# Branch Prediction (3.4, 3.5)

Instructor: Prof. Yan Luo

# Why do we want to predict branches?

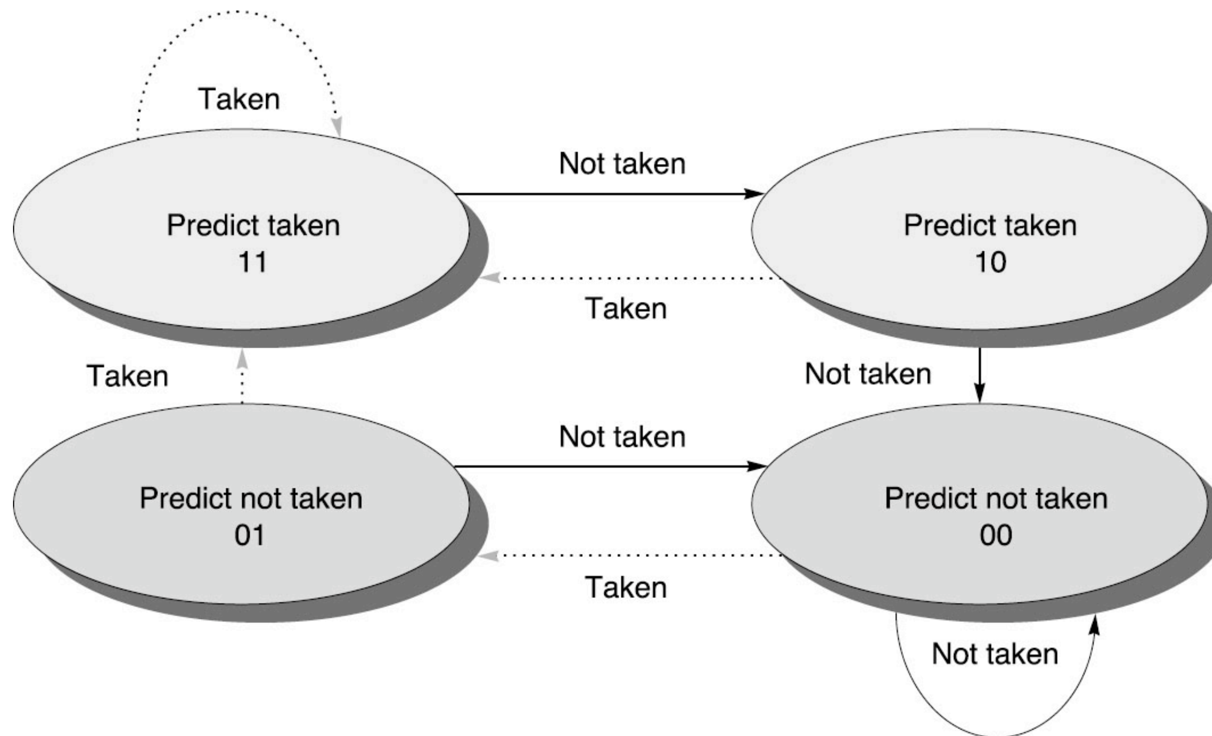
- MIPS based pipeline – 1 instruction issued per cycle, branch hazard of 1 cycle.
  - Delayed branch
- Modern processor and next generation – multiple instructions issued per cycle, more branch hazard cycles will incur.
  - Cost of branch misfetch goes up
  - Pentium Pro – 3 instructions issued per cycle, 12+ cycle misfetch penalty
    - HUGE penalty for a misfetched path following a branch

# Branch Prediction

- Easiest (static prediction)
  - Always taken, always not taken
  - Opcode based
  - Displacement based (forward not taken, backward taken)
  - Compiler directed (branch likely, branch not likely)
- Next easiest
  - **1 bit predictor** – remember last taken/not taken per branch
    - Use a *branch-prediction buffer* or *branch-history table*
    - Use part of the PC (low-order bits) to index buffer/table
      - Multiple branches may share the same bit
    - Invert the bit if the prediction is wrong
    - Backward branches for loops will be mispredicted twice

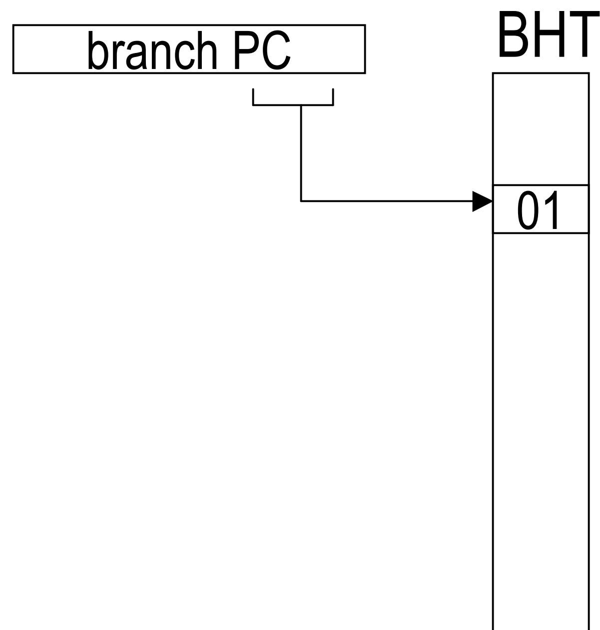
# 2-bit Branch Prediction

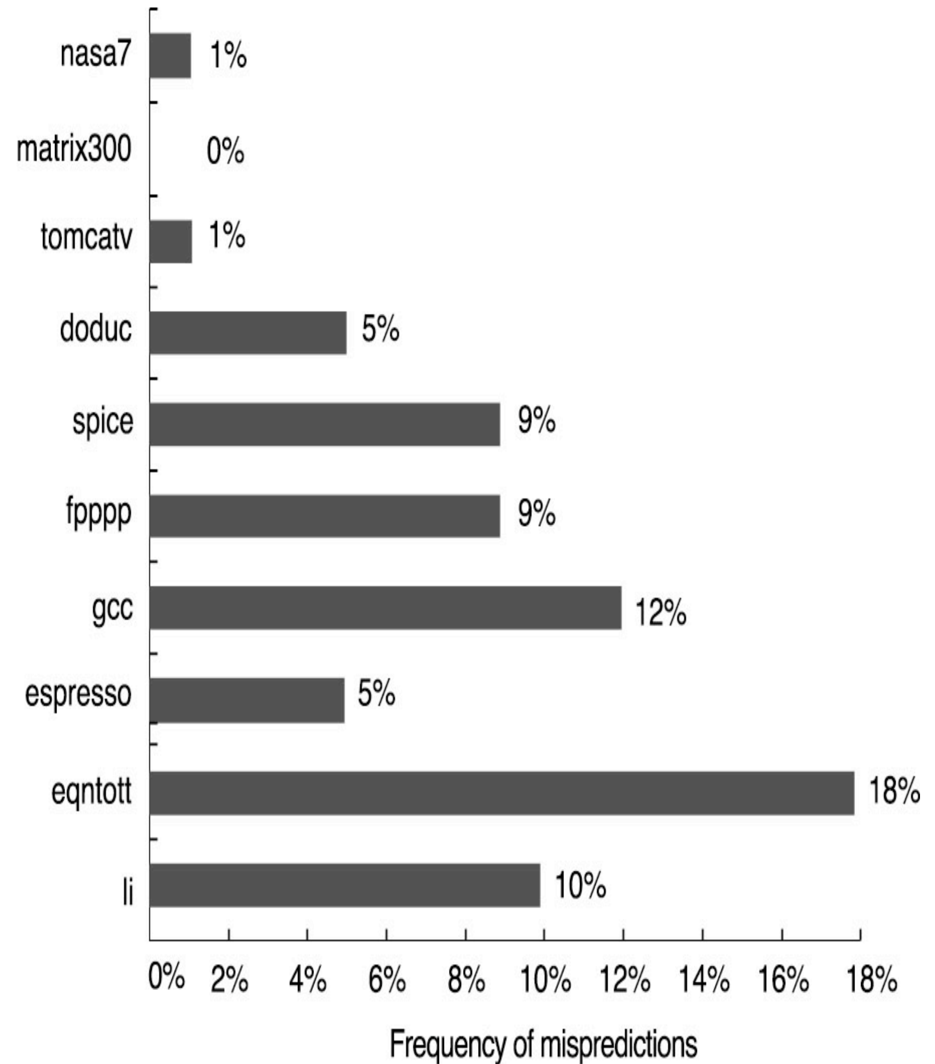
- Has 4 states instead of 2, allowing for more information about tendencies
- A prediction must miss twice before it is changed
- Good for backward branches of loops



# Branch History Table

- Has limited size
- 2 bits by N (e.g. 4K)
- Uses low-order bits of branch PC to choose entry

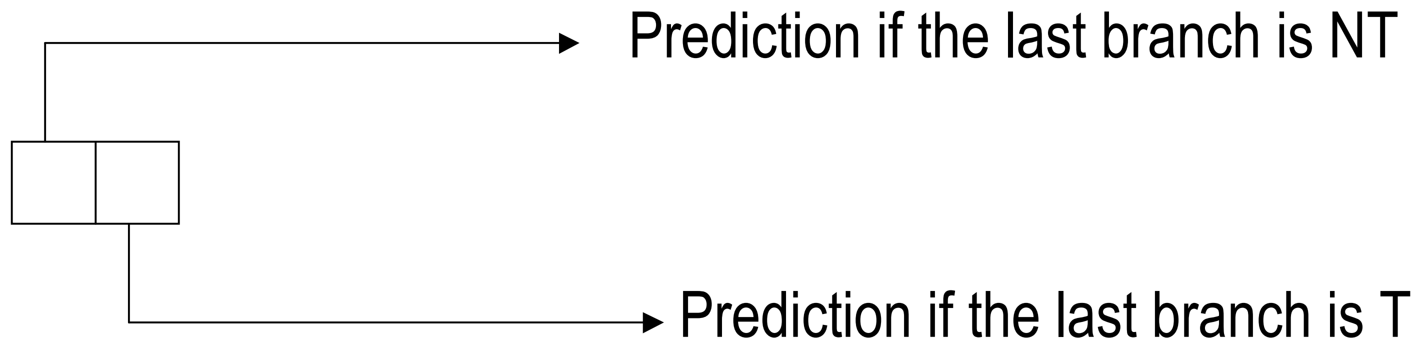


# Correlating or Two-level Predictors

- Correlating branch predictors also look at other branches for clues

```
if (aa==2)  -- branch b1
        aa = 0;
if (bb==2)  ---   branch b2
        bb = 0;
if(aa!=bb) { …   --- branch b3 – Clearly depends on the results of b1 and b2
```




(1,1) predictor – uses history of 1 branch and uses a 1-bit predictor

# Another Example

```
If (d==0)
        d=1;
If (d=1) ---
```

Code Sequence assuming d is assigned to R1:

```
        BNEZ R1, L1         ; branch b1 (d!=0)
        DADDU R1,R0,#1      ; d==0, so d=1
L1:     DADDIU R3,R1,#-1
        BNEZ  R3,L2         ; branch b2  (d!=1)
```

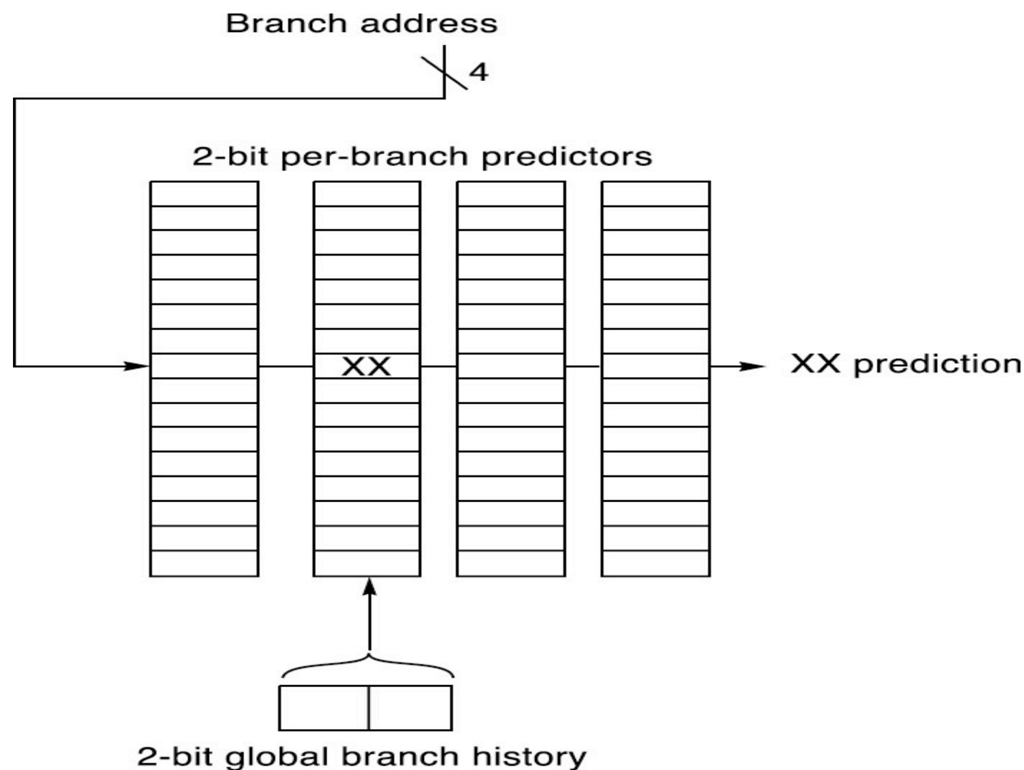
Possible Execution Sequence for the code fragment;

| **Initial d** | **d==0?** | **B1** | **d before b2** | **d==1?** | **b2** |
|---|---|---|---|---|---|
| 0 | yes | not taken | 1 | yes | not taken |
| 1 | No | taken | 1 | yes | not taken |
| 2 | no | taken | 2 | no | taken |

Clearly, if b1 is not taken b2 will not be taken => correlation

# Correlating Branch Predictor

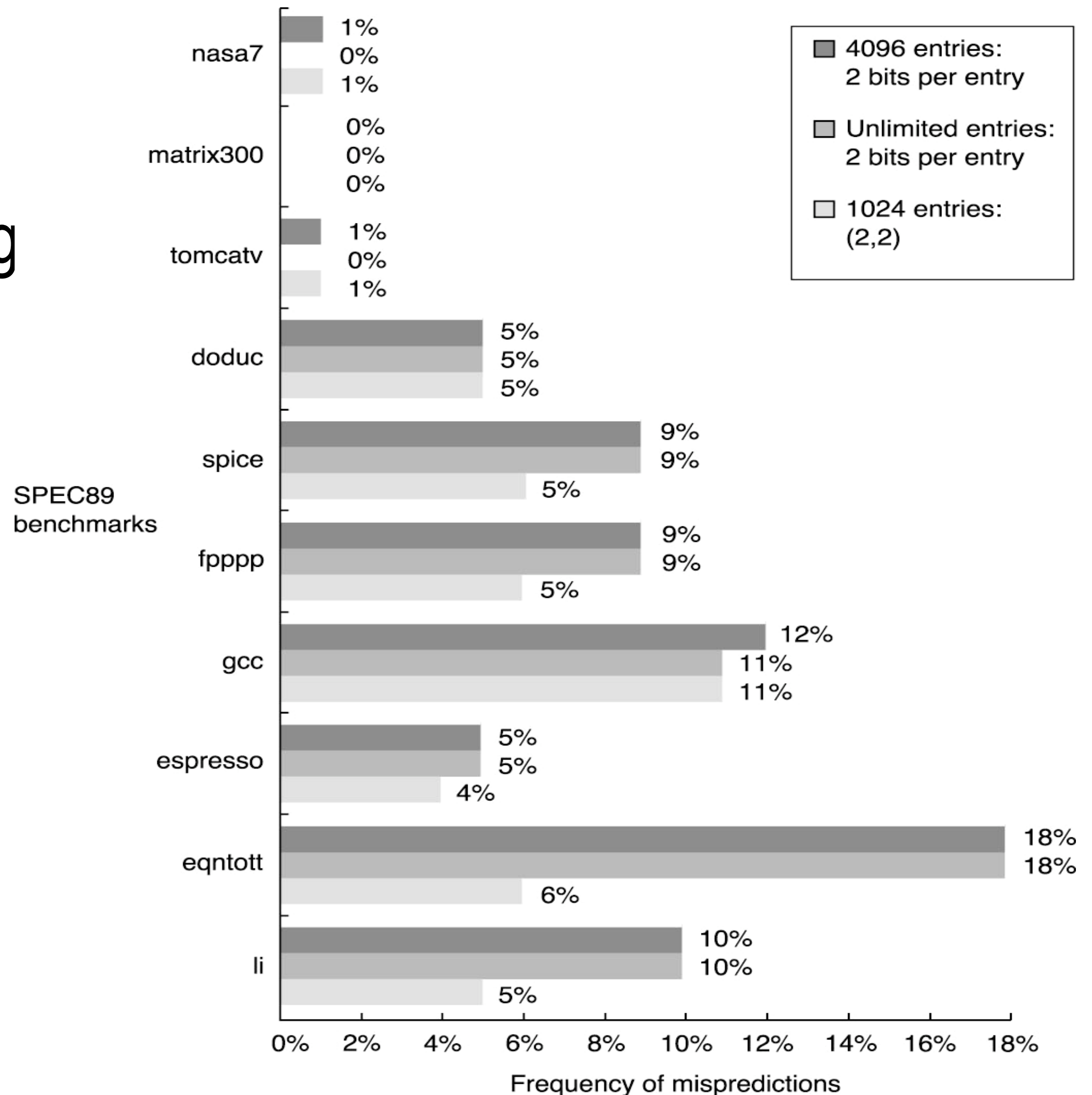
- If we use 2 branches as histories, then there are 4 possibilities (T-T, NT-T, NT-NT, NT-T).
- For each possibility, we need to use a predictor (1-bit, 2-bit).
- And this repeats for every branch.




(2,2) branch prediction

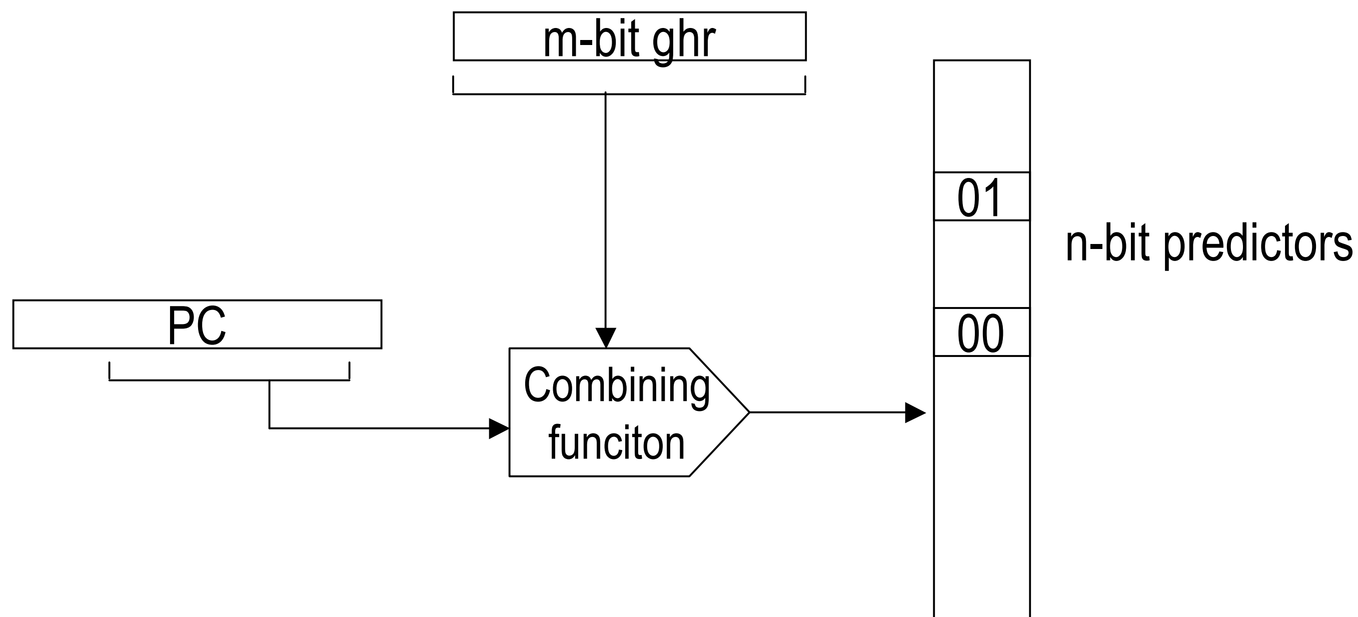
# Performance of Correlating Branch Prediction

- With same number of state bits, (2,2) performs better than noncorrelating 2-bit predictor.
- Outperforms a 2-bit predictor with infinite number of entries



# General (m,n) Branch Predictors

- The *global history register* is an *m-bit* shift register that records the last *m* branches encountered by the processor
- Usually use both the PC address and the GHR (2-level)



# Is Branch Predictor Enough?

- When is using branch prediction beneficial?
  - When the outcome is known later than the target
  - For example, in our standard MIPS pipeline, we compute the target in ID stage but testing the branch condition incur a structure hazard in register file.
- If we predict the branch is taken and suppose it is correct, what is the target address?
  - Need a mechanism to provide target address as well
- Can we eliminate the one cycle delay for the 5-stage pipeline?
  - Need to fetch from branch target immediately after branch

# Branch Target Buffer (BTB)

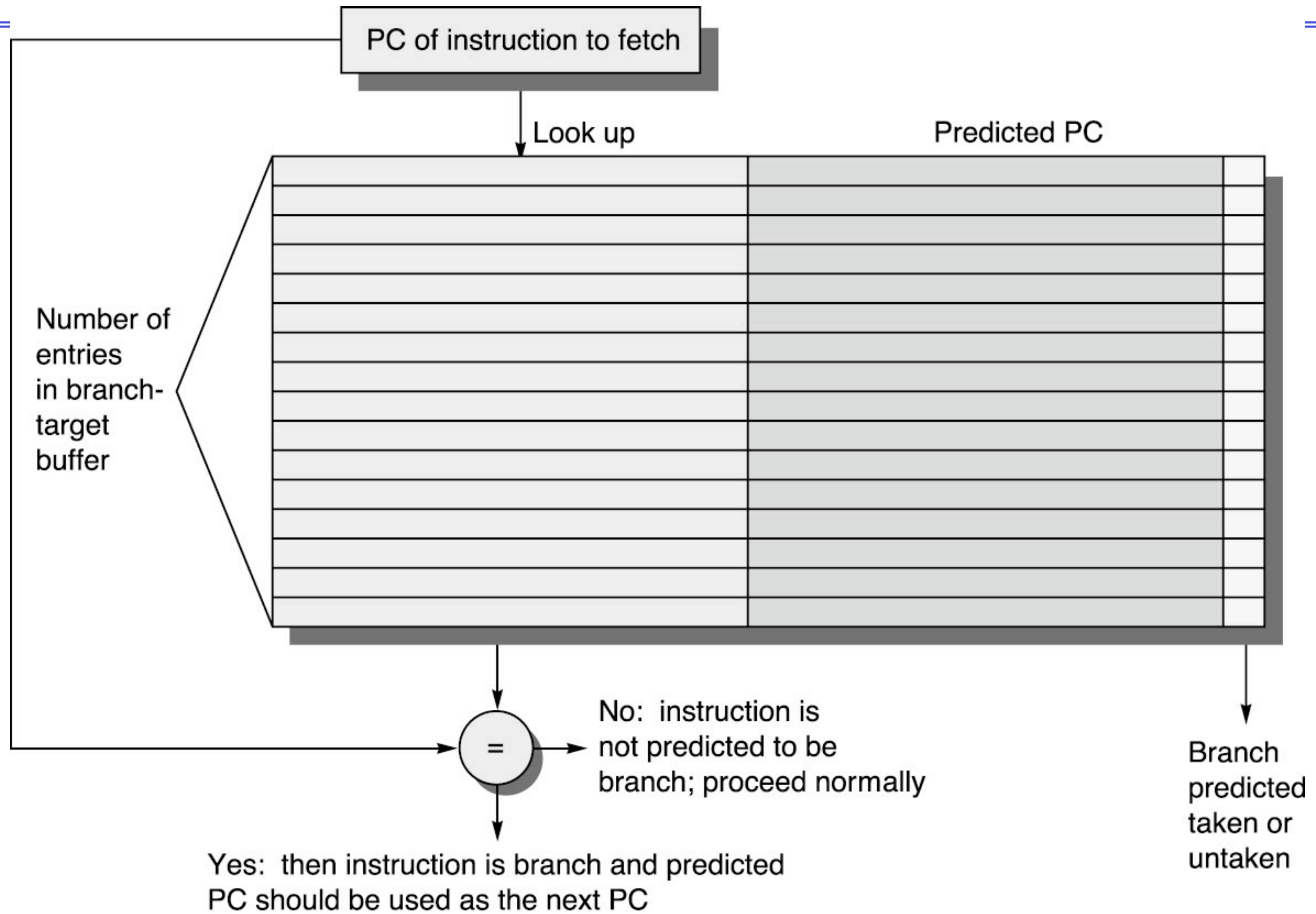
## Is the current instruction a branch ?

- BTB provides the answer before the current instruction is decoded and therefore enables fetching to begin after IF-stage .

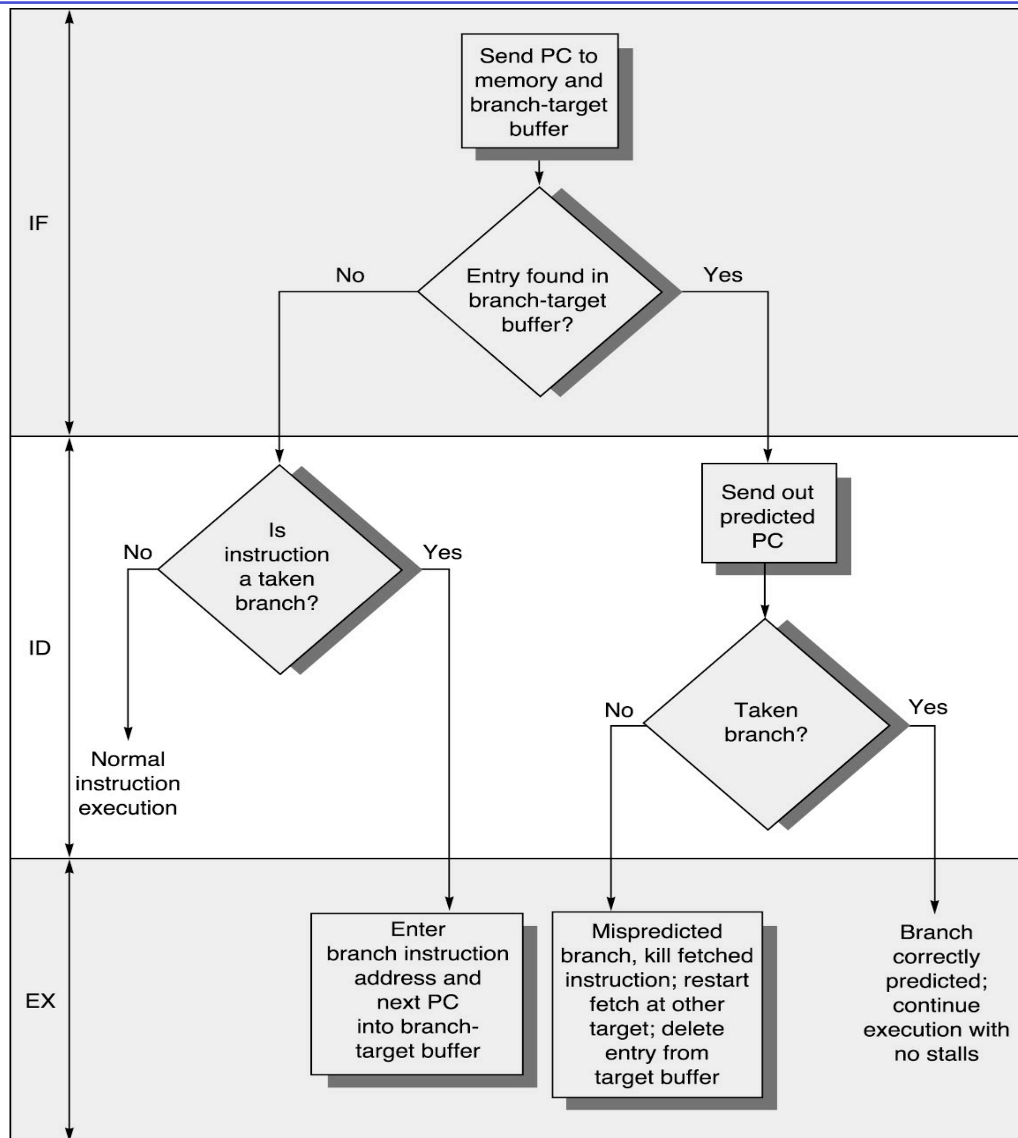
## What is the branch target ?

- BTB provides the branch target if the prediction is a taken direct branch (for not taken branches the target is simply PC+4 ) .

# BTB



# BTB operations




- BTB hit, prediction correct → 0 cycle delay
- BTB hit, misprediction ≥ 2 cycle penalty
- BTB miss, branch ≥ 1 cycle penalty

# BTB Performance

- Two things can go wrong
  - BTB miss (misfetch)
  - Mispredicted a branch (mispredict)
- Suppose for branches, BTB hit rate of 85% and predict accuracy of 90%, misfetch penalty of 2 cycles and mispredict penalty of 5 cycles. What is the average branch penalty?

  2*(15%) + 5*(85%*10%)

  see also the example on Pg. 211
- BTB and BPT can be used together to perform better prediction

# Integrated Instruction Fetch Unit

Separate out IF from the pipeline and integrate with the following components. So, the pipeline consists of Issue, Read, EX, and WB (scoreboarding) ; Or Issue, EX and WB stages (Tomasulo).

1. Integrated Branch Prediction – Branch predictor is part of the IFU.

2. Instruction Prefetch – Fetch instn from IM ahead of PC with the help of branch predictor and store in a prefetch buffer.

3. Instruction Memory Access and Buffering - Keep on filling the Instruction Queue independent of the execution.

# Branch Prediction Summary

- The better we predict, the higher penalty we might incur
- 2-bit predictors capture tendencies well
- Correlating predictors improve accuracy, particularly when combined with 2-bit predictors
- Accurate branch prediction does no good if we don't know there was a branch to predict
- BTB identifies branches in IF stage
- BTB combined with branch prediction table identifies branches to predict, and predicts them well

# Speculation
# Exploring ILP with Multi-Issue
# (3.6)

# How to obtain CPI>1?

- Issue more than one instruction per cycle
- Compiler needs to do a good job in scheduling code (rearranging code sequence) – statically scheduled
- Fetch up to *n* instructions as an *issue packet* if issue width is n
- Check hazards during issue stage (including decode)
  - Issue checks are too complex to perform in one clock cycle
  - Issue stage is split and pipelined
  - Needs to check hazards within a packet, between two packets, among current and all the earlier instructions in execution.

In effect an n-fold pipeline with complex issue logic and large set of bypass paths.

| *Type* | *Pipe* | *Stages* | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Int. instruction | | IF | ID | EX | MEM | WB | | |
| FP instruction | | IF | ID | EX | MEM | WB | | |
| Int. instruction | | | IF | ID | EX | MEM | WB | |
| FP instruction | | | IF | ID | EX | MEM | WB | |
| Int. instruction | | | | IF | ID | EX | MEM | WB |
| FP instruction | | | | IF | ID | EX | MEM | WB |

# Superscalar with Speculation

- Speculative execution – *execute* control dependent instructions even when we are not sure if they should be executed
- With branch prediction, we speculate on the outcome of the branches and execute the program as if our guesses were correct. Misprediction? Hardware undo
  - Instructions after the branch can be fetched and issued, but can not execute before the branch is resolved
  - Speculation allows them to execute with care.
- Multi-issue + branch prediction + Tomasulo
- Implemented in a number of processors:

  PowerPC 603/604/G3/G4, Pentium II/III/4, Alpha 21264, AMD K5/K6/Athlon, MIPS R10k/R12k

# Hardware Modifications

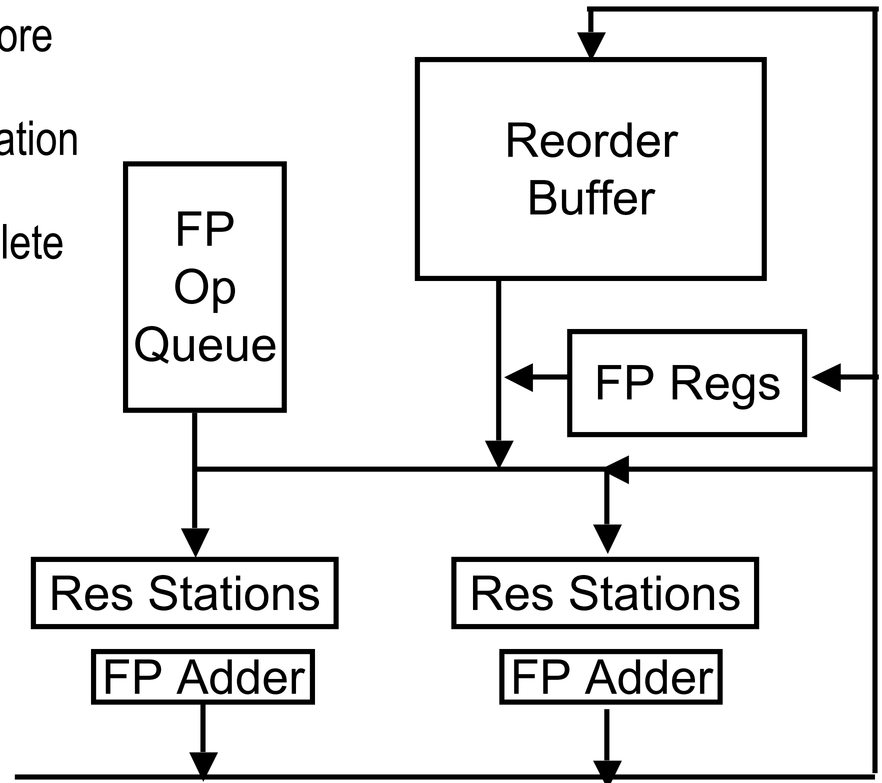
- Speculated instructions execute and generate results. Should they be written into register file? Should they be passed onto dependent instructions (in reservation stations)?
- Separate the bypassing paths from actual completion of an instruction. Do not allow speculated instructions to perform any updates that cannot be undone.
- When instructions are no longer speculative, allow them to update register or memory – *instruction commit*.
  - Out-of-order execution, in-order commit (provide precise exception handling)
- Then where are the instructions and their results between execution completion and instruction commit? Instructions may finish considerably before their commit.
- *Reorder buffer (ROB)* holds the results of instructions that have finished execution but have not committed.
  - ROB is a source of operands for instructions, much like the store buffer

# HW support for More ILP

- *Speculation*: allow an instruction to issue that is dependent on branch predicted to be taken *without* any consequences (including exceptions) if branch is not actually taken ("HW undo"); called "boosting"
- Combine branch prediction with dynamic scheduling to execute before branches resolved
- Separate *speculative* bypassing of results from real bypassing of results
  - When instruction no longer speculative, write boosted results (*instruction commit*) or discard boosted results
  - execute out-of-order but commit in-order to prevent irrevocable action (update state or exception) until instruction commits

# HW support for More ILP

- Need HW buffer for results of uncommitted instructions: *reorder buffer*
  - 3 fields: instr, destination, value
  - Reorder buffer can be operand source => more registers like RS
  - Use reorder buffer number instead of reservation station when execution completes
  - Supplies operands between execution complete & commit
  - Once operand commits, result is put into register
  - Instructions commit in order
  - As a result, its easy to undo speculated instructions on mispredicted branches or on exceptions



# Four Steps of Speculative Tomasulo Algorithm

1. Issue— get instruction from FP Op Queue

   If reservation station and reorder buffer slot free, issue instr & send operands & reorder buffer no. for destination (this stage sometimes called "dispatch")

2. Execution— operate on operands (EX)

   When both operands ready then execute; if not ready, watch CDB for result; when both in reservation station, execute; checks RAW (sometimes called "issue")

3. Write result— finish execution (WB)

   Write on Common Data Bus to all awaiting FUs
   & reorder buffer; mark reservation station available.

4. Commit— update register with reorder result

   - When instr. at head of reorder buffer & result present, update register with result (or store to memory) and remove instr from reorder buffer. Mispredicted branch flushes reorder buffer (sometimes called "graduation")

# Additional Functionalities of ROB

- Dynamically execute instructions while maintaining precise interrupt model.
  - In-order commit allows handling interrupts in-order at commit time
- Undo speculative actions when a branch is mispredicted
  - In reality, misprediction is expected to be handled as soon as possible. Flushing all the entries that appear after the branch, allowing those preceding instructions to continue.
  - Performance is very sensitive to branch-prediction mechanism
    - Prediction accuracy, misprediction detection and recovery
- Avoids hazards through memory (memory disambiguation)
  - WAW and WAR are removed since updating memory is done in order
  - RAW hazards are maintained by 2 restrictions:
    - A load's effective address is computed after all earlier stores
    - A load can not read from memory if there is an earlier store in ROB having the same effective address (some machine simply bypass the value from store to the load)