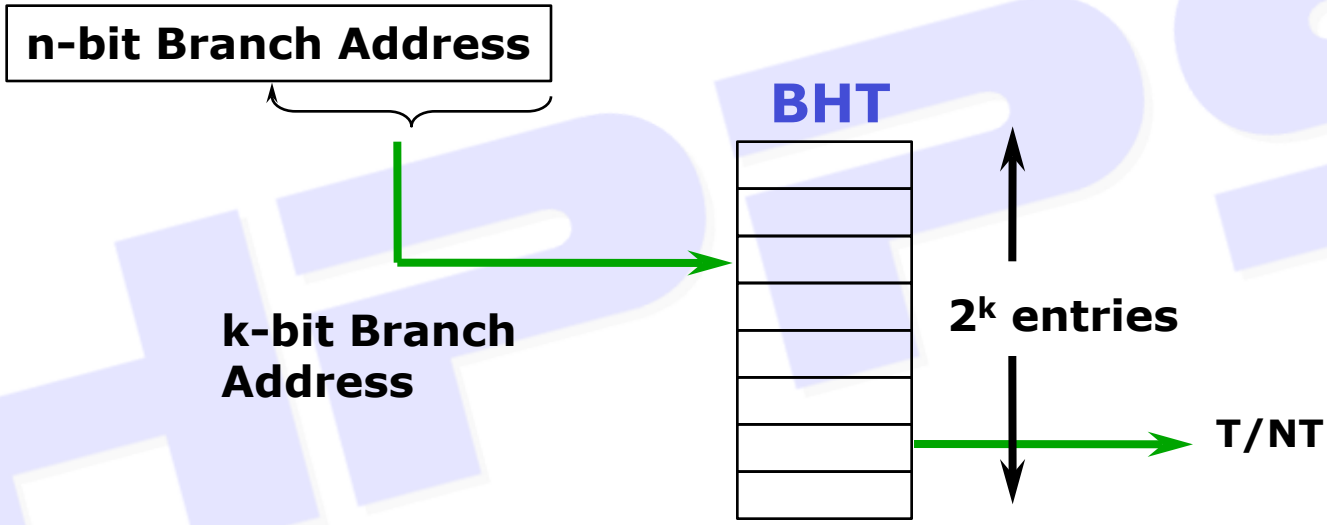
Dynamic Branch Prediction Schemes

- Dynamic branch prediction is based on two interacting mechanisms:
- **Branch Outcome Predictor:**
  - To predict the direction of a branch (i.e. taken or not taken).
- **Branch Target Predictor:**
  - To predict the branch target address in case of taken branch.
- These modules are used by the Instruction Fetch Unit to predict the next instruction to read in the I-cache.
  - If branch is not taken ⇒ PC is incremented.
  - If branch is taken ⇒ BTP gives the target address

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 of the address of the branch instruction.
- 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.
- 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!*

Branch History Table



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).

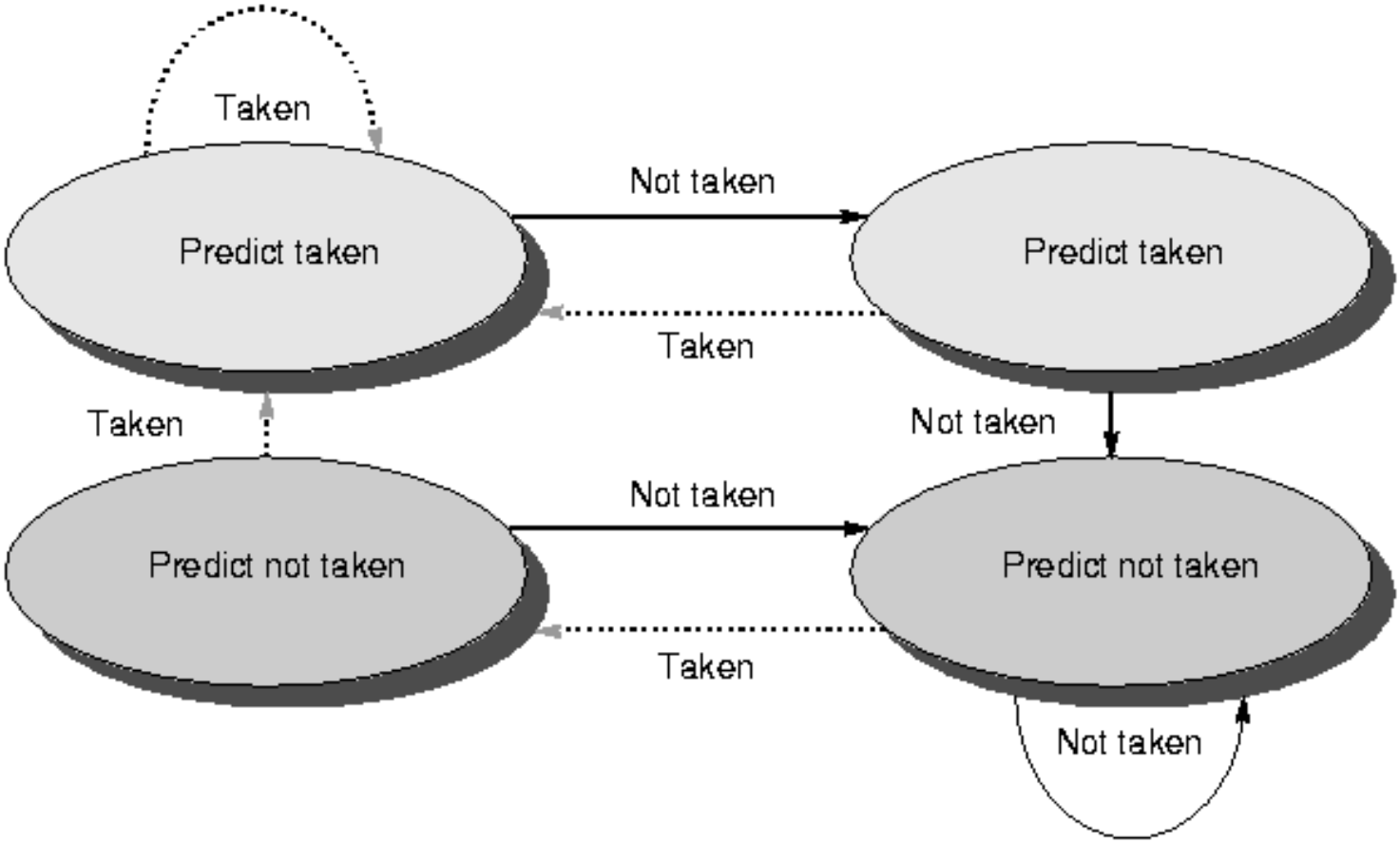
1-bit Branch History Table

- 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).

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.

FSM for 2-bit Branch History Table



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.

Accuracy of 2-bit Branch History Table

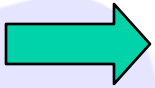
- For IBM Power architecture executing SPEC89 benchmarks , a 4K-entry BHT with 2-bit per entry results in:
  - Prediction accuracy from 99% to 82% (i.e. misprediction rate from 1% to 18%)
  - Almost similar performance with respect to an infinite buffer with 2-bit per entry.

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.

Example of Correlating Branches

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



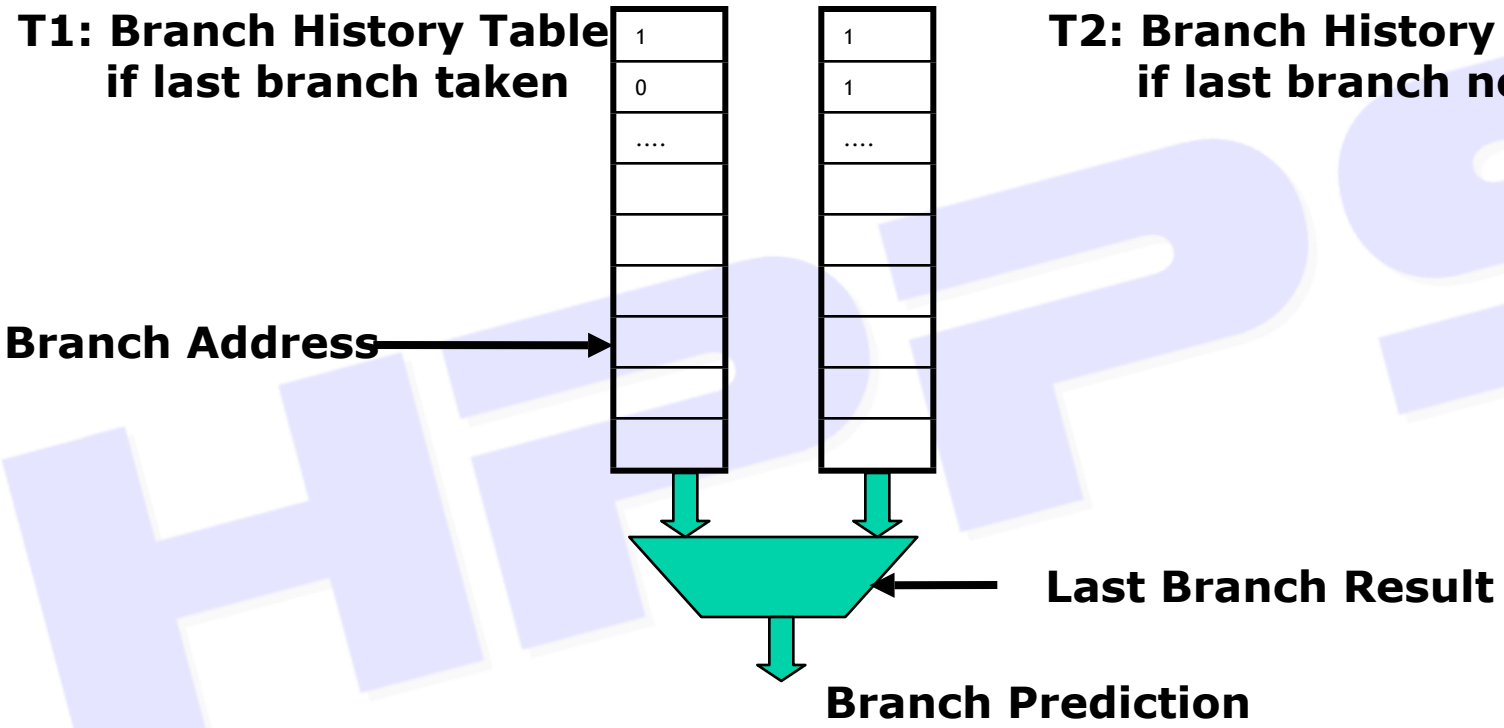
```
      subi r3,r1,2
      bnez r3,L1; bb1
      add  r1,r0,r0
L1:    subi r3,r2,2
      bnez r3,L2; bb2
      add  r2,r0,r0
L2:   sub  r3,r1,r2
      beqz r3,L3; bb3
L3:
```

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

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.

Correlating Branch Predictors: Example



Correlating Branch Predictors

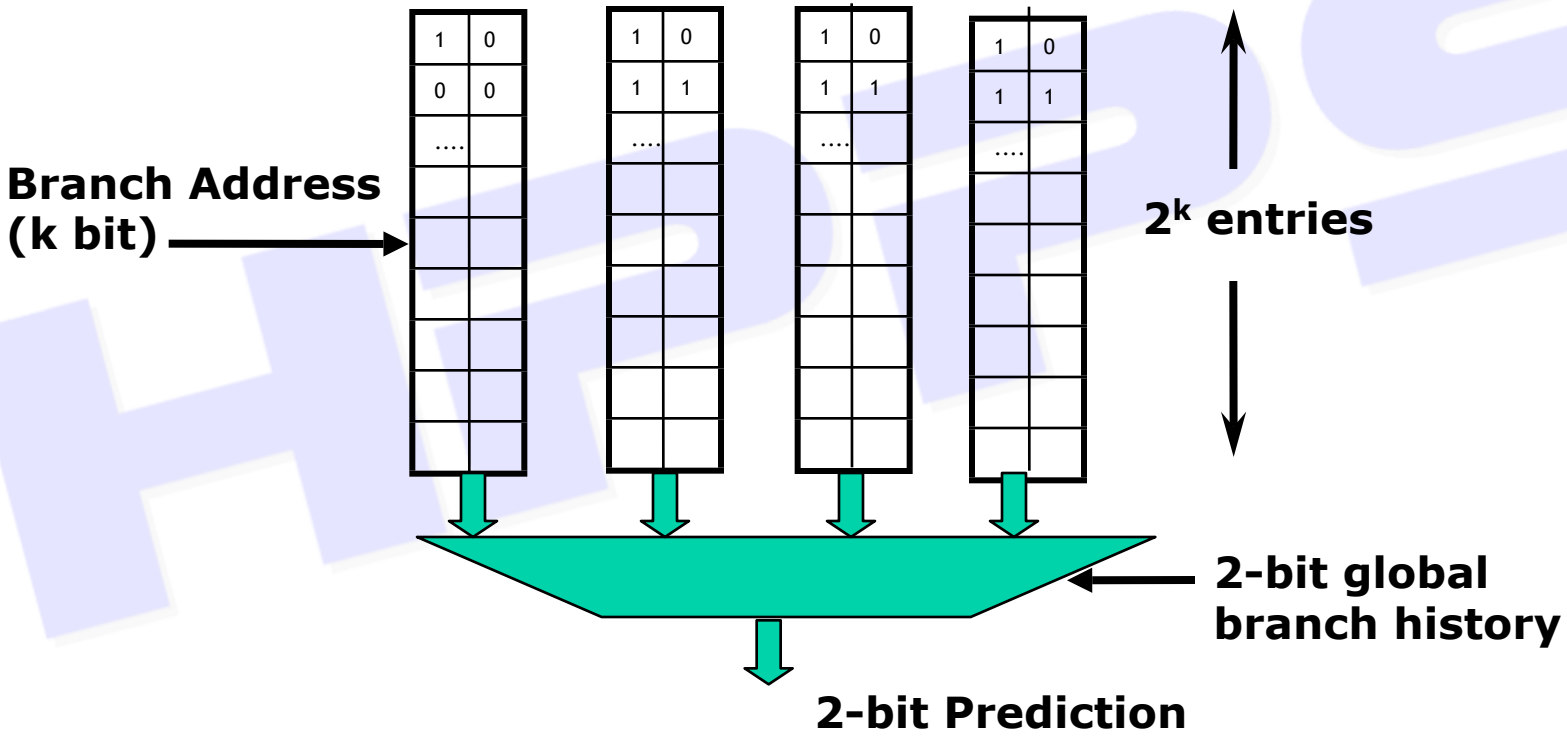
- Record if the k most recently executed branches have been taken o not taken.
- The branch is predicted based on the previous executed branch by selecting the appropriate 1-bit BHT:
  - One prediction is used if the last branch executed was taken
  - Another prediction is used if the last branch executed 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).

(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).

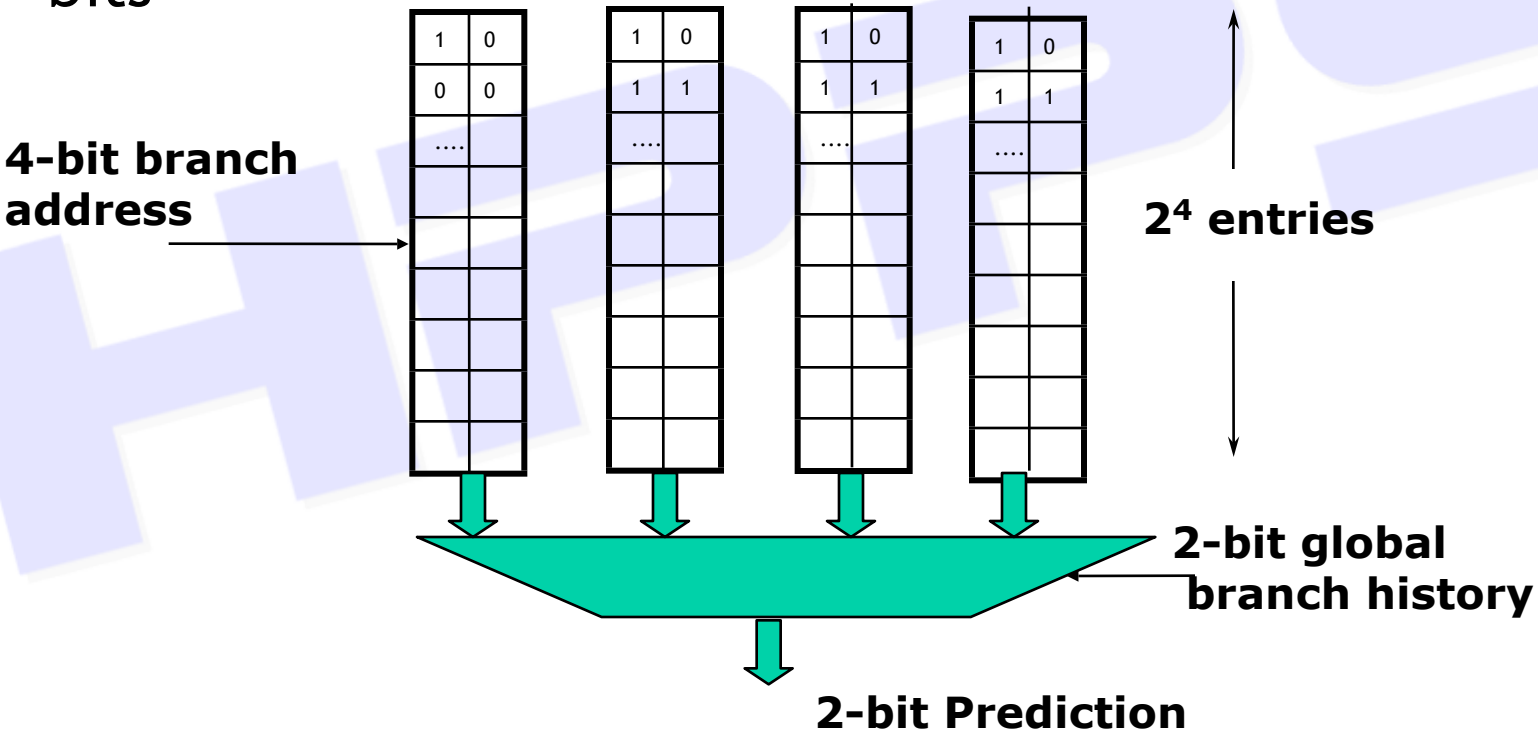
(2, 2) Correlating Branch Predictors

- 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.



Example of (2, 2) Correlating Predictor

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



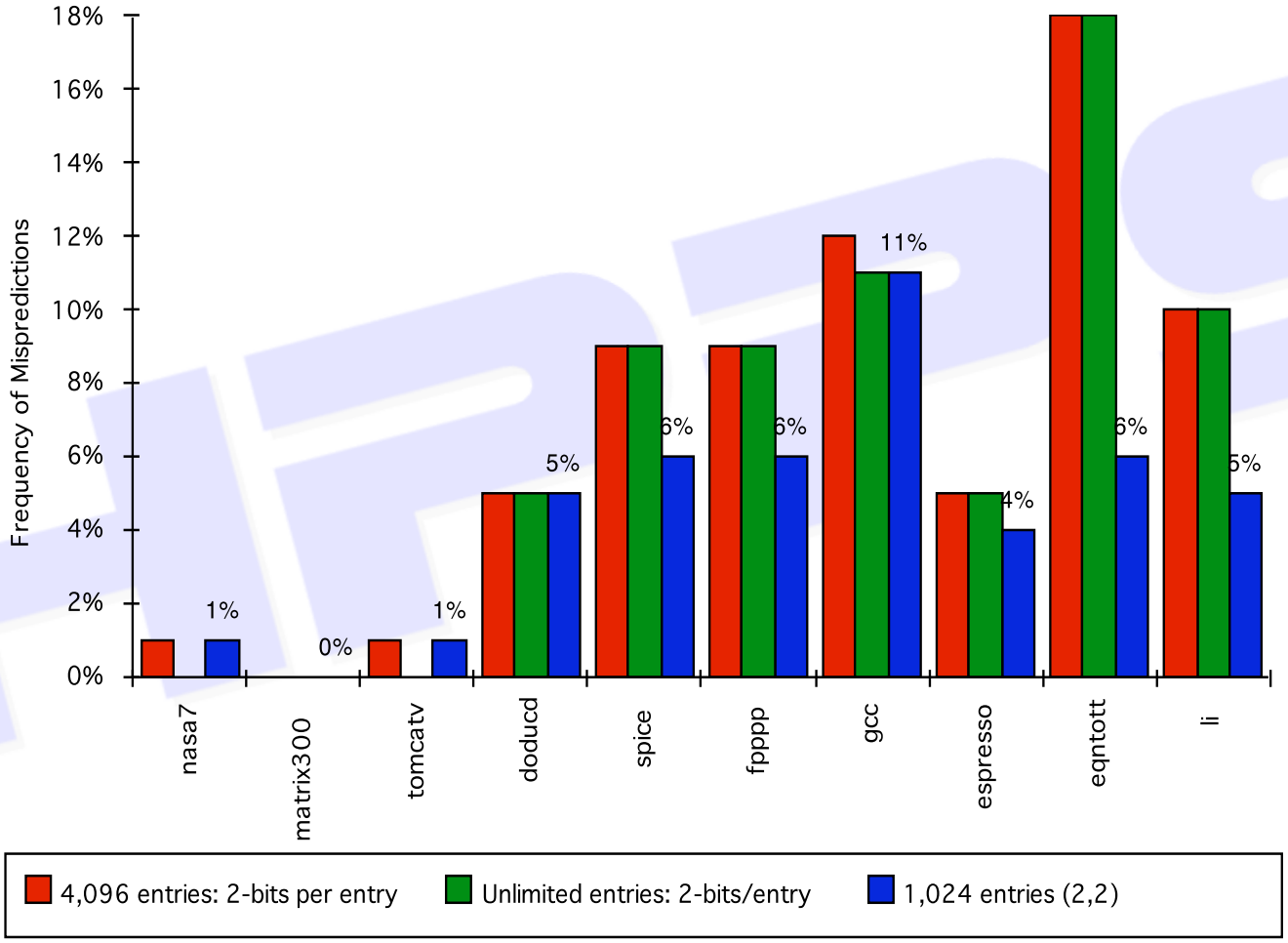
Example of (2, 2) Correlating Predictor

- 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 of four BHTs)

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.

Accuracy of Correlating Predictors

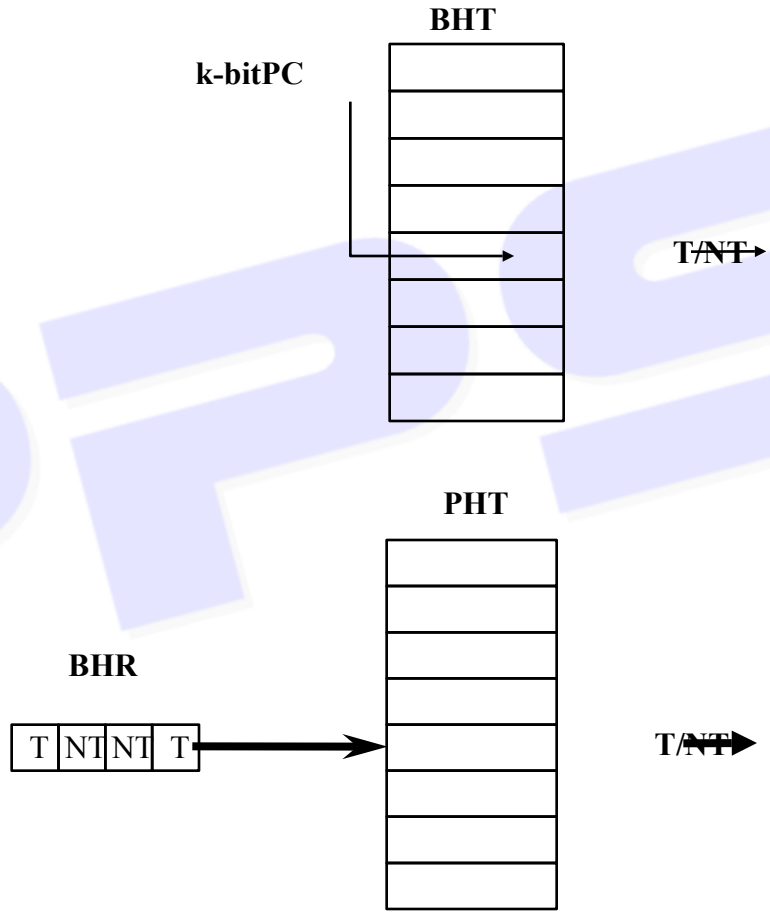


Two-Level Adaptive Branch Predictors

- 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
- The second level history is recorded in one (or more) tables called **Pattern History Table (PHT)** of two-bit saturating counters
- 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.

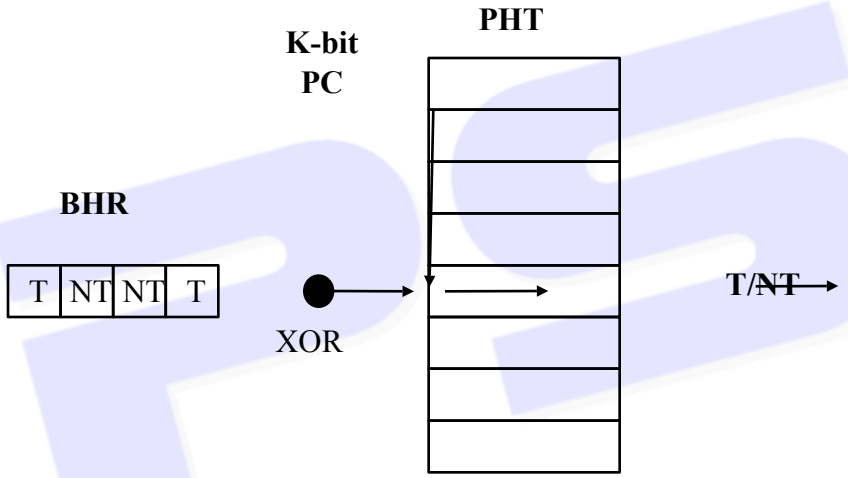
GA Predictor

- **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)



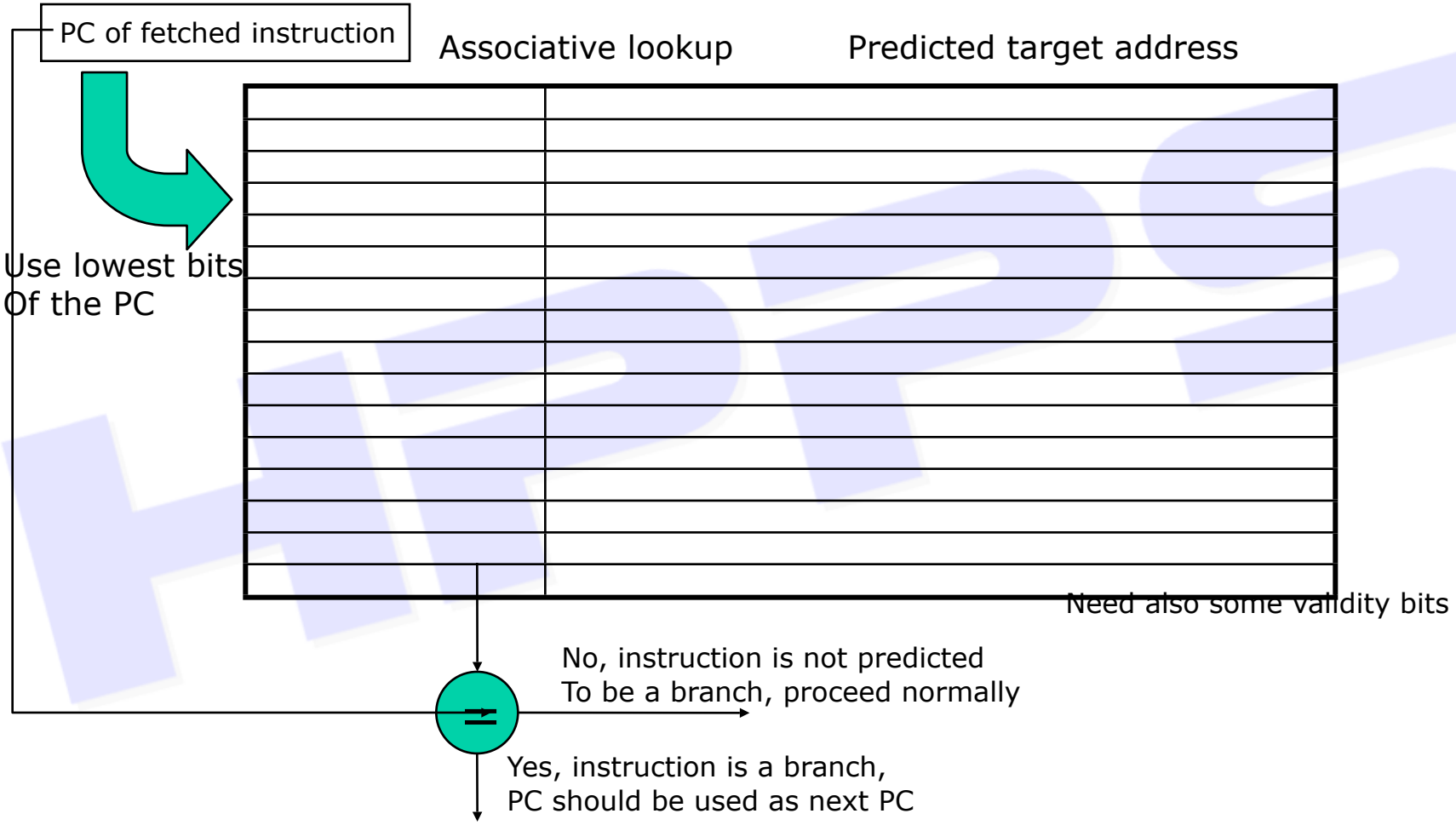
Branch Target Buffer

- **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

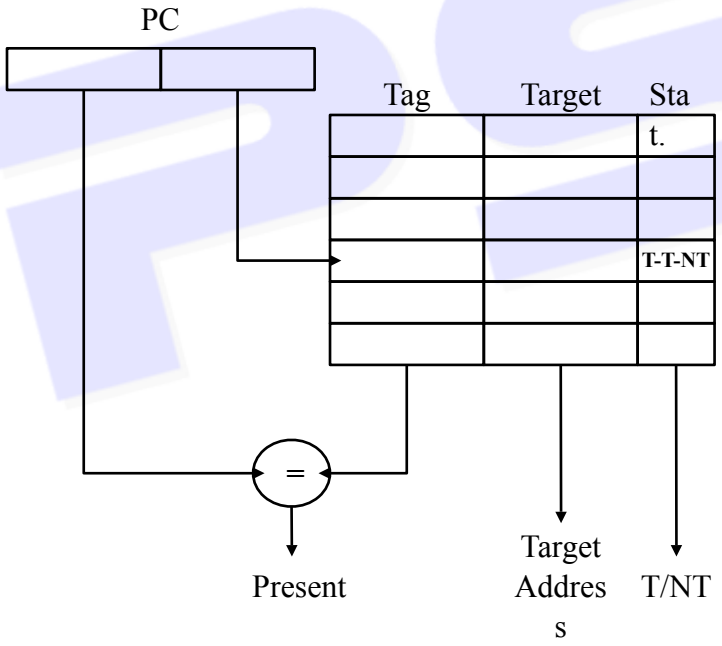
- The predicted target address is expressed as PC-relative

Structure of a Branch Target Buffer



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 target address (expressed as PC-relative for conditional branches) + Prediction state bits as in a Branch Outcome Predictor (optional).



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.

Questions