Course on: “Advanced Computer Architectures”

Branch Prediction Techniques

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

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)

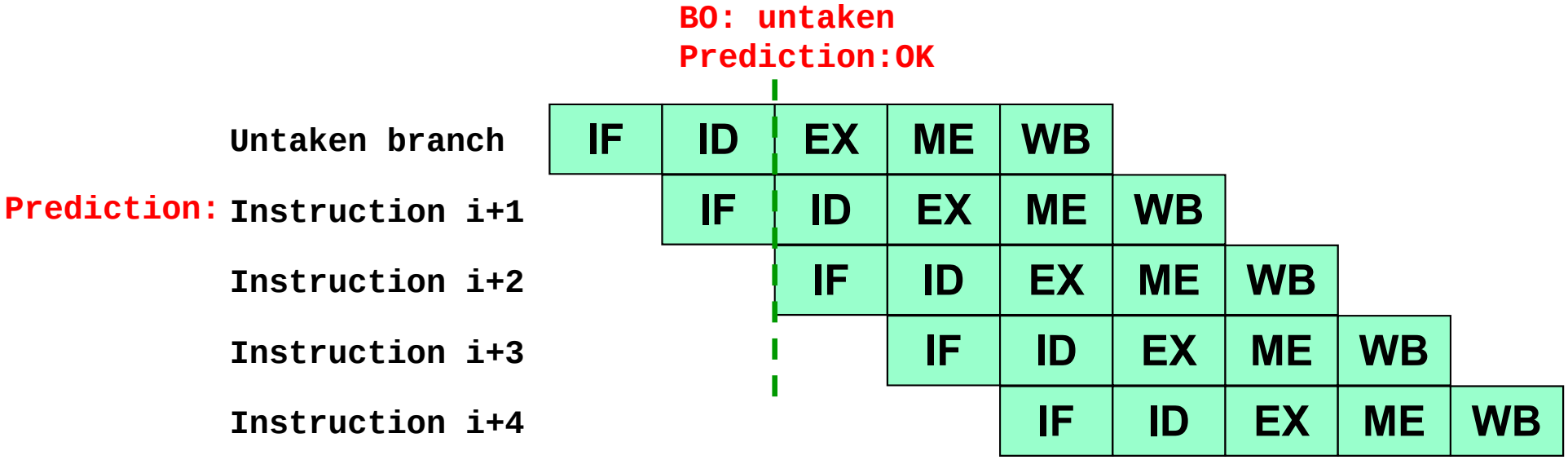
3) Backward Taken Forward Not Taken (BTFNT)

4) Profile-Driven Prediction

5) Delayed Branch

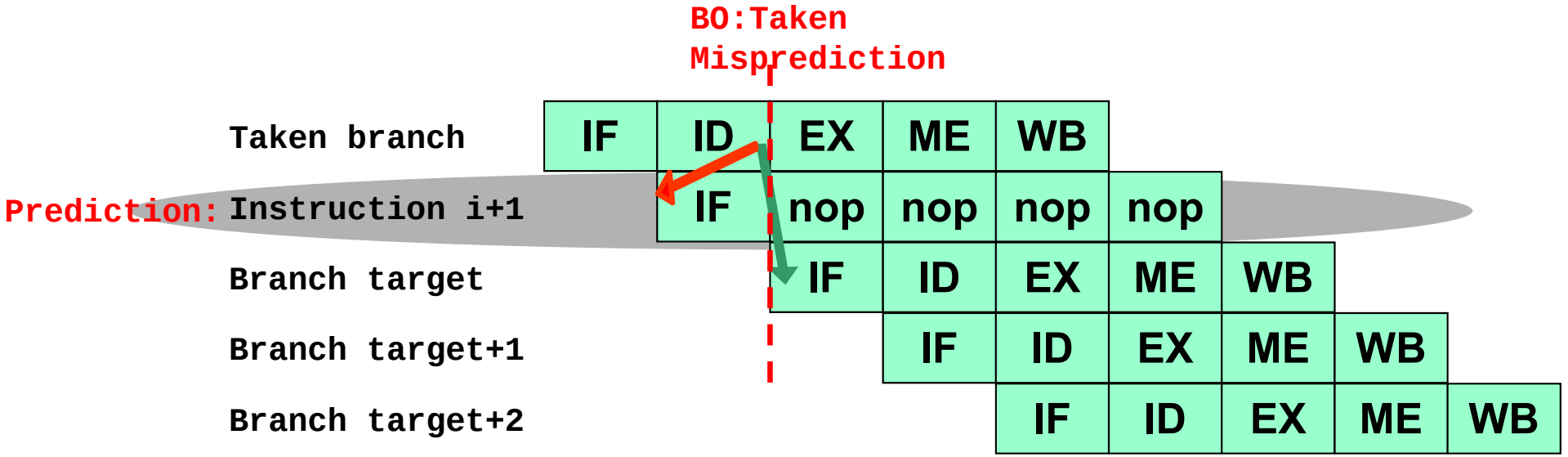
1) Branch Always Not Taken

- 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
    ⇒ **One-cycle performance penalty**



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

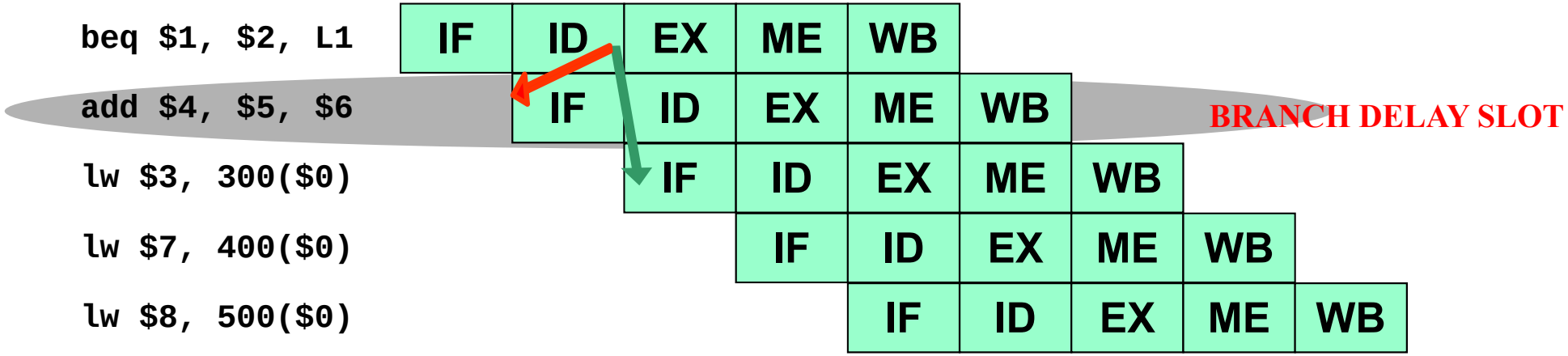
- Let us assume we can profile the behavior of a target application program by executing several runs with different data sets
- The branch prediction is based on profiling information collected from earlier runs.
- The profile-driven prediction method can use compiler hints associated to each branch.

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

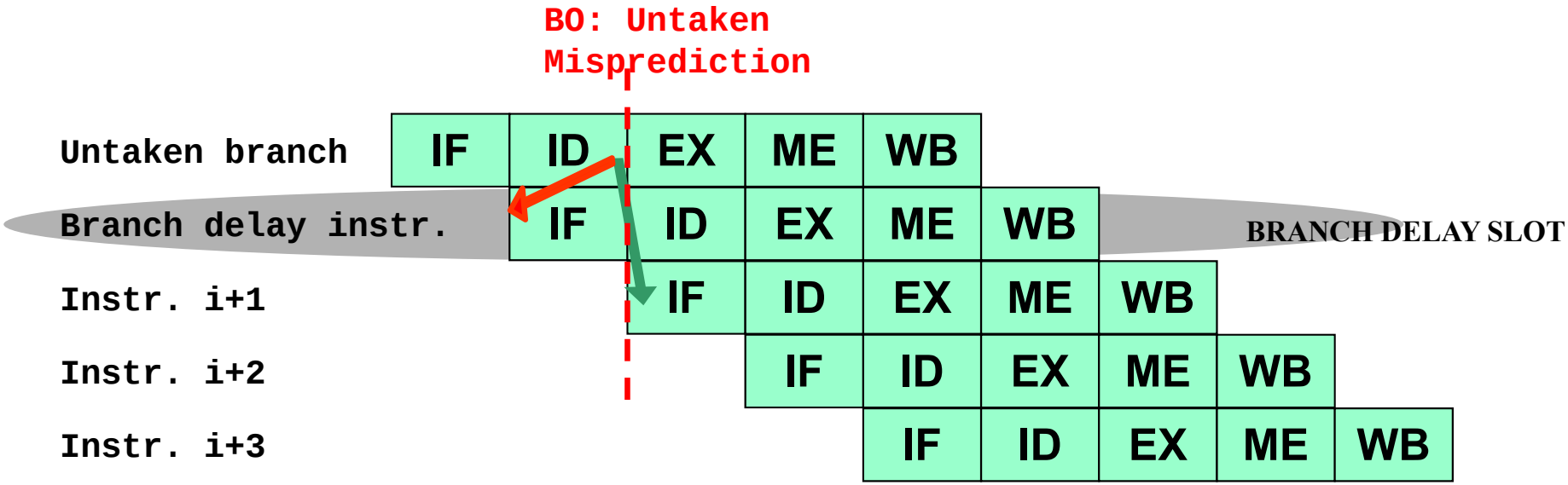
5) Delayed Branch Technique

- 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**



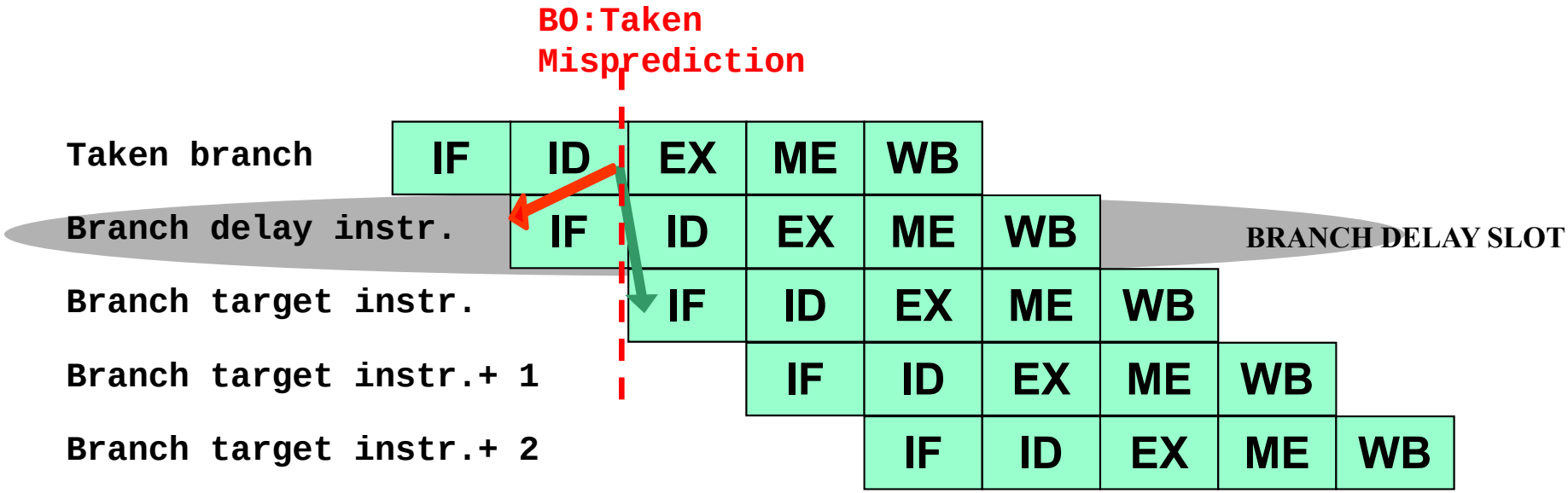
5) Delayed Branch Technique

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



5) Delayed Branch Technique

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

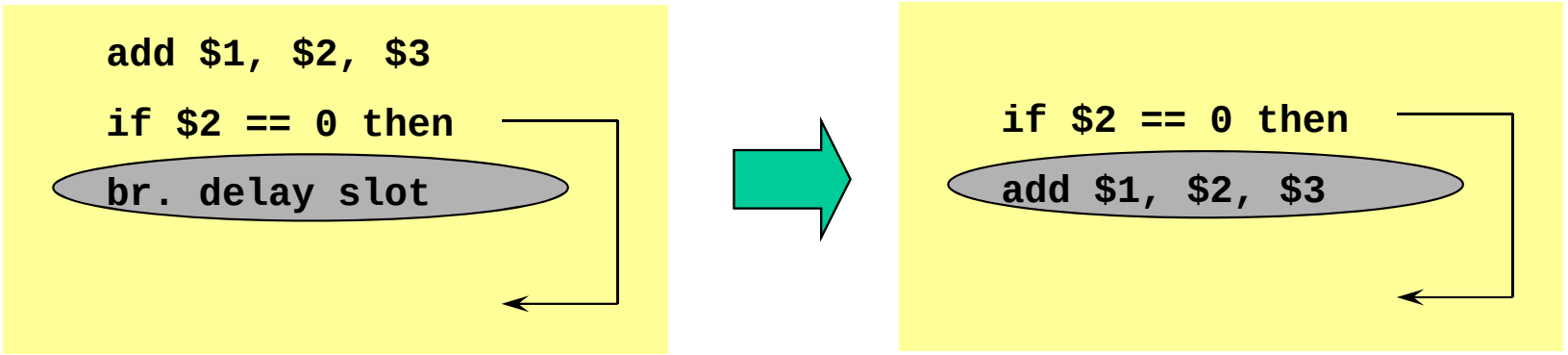


5) Delayed Branch Technique

- 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**

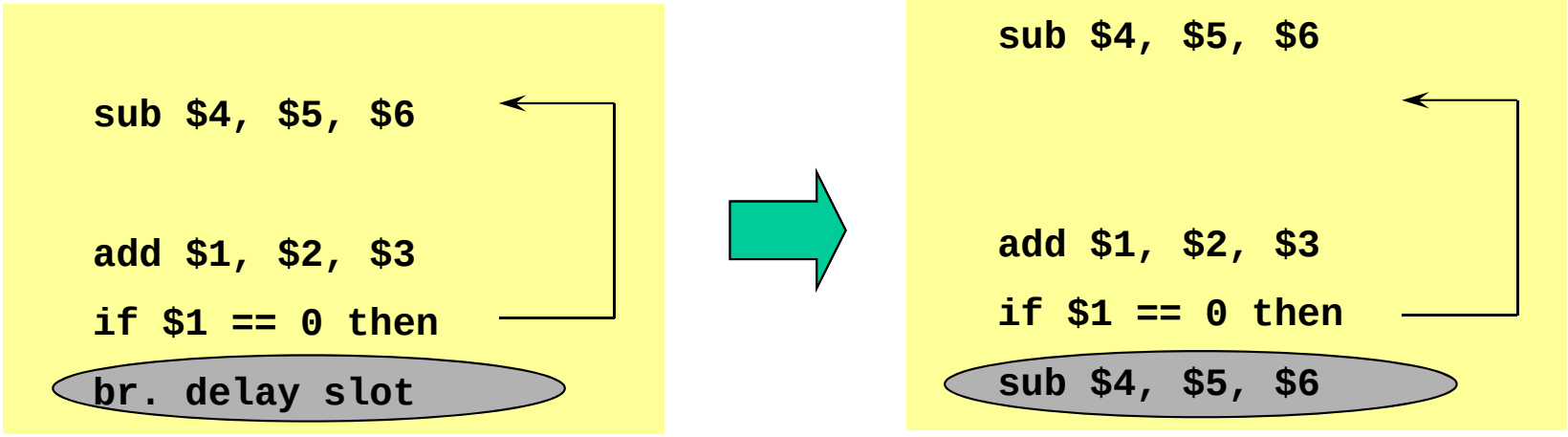
5) Delayed Branch Technique: From Before

- 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.



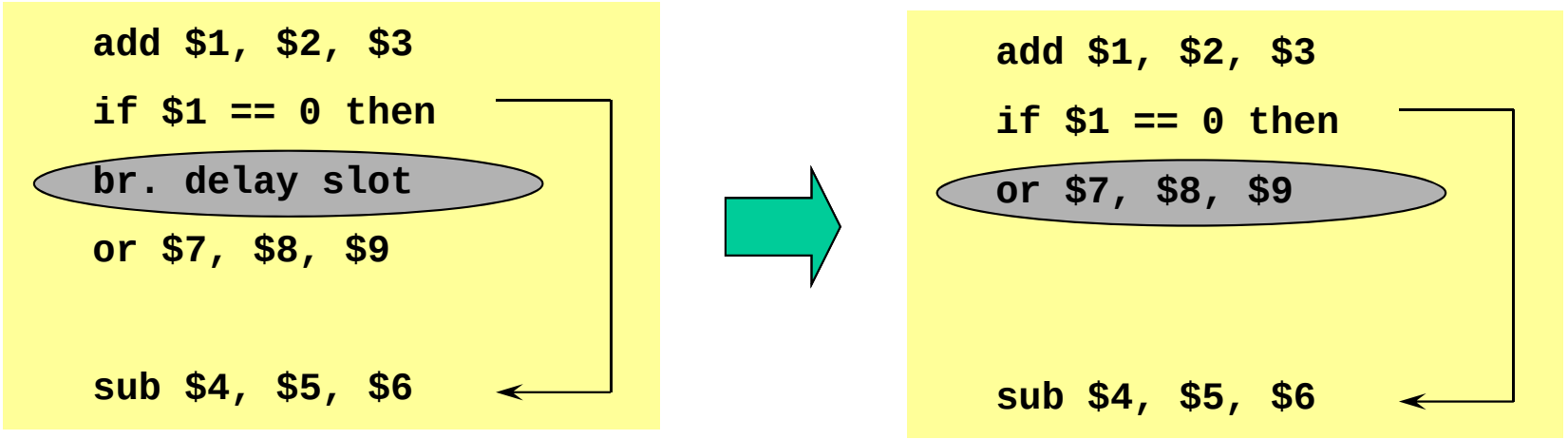
5) Delayed Branch Technique: From Target

- 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.



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.



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.