ECE 341

Lecture # 14

Instructor: Zeshan Chishti
zeshan@ece.pdx.edu

November 17, 2014

Portland State University

Lecture Topics

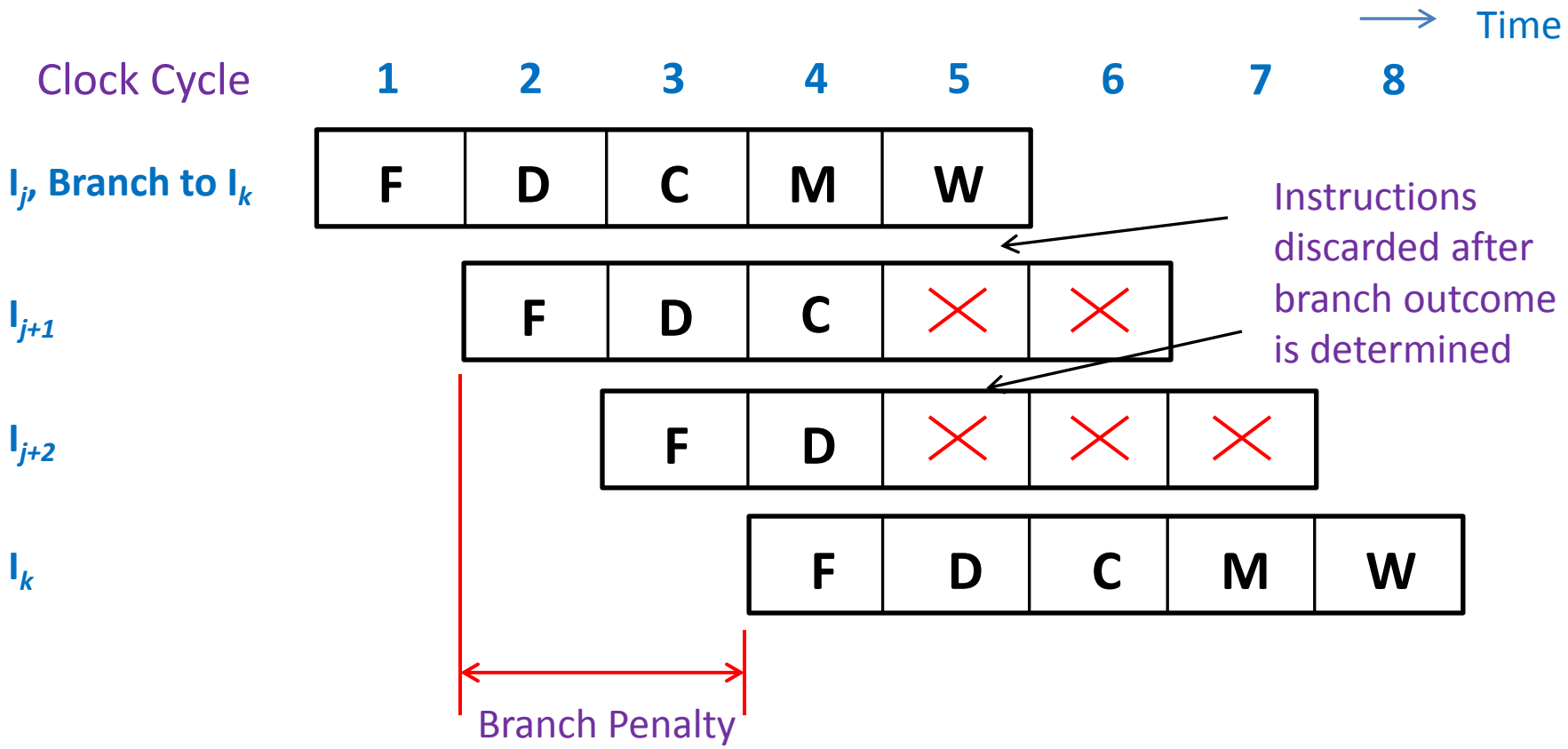
- Pipelining
  - Instruction Hazards
    - Branch penalty
    - Branch delay slot optimization
    - Branch prediction
      - Static branch prediction
      - Dynamic branch prediction
      - Branch target buffer (BTB)

- Reference:
  - Chapter 6: Section 6.6 (Pages 202 - 209 of textbook)

Instruction Hazards

- Instruction hazards (also called control hazards) are caused by a delay in the availability of an instruction or the memory address needed to fetch the instruction
- In ideal pipelined execution, a new instruction is fetched every cycle, while the previous instruction is being decoded
- This will work fine as long as the instruction addresses follow a pre-determined sequence (e.g., PC ← [PC]+4)
- However, branch instructions can alter this sequence
- Branch instructions first need to be executed to determine *whether* and *where* to branch
- Pipeline needs to be stalled before the branch outcome is decided

Unconditional Branches

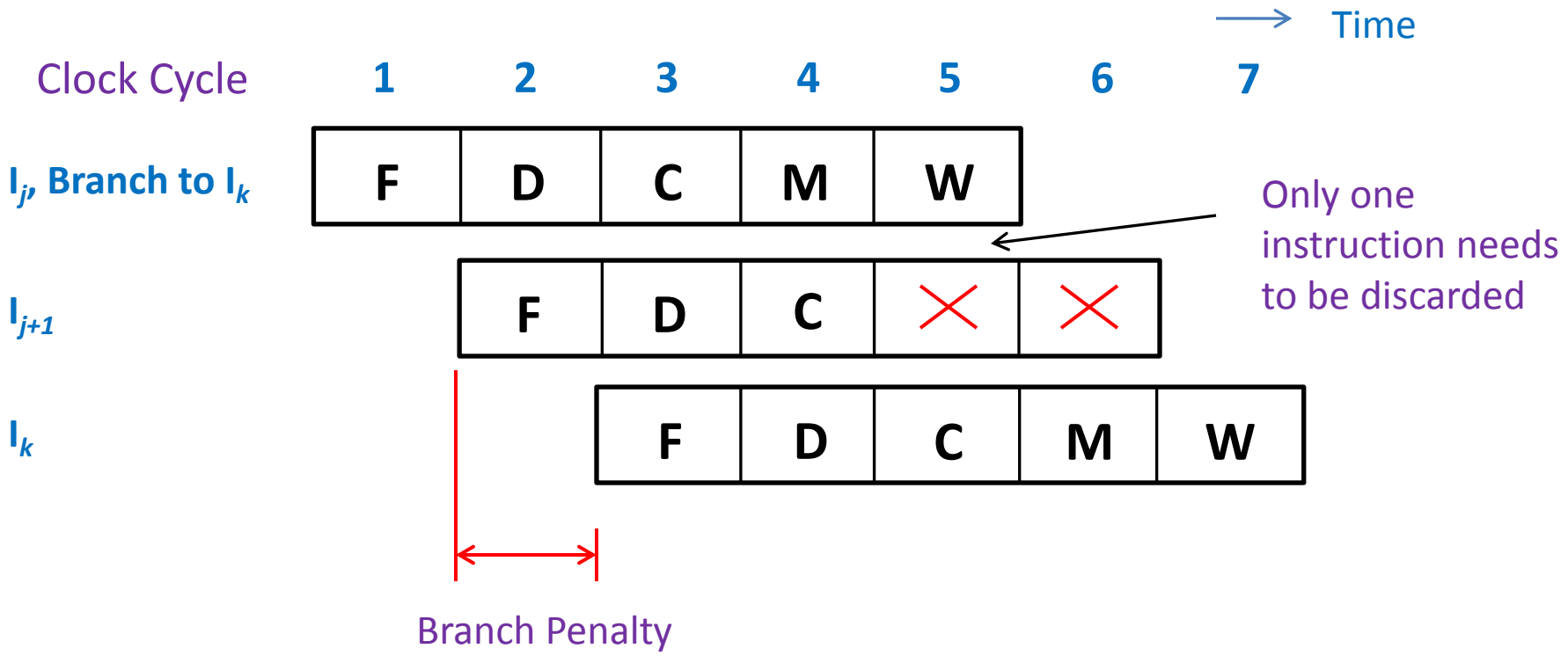


- Branch target computed in cycle-3. Two instructions on the wrong path (I_{j+1} and I_{j+2}) have already been fetched in cycles 2 and 3 and need to be discarded
- Branch Penalty in this case is 2 cycles

Reducing Branch Penalty

- Branch instructions occur frequently (~ 20% of dynamic instruction count in a typical program), 2-cycle branch penalty can increase execution time by 40%
- How to reduce branch penalty for unconditional branches?
- Compute branch target address earlier in the pipeline
  - Include an adder in the “decode” stage to compute target address
  - When the decoder determines that the instruction is an unconditional branch, the computed target address is available before the end of the decode stage

Reducing Branch Penalty (cont.)



- Branch target address computed in cycle # 2
- Branch penalty reduced to one cycle
- Only one instruction (I_{j+1}) is fetched incorrectly and needs to be discarded

Conditional Branches

- Consider a conditional branch instruction such as

  Branch_if_[R5]=[R6] LOOP

- Testing the branch condition (comparison between [R5] and [R6]) determines whether the branch is *taken* or not *taken*
- Recall from the datapath description that this comparison is done in compute stage (cycle # 3 of instruction processing)
- If the branch is taken, the penalty will be 2 cycles

- How to reduce the branch penalty for conditional branches?
- Test the branch condition in the "Decode" stage
  - Move the comparator from "Compute" stage to "Decode" stage
  - Branch condition tested in parallel with target address computation
  - Branch penalty reduced to one cycle

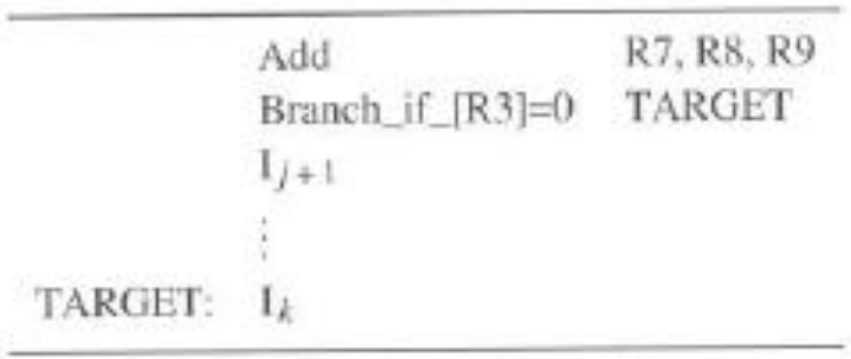
Branch Delay Slot

- Moving the branch decision and effective address calculation to the "Decode" stage reduces the branch penalty from two to one cycle
- Are there ways to reduce the branch penalty to zero?
- Yes. The basic idea behind Branch delay slot is to find an instruction that appears before the branch in program order but the branch outcome is independent of that instruction
- **If** the compiler finds such an instruction, it places this instruction in the branch delay slot (the location immediately after the branch instruction)
- This instruction is always executed irrespective of whether the branch is taken or not taken => no discarding and no branch penalty

Branch Delay Slot Example

Without the delay slot optimization:

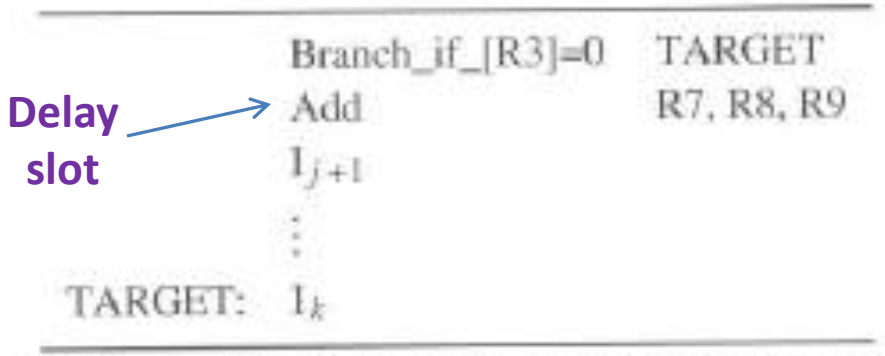
- If the branch condition is evaluated to be true ([R3] == 0), the branch is *taken* and instruction I_{j+1} needs to be discarded => penalty = 1 cycle
- If the branch condition is false ([R3] ! = 0), there is no penalty



(a) Original sequence of instructions containing a conditional branch instruction

After the delay slot optimization:

- The "Add" instruction is always executed, even if the branch is taken
- Instruction I_{j+1} is fetched only if the branch is *not taken*
- Branch penalty if zero irrespective of the branch outcome



(b) Placing the Add instruction in the branch delay slot where it is always executed

Branch Delay Slot Limitations

- The delay slot optimization must preserve all the data dependences
- In the previous example, it was safe to move the “Add” instruction to the branch delay slot, because the branch outcome was independent of that particular “Add” instruction
- But because of data dependences, it is not always possible to find a suitable instruction to fill the delay slot
- If no useful instruction can be placed in the delay slot, the compiler places a NOP (no operation) instruction in that slot
  - In this case, branch penalty = 1, irrespective of the branch outcome
- Effectiveness of delay slot optimization depends on the ability of compiler to find a useful delay slot candidate (~ 70% of the cases)
- Other ways to reduce branch penalty to zero?

Branch Prediction

- The decision about control flow (where to fetch the next instruction from?) is made in the *fetch* stage
- The branch penalty is non-zero because when the processor computes the branch outcome (in *decode* stage), a useless instruction may have already been fetched and needs to be discarded
- To prevent the fetching of useless instruction, the processor needs to know about the branch outcome in the *fetch* stage
- This involves the following two steps:
  - Anticipating that the instruction being fetched is a branch instruction
  - Predicting whether the branch instruction will be taken or not taken
  - Predicting the branch target address (for a taken branch)

Static Branch Prediction

- In static branch prediction, the prediction made for a conditional branch remains constant (static) throughout the execution of a program
- Example 1: Always-predict-not-taken
  - Simplest form of prediction, always fetch next instruction in the sequential order
  - In case of a misprediction, the incorrectly fetched instruction is discarded and branch penalty is incurred
  - Low prediction accuracy because many branches in the program are *taken*
- Typically, branch outcomes are *not* completely random
- In a loop with many iterations, forward branches (beginning of loop) are mostly *not taken* and backward branches (end of loop) are mostly *taken*
- Example 2: Predict *not-taken* for forward branches and *taken* for backward branches
  - Improves prediction accuracy as compared to the always-not-taken prediction
  - Mispredictions still happen during the last loop iteration

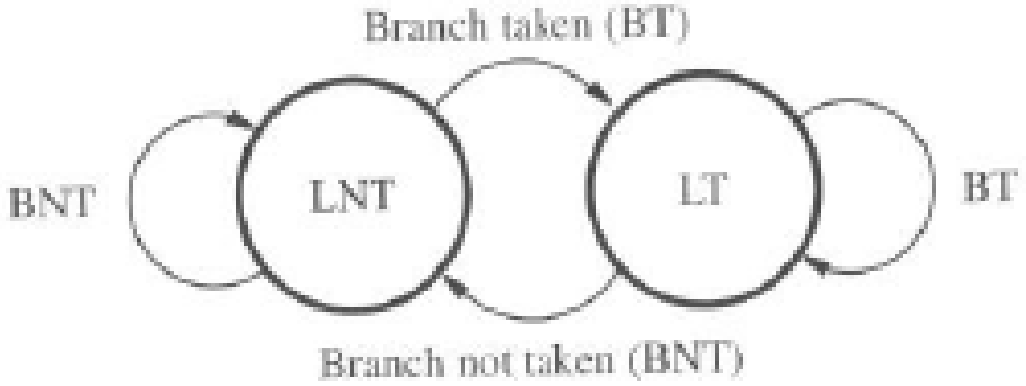
Dynamic Branch Prediction

- Outcomes for a branch instruction often change during program execution
  - Static prediction may result in high misprediction accuracy
- But, outcomes for a particular branch often follow a predictable pattern

- Key idea behind dynamic branch prediction:
  - Track the past outcomes for a branch instruction to make predictions about future outcomes

- In its simplest form, a dynamic prediction algorithm can use the result of the most recent execution of a branch instruction
  - This result can be captured in a single bit (e.g., “0” if the branch was taken and “1” if the branch was not taken)
  - The processor assumes that the next time, the branch instruction is executed, its outcome is the same as the last time

1-bit Branch Prediction



- The algorithm is implemented by a 2-state state machine:

  LT -- Branch is likely to be taken

  LNT -- Branch is likely not to be taken

- The prediction for a branch is based on the *current state* of the state machine
- The state transitions are based on the actual outcome computed after the branch has been executed

Example

- Consider a branch instruction which is executed 6 times in a program. The actual outcomes of the branch are T, T, NT, T, T, NT where “T” = *Taken* and “N” = *Not taken*. Assume that the 1-bit branch predictor starts in the LNT state. What predictions will it make for each instance of the branch?

Example (cont.)

- Consider a branch instruction which is executed 6 times in a program. The actual outcomes of the branch are T, T, NT, T, T, NT where "T" = *Taken* and "N" = *Not taken*. Assume that the 1-bit branch predictor starts in the LNT state. What predictions will it make for each instance of the branch?

Instance	Current State	Prediction	Actual Outcome	Next State
1	LNT	NT	T	LT
2				
3				
4				
5				
6				

Example (cont.)

- Consider a branch instruction which is executed 6 times in a program. The actual outcomes of the branch are T, T, NT, T, T, NT where "T" = *Taken* and "N" = *Not taken*. Assume that the 1-bit branch predictor starts in the LNT state. What predictions will it make for each instance of the branch?

Instance	Current State	Prediction	Actual Outcome	Next State
1	LNT	NT	T	LT
2	LT	T	T	LT
3				
4				
5				
6				

Example (cont.)

- Consider a branch instruction which is executed 6 times in a program. The actual outcomes of the branch are T, T, NT, T, T, NT where "T" = *Taken* and "N" = *Not taken*. Assume that the 1-bit branch predictor starts in the LNT state. What predictions will it make for each instance of the branch?

Instance	Current State	Prediction	Actual Outcome	Next State
1	LNT	NT	T	LT
2	LT	T	T	LT
3	LT	T	NT	LNT
4	LNT	NT	T	LT
5	LT	T	T	LT
6	LT	T	NT	LNT

Example (cont.)

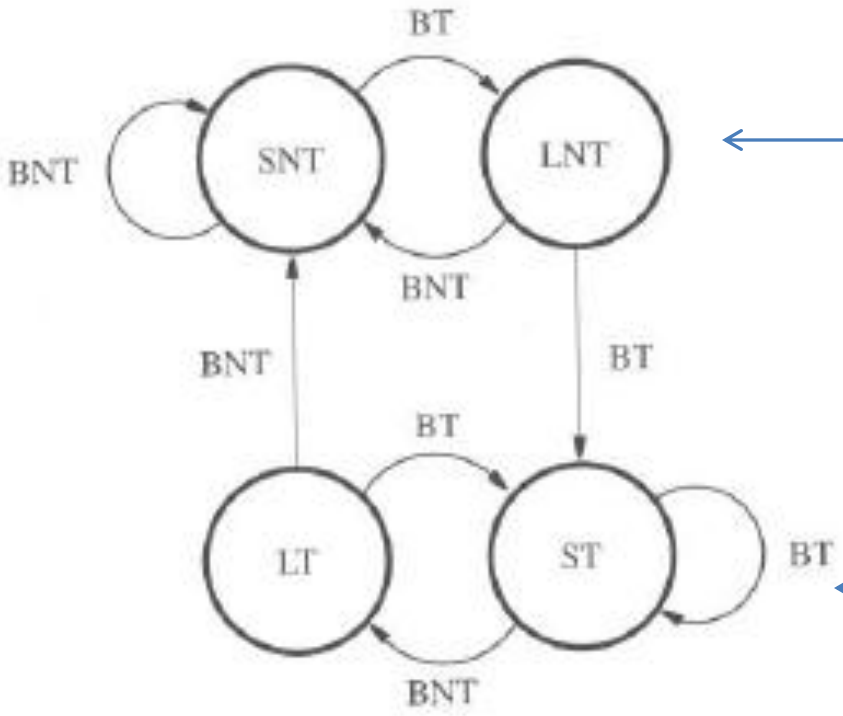
- Consider a branch instruction which is executed 6 times in a program. The actual outcomes of the branch are T, T, NT, T, T, NT where "T" = *Taken* and "N" = *Not taken*. Assume that the 1-bit branch predictor starts in the LNT state. What predictions will it make for each instance of the branch?

Instance	Current State	Prediction	Actual Outcome	Next State
1	LNT	NT	T	LT
2	LT	T	T	LT
3	LT	T	NT	LNT
4	LNT	NT	T	LT
5	LT	T	T	LT
6	LT	T	NT	LNT

Prediction Accuracy = 2/6

Mispredictions happen during both the first and last iterations of the loop => one bit of state not enough to capture the branch outcome pattern accurately

2-bit Branch Prediction



Branch predicted as ***Not taken*** in these two states

Branch predicted as ***Taken*** in these two states

ST: Strongly likely to be taken
LT: Likely to be taken
LNT: Likely not to be taken
SNT: Strongly likely not to be taken

Example

- Consider a branch instruction which is executed 6 times in a program. The actual outcomes of the branch are T, T, NT, T, T, NT where "T" = *Taken* and "N" = *Not taken*. Assume that the 2-bit branch predictor starts in the LT state. What predictions will it make for each instance of the branch?

Instance	Current State	Prediction	Actual Outcome	Next State
1	LT	T	T	ST
2	ST	T	T	ST
3	ST	T	NT	LT
4	LT	T	T	ST
5	ST	T	T	ST
6	ST	T	NT	LT

Example (cont.)

- Consider a branch instruction which is executed 6 times in a program. The actual outcomes of the branch are T, T, NT, T, T, NT where "T" = *Taken* and "N" = *Not taken*. Assume that the 2-bit branch predictor starts in the LT state. What predictions will it make for each instance of the branch?

Instance	Current State	Prediction	Actual Outcome	Next State
1	LT	T	T	ST
2	ST	T	T	ST
3	ST	T	NT	LT
4	LT	T	T	ST
5	ST	T	T	ST
6	ST	T	NT	LT

Prediction Accuracy = 4/6

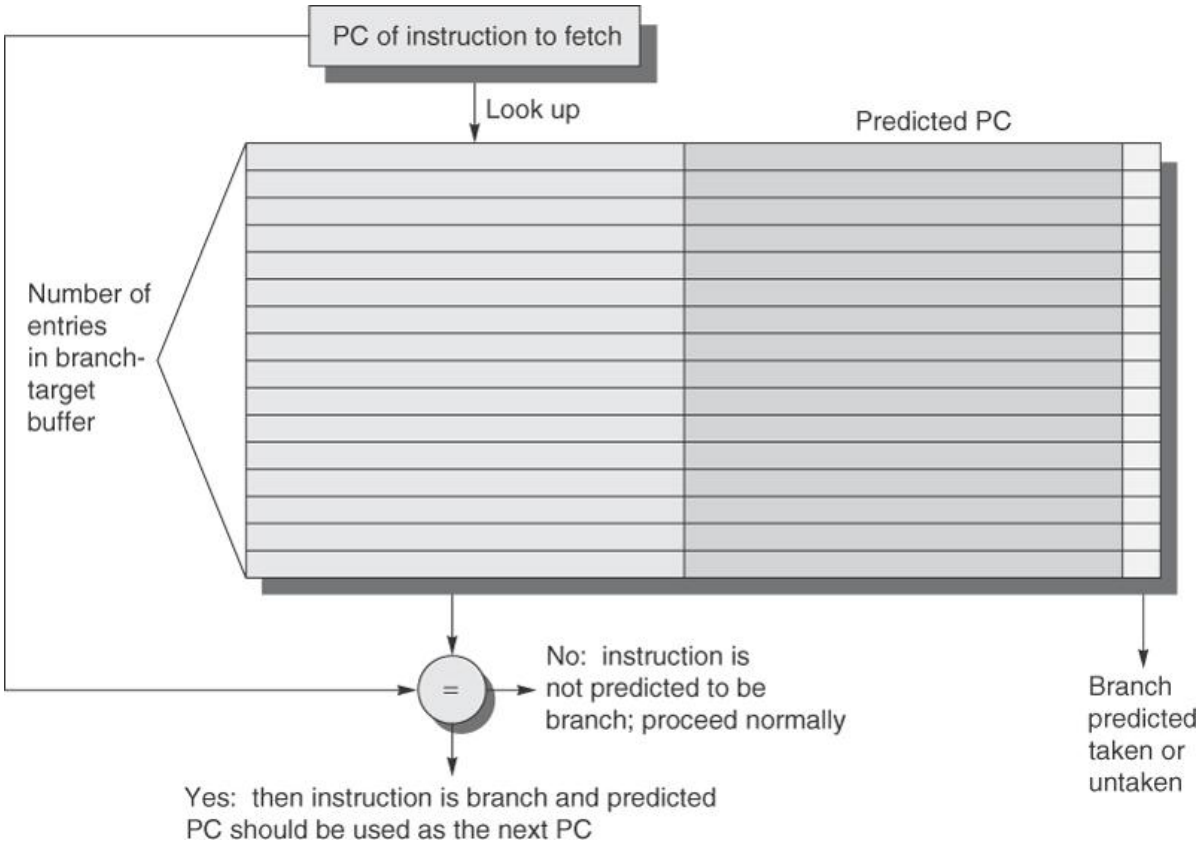
Mispredictions happen only during the last iteration of the loop => less mispredictions that 1-bit prediction

Predicting Branch Targets

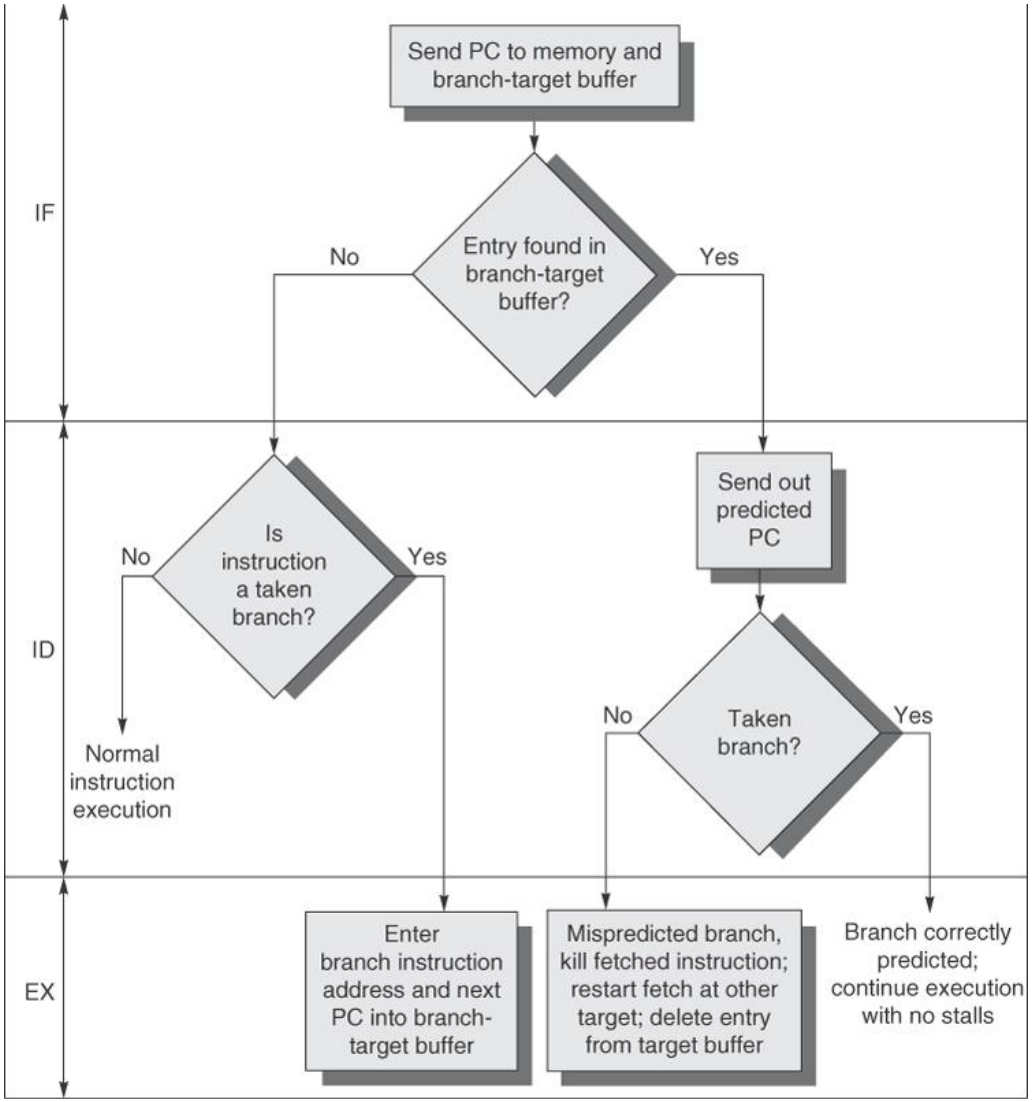
- To avoid branch penalty in 5-stage pipeline, we need to know which address to fetch next instruction from before end of IF stage
- Requires us to know whether the (as-yet undecoded) instruction is a branch and, if so, what the next PC should be
- Solution: Predict the target address for a potential branch during the IF stage

Branch Target Buffer (BTB)

- During IF stage, use PC of current instruction (possible branch) to index into table of predicted target PCs for that branch
- Fetch of predicted target begins at the start of next cycle



Branch target Buffer Behavior



Branch Penalty

Instruction in Buffer	Prediction	Actual Branch	Penalty Cycles
Yes	Taken	Taken	0
Yes	Taken	Not Taken	2
No		Taken	2
No		Not Taken	0