



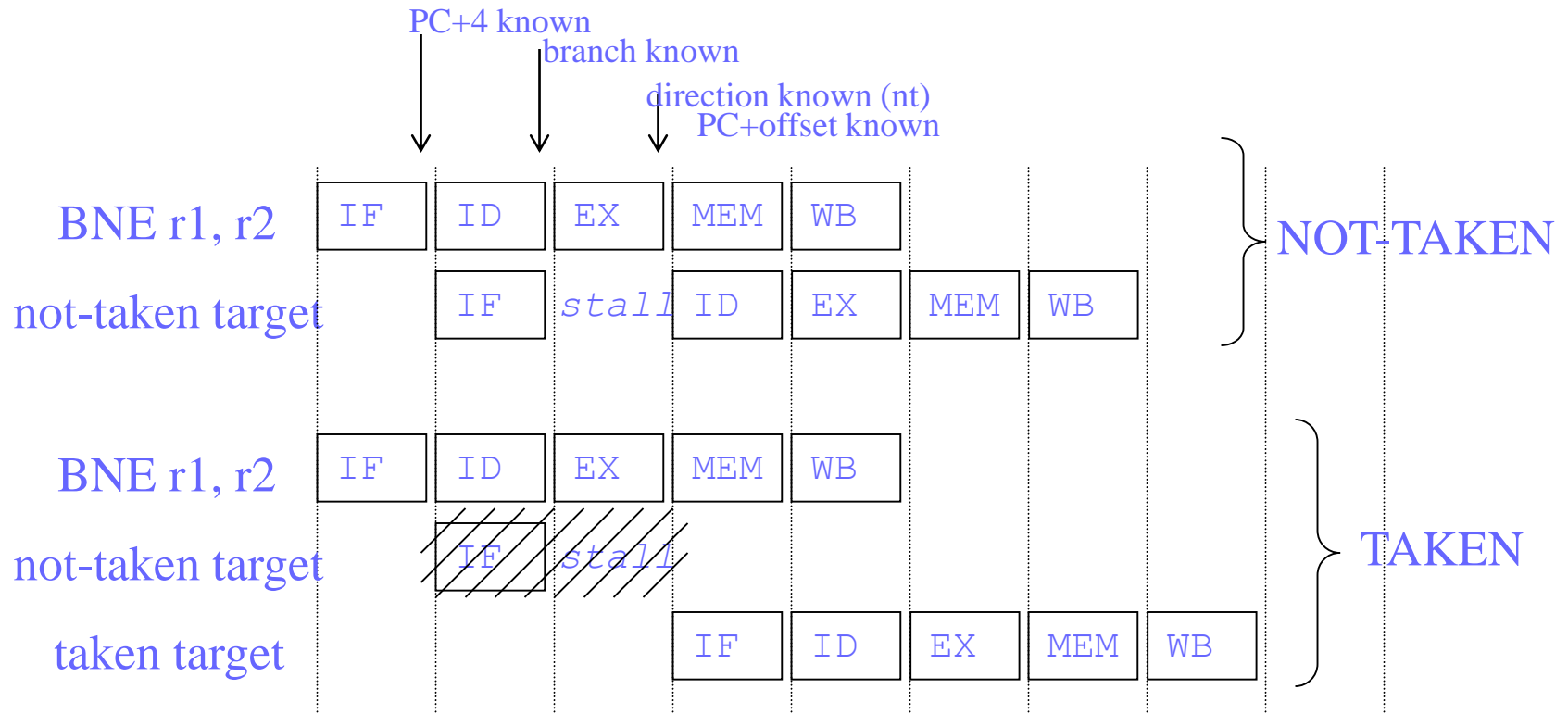
Control hazards

- Branches disrupts pipeline because we don't know what to fetch next
 - Problems
 - Don't know we have a branch until decode (ID)
 - Don't know taken target until execute (EX)
 - Don't know branch direction (taken/not taken) until execute (EX) or Memory stage MEM)



Handling Control Hazards: method 1

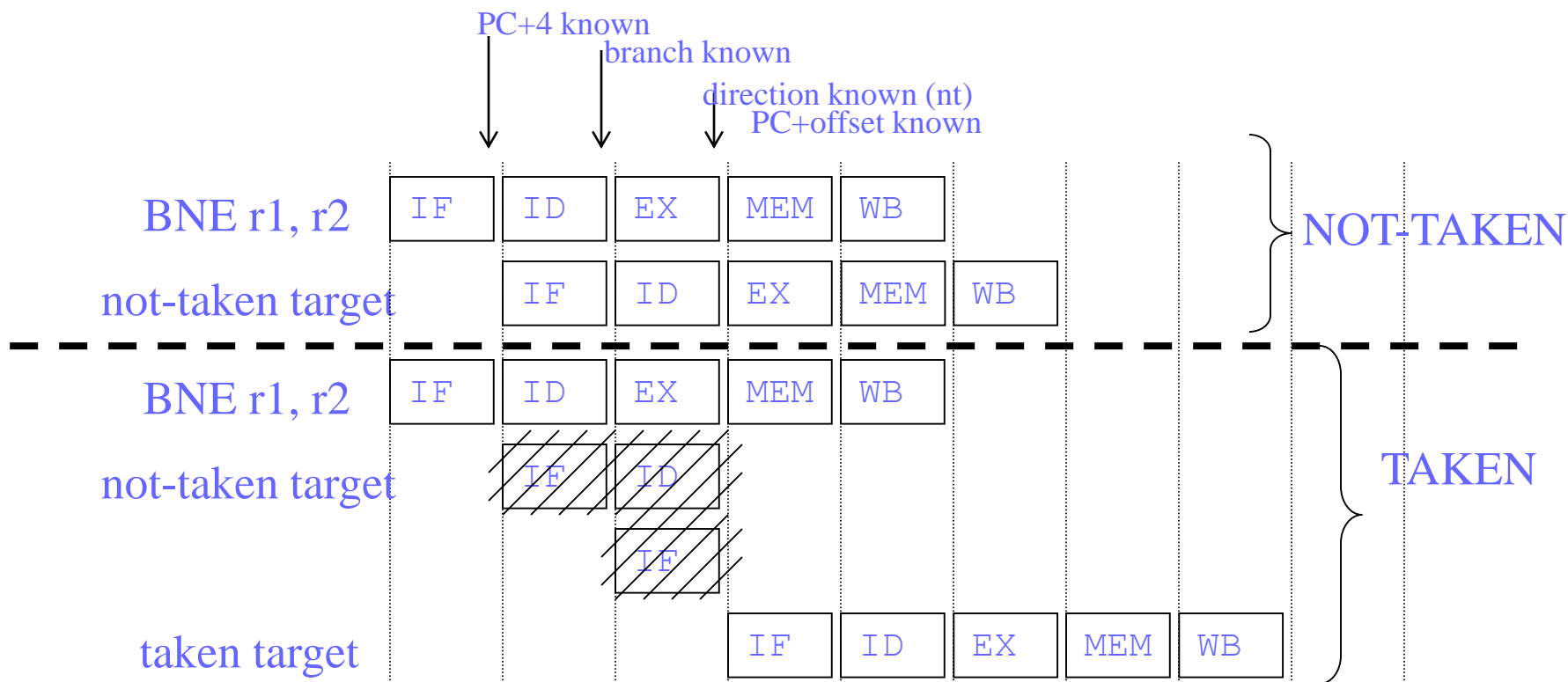
- stall





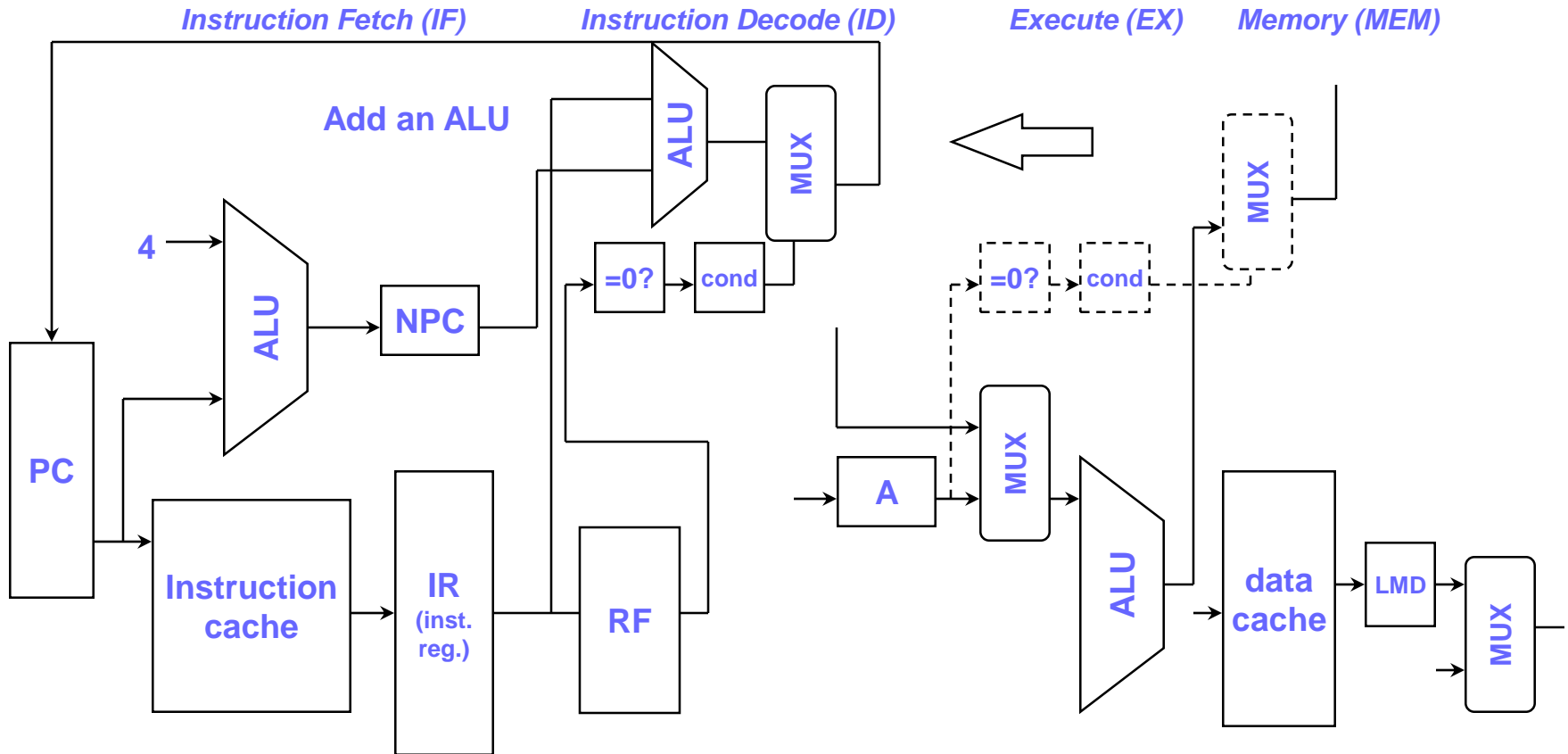
Handling Control Hazards: method 2

- predict not-taken





Move branch target back



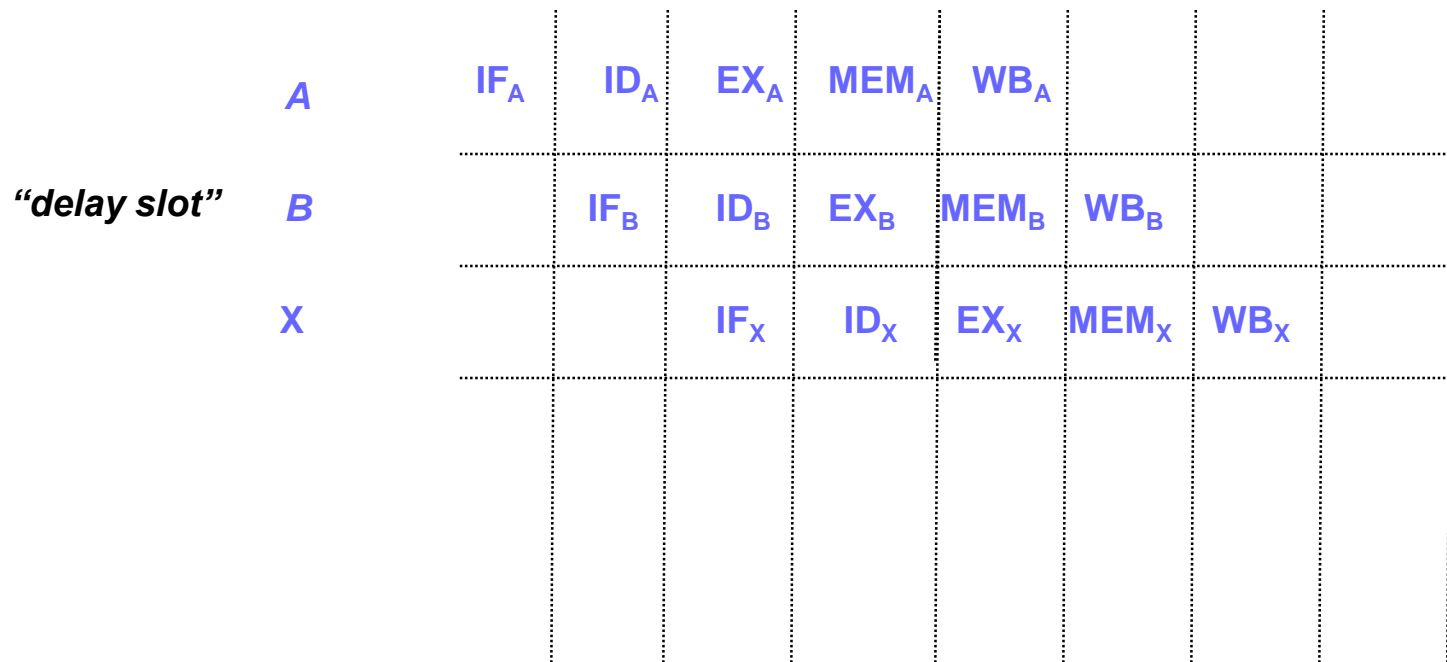
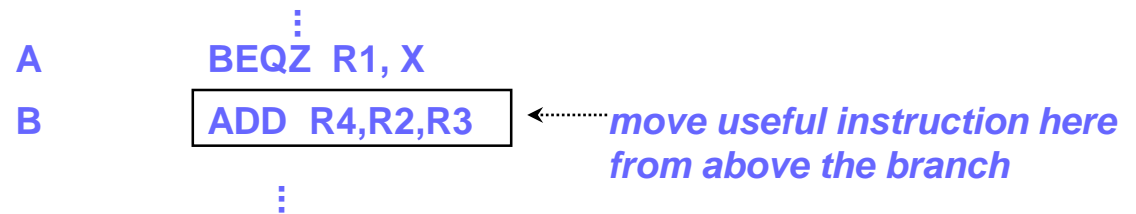
May increase time for ID

Eliminates 1 cycles, but still 1 additional stall



Reducing branch penalty via the compiler: Delay slots

- Change the meaning of a branch so that next instruction after branch holds something useful





Delay slots

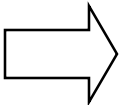
- Add n slots to cover n holes
- ISA is changed to mean “ n instructions after any branch are always executed”
- Problem:
 - ISA feature that encodes pipeline structure
 - Difficult to maintain across generations
 - Typically can fill:
 - 1 slot 75% of time
 - 2 slots about 25% of time
 - >2 slots almost never



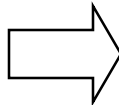
Filling slot from above branch

- Advantage
 - Delay slot instruction can always execute regardless of branch outcome (don't ever need to squash it)
- Disadvantage
 - Need a “safe” instruction from above the branch
 - Safe means: moving the instruction to the delay slot doesn't violate any data dependencies

GOOD SCENARIO

ADD R4, R2, R3		BEQZ R1, X
BEQZ R1, X		ADD R4, R2, R3
NOP		

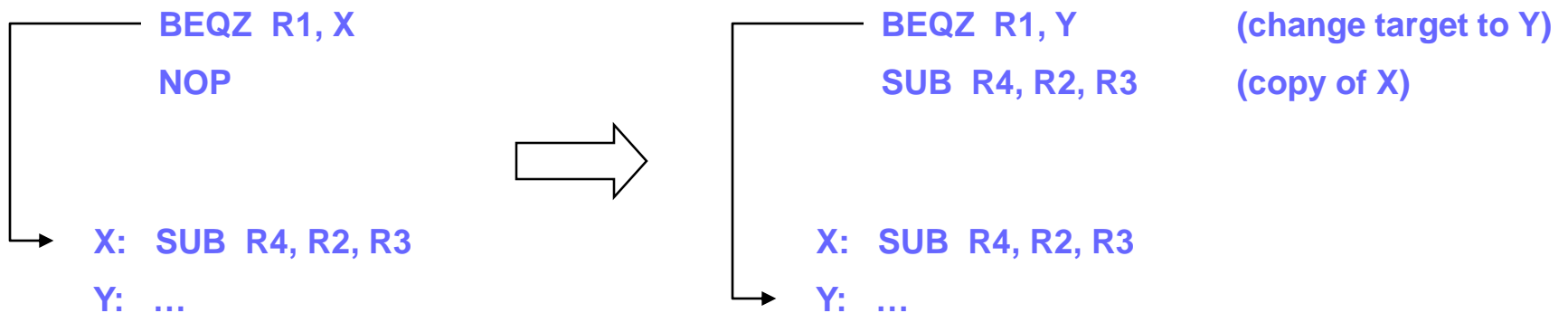
BAD SCENARIO

ADD R1, R2, R3		ADD R1, R2, R3
BEQZ R1, X		BEQZ R1, X
NOP		NOP



Filling slot from target or fall-through

- If you can't fill slot(s) from above the branch, use instructions from either:
 - Target of branch (if frequently taken)
 - Fall-through of branch (if frequently not-taken)
- Example: fill from target (branch is frequently taken)

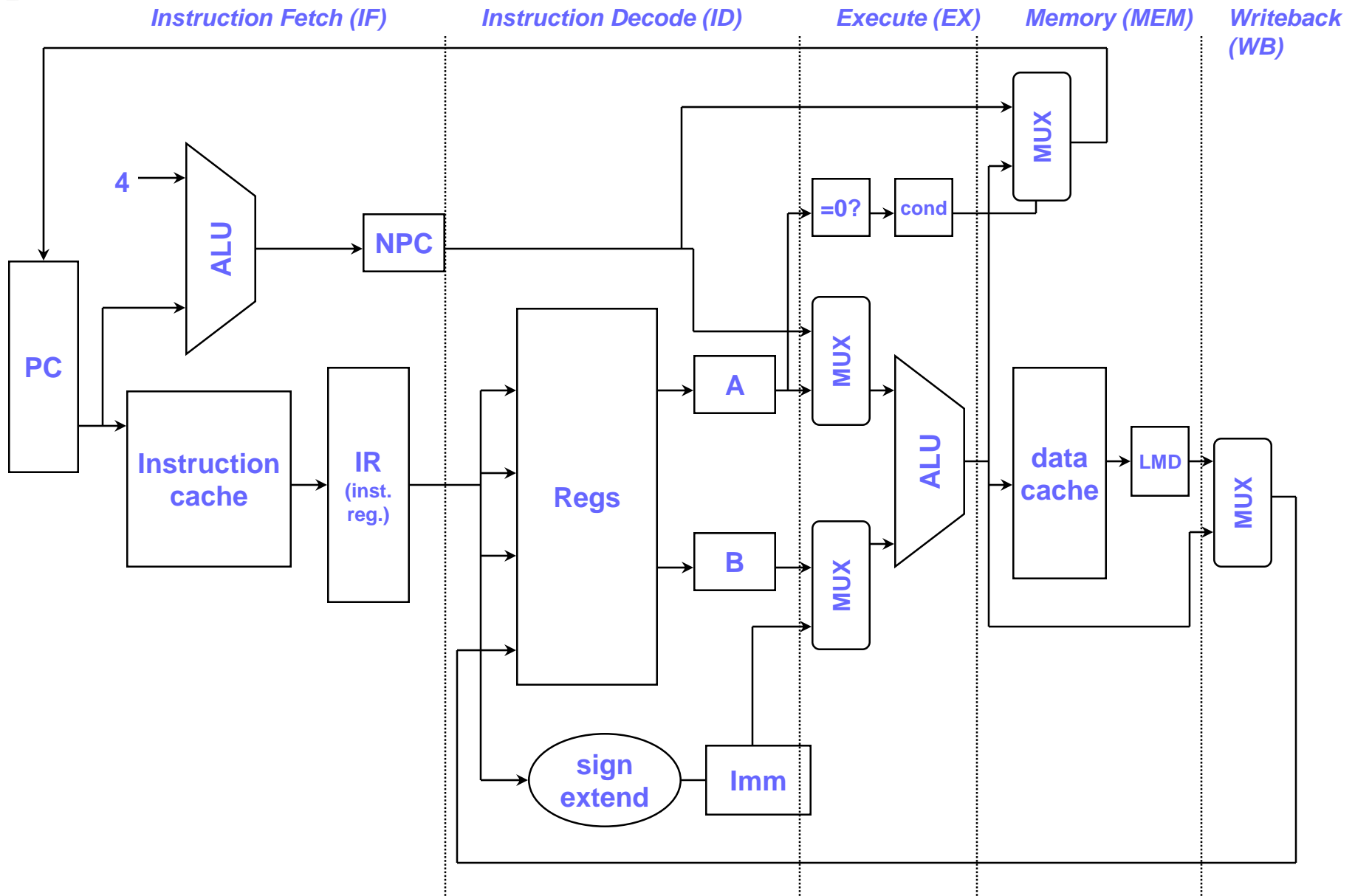


- Disadvantages
 - Only works if delay slot instruction is safe to execute when branch goes the opposite (infrequent) direction



Eliminating all stalls: Issues

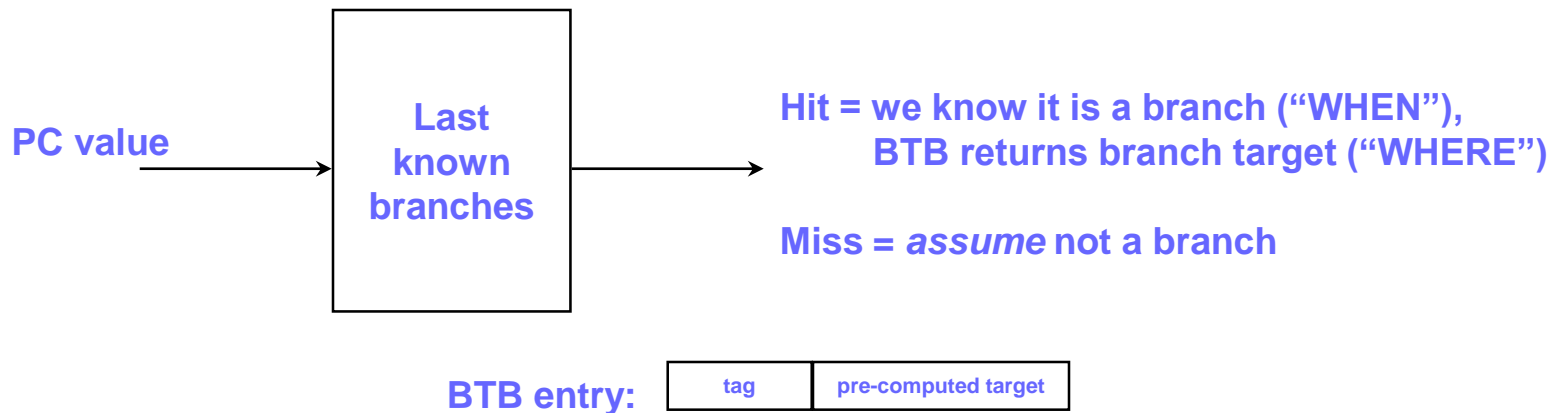
- When:
 - Detect an un-decoded instruction is a branch
- Where:
 - Predict where the branch will go (if taken)
- Whether:
 - (For conditional branches) Predict if it will be taken or not, *before* execution
- Optimal: try to determine all three in IF stage
 - Won't work perfectly (*prediction*), but we can try our best





Issues with When and Where

- What does IF know?
 - Only the address of the instruction (PC)
 - Keep buffer (cache) of last known branch targets around
 - Buffer is written to by WB stage



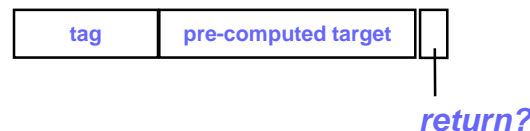
- Traditional name for this is a *Branch Target Buffer (BTB)*



Predicting Where with returns

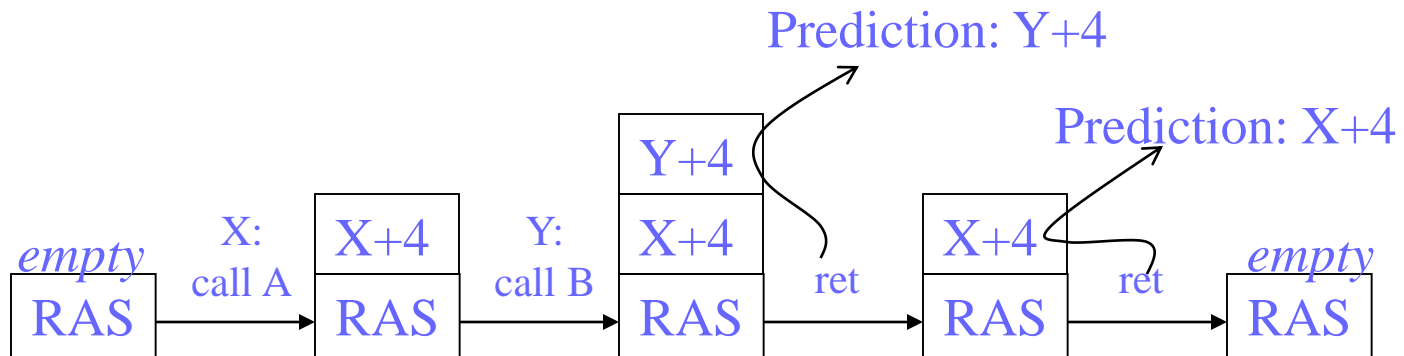
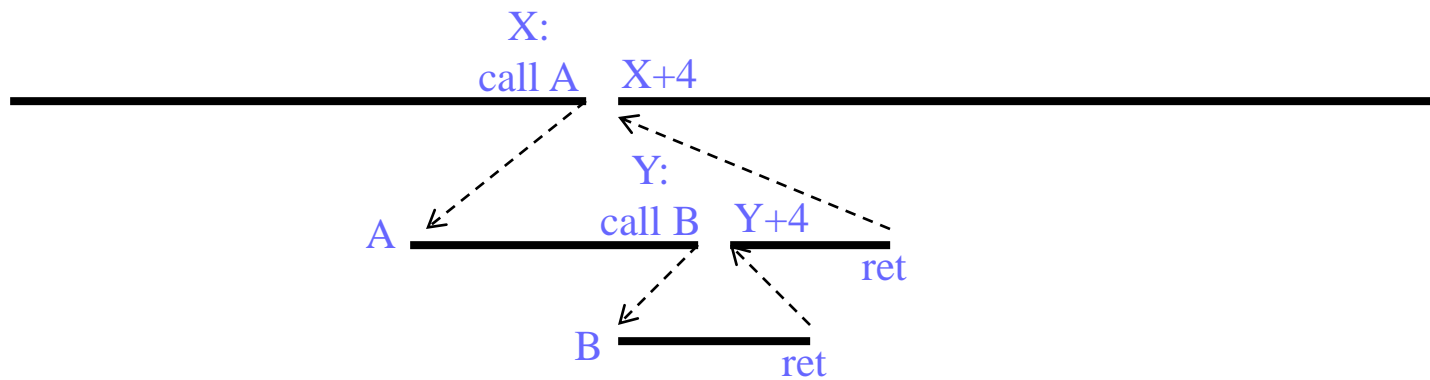
- Problem: A lot of jumps are returns from procedures
 - Holding the last target address is a poor predictor
- Solution: Keep a hardware “stack” of return addresses
 - Push return address when a “call” is executed
 - Pop buffer on returns to get prediction
 - Bottom of stack is filled with old value on a pop
 - Need approx 4-8 entries for integer code

*Each entry
in BTB now
contains:*





Return Address Stack





Issues with Whether

- Predicting conditional branches
 - And sometimes unconditional branches if needed before decode
- Two approaches:
 - Hardware to supply prediction
 - Software
 - Heuristics
 - Profiling



Hardware branch prediction

- 1-bit schemes (Pattern History Table):
 - Add 1-bit prediction field to branch target buffer
 - Set prediction field = 1 if branch was *taken*, 0 if branch was not *taken*
 - At IF, check “branch prediction buffer”:
 - if prediction field = 1 then predict *taken*
 - else predict *not-taken*
 - Problems:
 - Some branches don’t do what they did last time!
 - Think of a simple 10 iteration loop, start predict NT
 - What is prediction accuracy?
 - Isn’t this high enough?!?
 - Need more sophisticated predictor



Why accuracy matters so much

$$\text{speedup} = \text{efficiency} \times n = \frac{n}{1 + \text{stall cycles}}$$

$$\text{efficiency} = \frac{1}{1 + \text{stall cycles}}$$

$$\text{stall cycles} = \text{branch penalty} \times (1 - \text{accuracy}) \times \text{fraction}_{\text{branch}}$$

$$\text{efficiency} = \frac{1}{1 + \text{branch penalty} \times (1 - \text{accuracy}) \times \text{fraction}_{\text{branch}}}$$

$$\text{branch penalty} \times (1 - \text{accuracy}) \times \text{fraction}_{\text{branch}} = \frac{1}{\text{efficiency}} - 1$$

$$1 - \text{accuracy} = \frac{\frac{1}{\text{efficiency}} - 1}{\text{branch penalty} \times \text{fraction}_{\text{branch}}}$$

$$\text{accuracy} = 1 - \frac{\frac{1}{\text{efficiency}} - 1}{\text{branch penalty} \times \text{fraction}_{\text{branch}}}$$

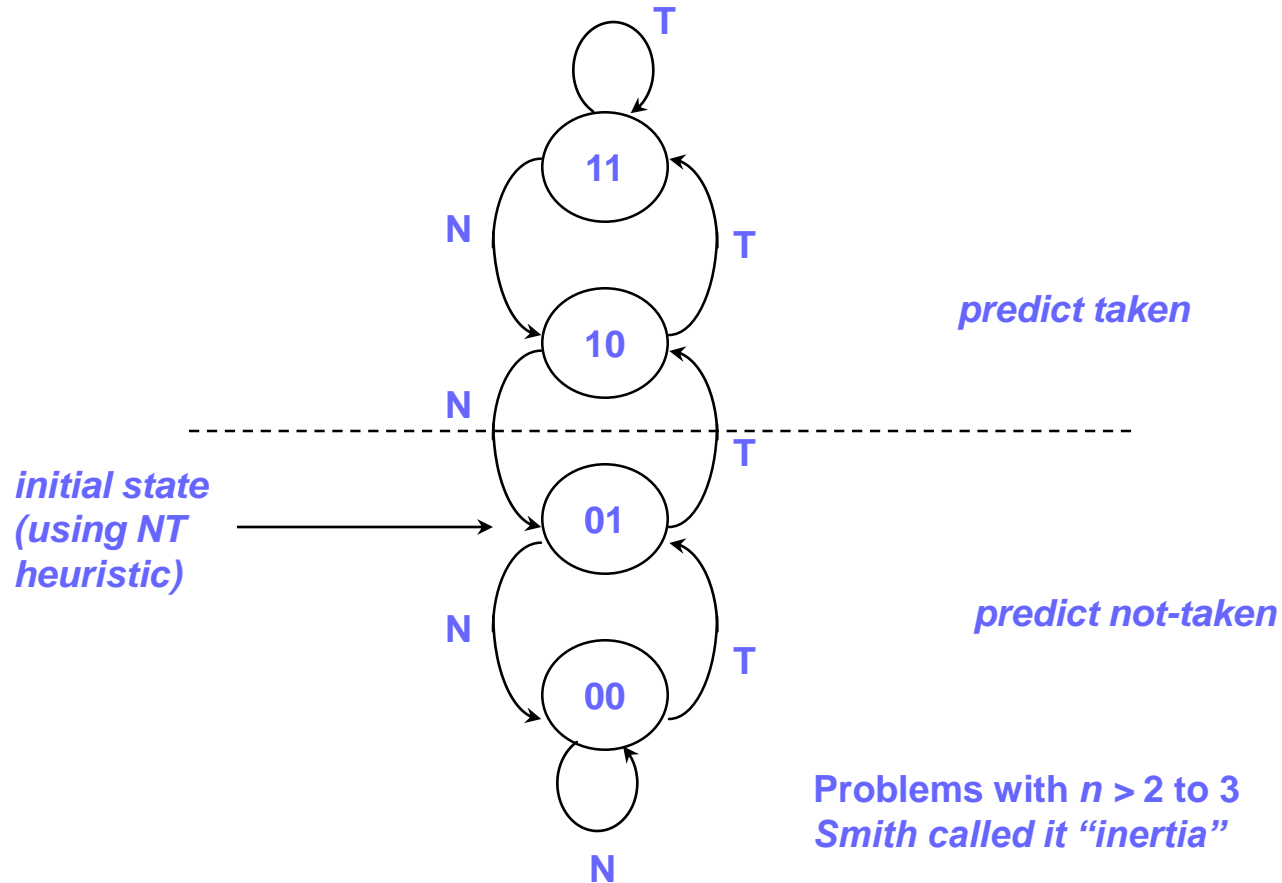
	accuracy	accuracy
branch penalty	eff = 0.9	eff = 0.99
1	44.44%	94.95%
2	72.22%	97.47%
3	81.48%	98.32%
4	86.11%	98.74%
10	94.44%	99.49%
20	97.22%	99.75%

- Reduce stalls by:
 - Decreasing branch penalty
 - Modify the pipeline (HW question)
 - Increasing accuracy
 - Fancy prediction schemes
 - Decreasing fraction of branches
 - compile-time code ordering



Smith n -bit counter predictor

- Replace prediction bit with n -bit counter:





Example of Smith counter

	<u>T</u>	<u>T</u>	<u>T</u>	<u>T</u>	<u>T</u>	<u>T</u>	<u>N</u>	<u>T</u>	<u>T</u>	<u>N</u>	<u>N</u>	<u>N</u>	<u>N</u>	<u>N</u>	<u>N</u>	<u>N</u>	<u>N</u>	<u>T</u>	<u>T</u>
previous state	01	10	11	11	11	11	11	10	11	11	10	01	00	00	00	00	00	00	01
new state	10	11	11	11	11	11	10	11	11	10	01	00	00	00	00	00	00	01	10

6 mispredictions out of 19 branch executions

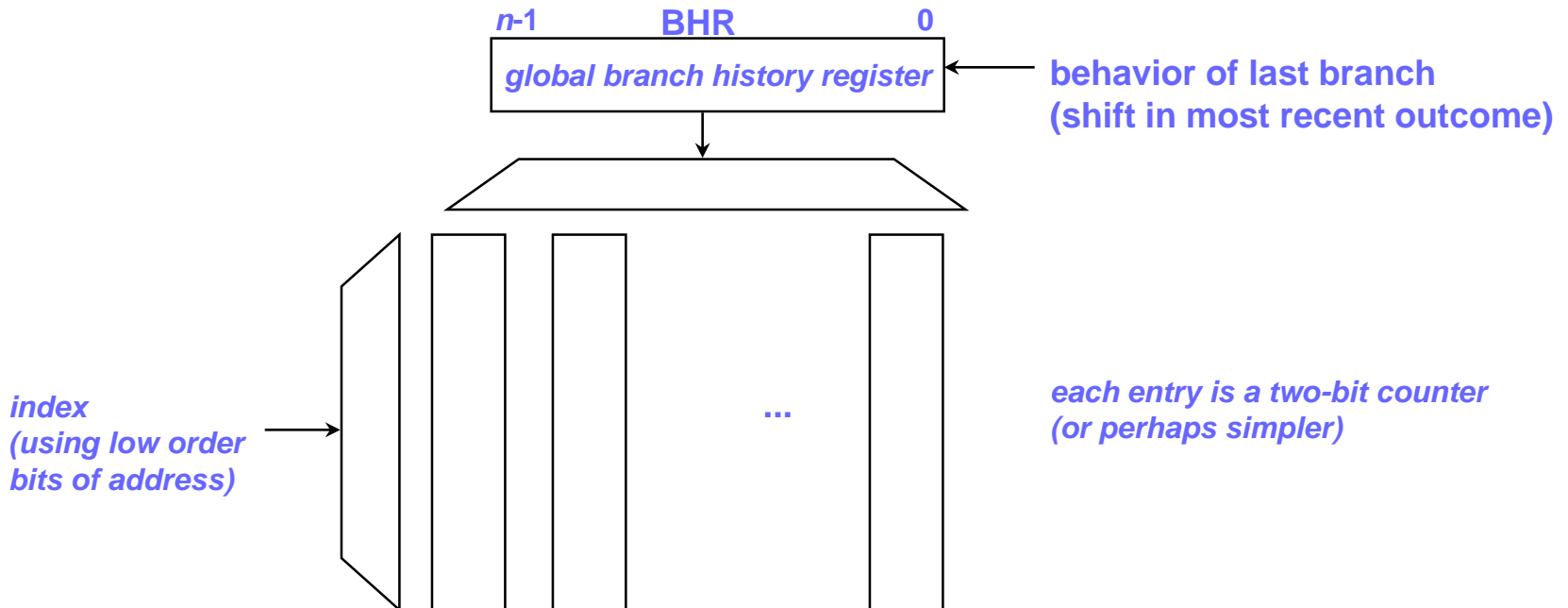
	<u>T</u>	<u>N</u>	<u>T</u>	<u>N</u>	<u>T</u>	<u>N</u>	<u>T</u>	<u>N</u>	<u>T</u>	<u>N</u>	<u>T</u>	<u>N</u>	<u>T</u>	<u>N</u>	<u>T</u>	<u>N</u>	<u>T</u>	<u>N</u>
previous state	01	10	01	10	01	10	01	10	01	10	01	10	01	10	01	10	01	10
new state	10	01	10	01	10	01	10	01	10	01	10	01	10	01	10	01	10	01

19 mispredictions out of 19: the infamous “*toggle branch*”



Improving the smith counter

- Options:
 - Capture correlations between branches (“global”)
 - Associate predictions with branch histories, not branch addresses (use different indexing scheme)
- Gselect (global history with index selection):





Gselect example

initially $R1 = 0$

A: BEQZ R1, D

...

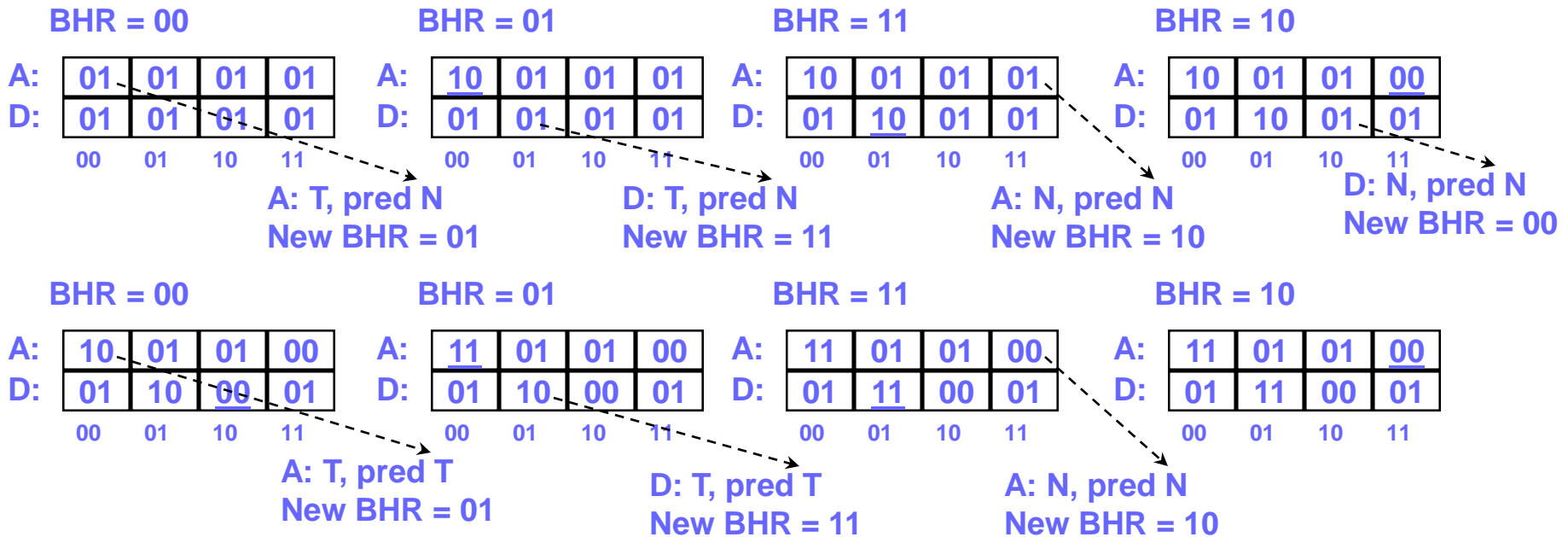
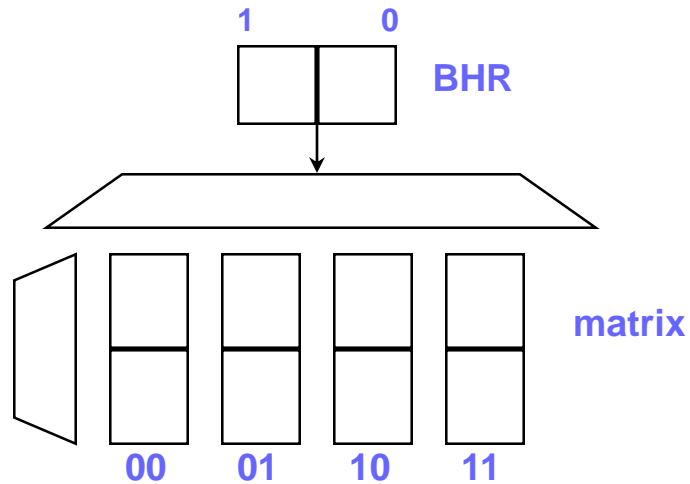
D: BEQZ R1, F

...

F: NOT R1, R1

G: JUMP A

underlined means entry
was updated due to last
branch execution





Gshare (global history with index sharing) Branch Predictor

