CS510 Computer Architecture

Lecture 8: Dynamic Branch Prediction

Soontae Kim

Spring 2017

School of Computing, KAIST

Instruction Dependencies

- **Determining instruction dependencies (dependence analysis) is important for pipeline scheduling and to determine the amount of instruction level parallelism (ILP) in the program to be exploited.**
- **Instruction Dependence Graph: A directed graph where nodes represent instructions and edges represent instruction dependencies.**
- **If two instructions are independent or parallel (no dependencies between them exist), they can be executed simultaneously in the pipeline without causing stalls (no pipeline hazards); assuming the pipeline has sufficient resources (no structural hazards).**
- **Instructions that are dependent are not parallel and cannot be reordered by the compiler or hardware.**
- **Instruction dependencies are classified as:**

  - **Data dependencies**
  - **Name dependencies** ← Name: Register or Memory Location
  - **Control dependencies**

Control Dependencies

- **Determines the ordering of an instruction with respect to a branch instruction.**
- **Every instruction in a program except those in the very first basic block of the program is control dependent on some set of branches.**
- **An instruction which is control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch.**
- **An instruction which is not control dependent on the branch cannot be moved so that its execution is controlled by the branch (in the then portion)**
- **Example of control dependence in the then part of an if statement:**

```
if  p1 {
        S1;
};
If p2  {
        S2;
}
```

S1 is control dependent on p1
S2 is control dependent on p2 but not on p1

What happens if S1 is moved here?

Reduction of Control Hazards Stalls with Dynamic Branch Prediction

- **So far we have dealt with control hazards in instruction pipelines by:**
  - **Assuming that the branch will not be taken (i.e stall cycles when branch is taken).**
  - **Branch delay slot and canceling branch delay slot. (ISA support needed)**
  - **Reducing the branch penalty by resolving the branch early in the pipeline**
    - **Branch penalty if branch is taken = stage resolved - 1**
  - **Compiler-based static branch prediction encoded in branch instructions**
    - **Prediction is based on program profile or branch direction**
    - **ISA support needed.**

How to further reduce the impact of branches on pipelined processor performance ? branch solve early 1. branch detection (taken or not taken) 2. branch target address

- **<u>Dynamic Branch Prediction:</u>**
  - **Hardware-based schemes that utilize run-time behavior of branches to make dynamic predictions:**
- **<u>Branch Target Buffer (BTB):</u>**
  - **To provide branch target addresses fast in the fetch stage**

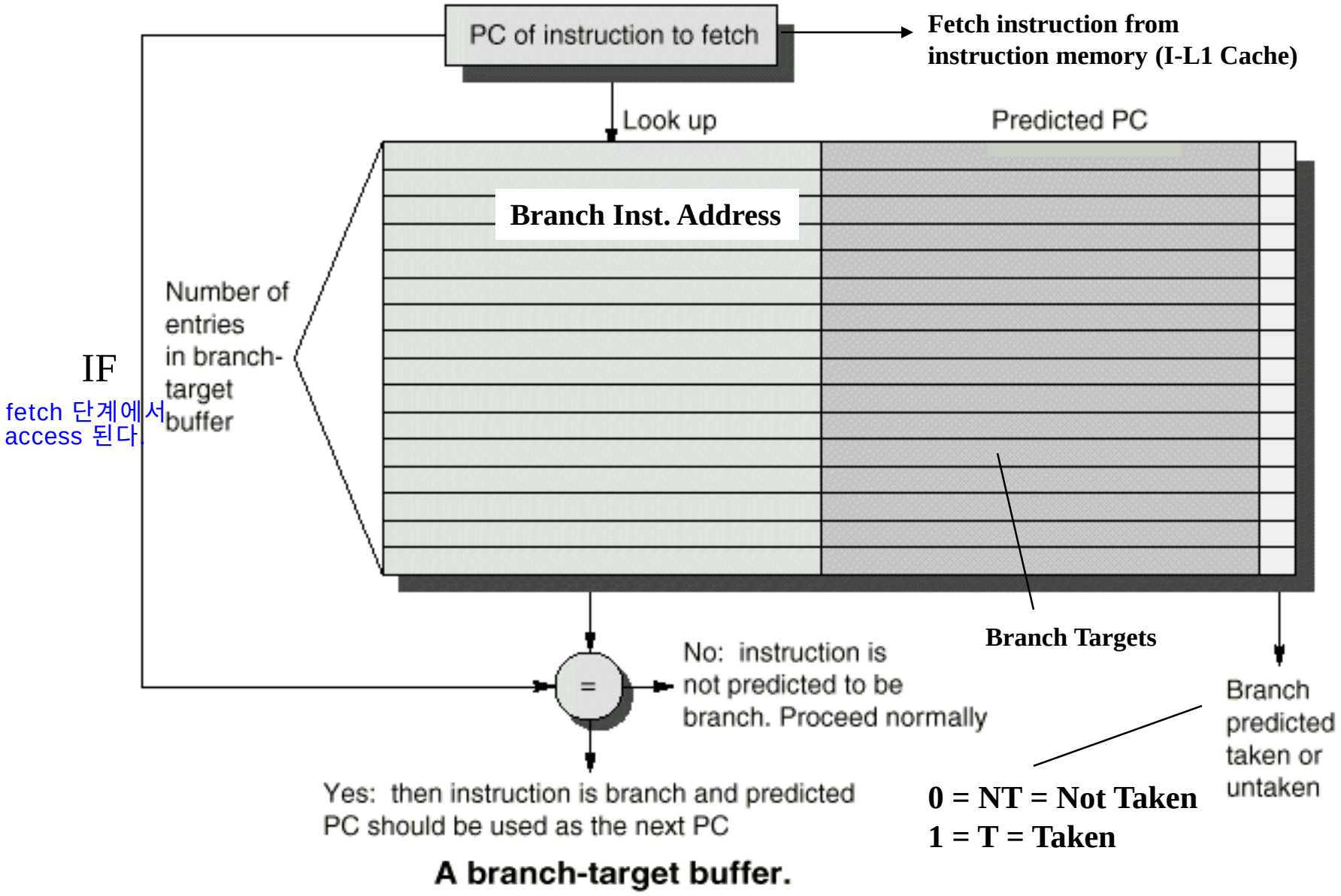
Dynamic Branch Prediction

- **Use the run-time behavior of branches to make more accurate predictions than possible using static prediction.**
- **Some of the proposed dynamic branch prediction mechanisms include:**
  - **One-level or Bimodal: Uses a Branch History Table (BHT), a table of usually two-bit saturating counters which is indexed by a portion of the branch instruction address (low-order address bits). (First proposed mid 1980s)**
  - **Two-Level Adaptive Branch Prediction. (First proposed early 1990s),**
  - **Hybrid or Tournament Predictors: Uses a combination of two or more (usually two) branch prediction mechanisms (1993).** single 보다 효율 좋다.
- **To reduce the stall cycles resulting from correctly predicted taken branches to zero cycles, a Branch Target Buffer (BTB) that includes the addresses of conditional branches that were taken along with their targets is added to the fetch stage.**

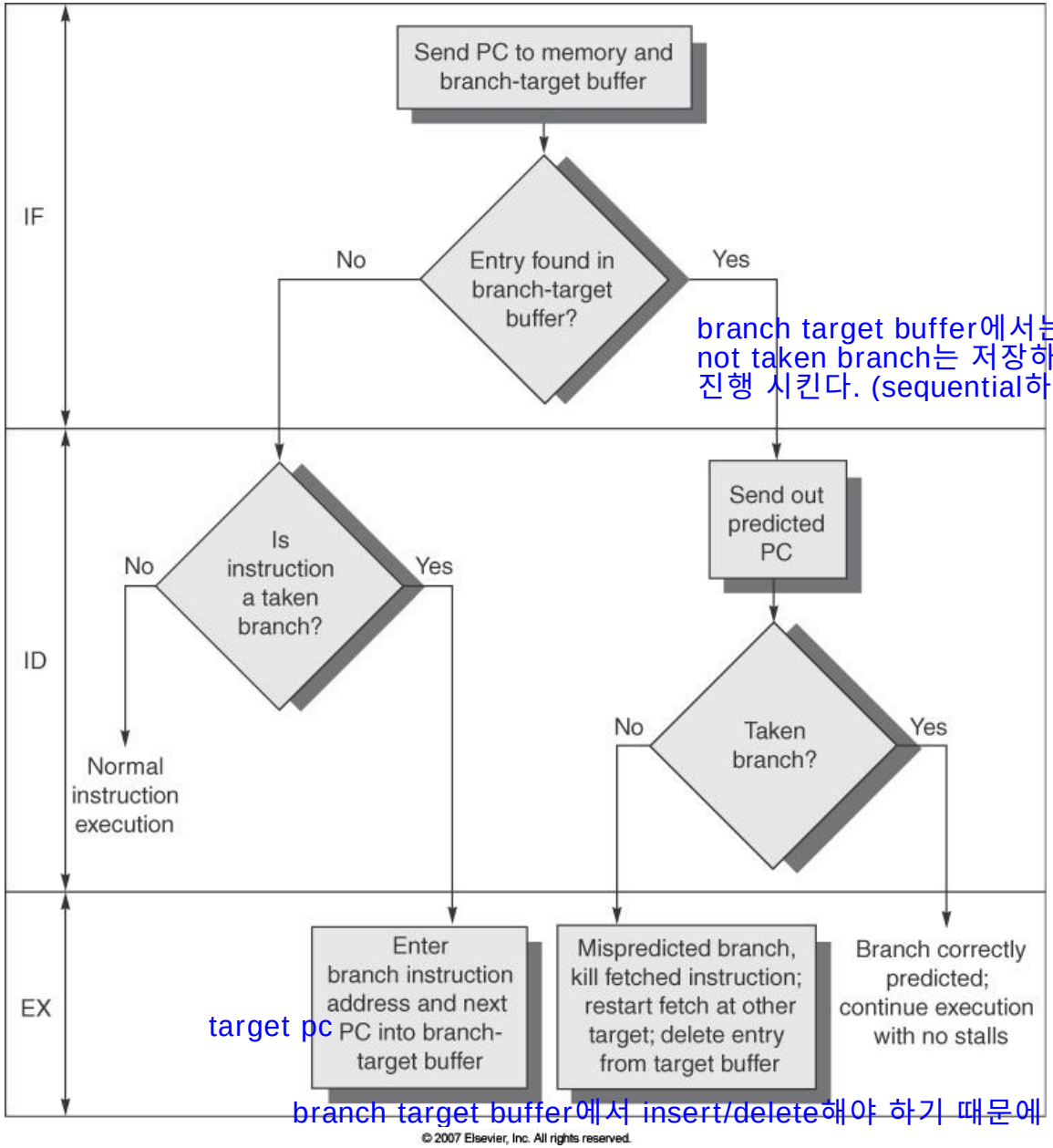
Branch Target Buffer (BTB)

- **Effective branch prediction requires the target address of the branch at an early pipeline stage.**
- **One can use additional adders to calculate the target, as soon as the branch instruction is decoded. This would mean that one has to wait until the ID stage and target instruction would be fetched with a one-cycle penalty (this was done in the enhanced MIPS pipeline)**
- **To avoid this problem one can use a Branch Target Buffer (BTB), where the addresses of taken branch instructions are stored together with their target addresses.** taken 되는 것만 저장한다. (taken 안되는거 저장 안해도 무관)
- **Some designs store n prediction bits as well, implementing a combined BTB and Branch history Table (BHT).**
- **Instructions are fetched from the target address stored in the BTB in case the branch is predicted-taken and found in BTB. After the branch has been resolved the BTB is updated. If a branch is encountered for the first time a new entry is created in BTB once it is resolved.**
- **Branch Target Instruction Cache (BTIC): A variation of BTB which caches also the branch target instruction in addition to its address. This eliminates the need to fetch the target instruction from the instruction cache or from memory.**

Basic Branch Target Buffer (BTB)



A branch-target buffer.



One more stall to update BTB
Penalty = 1 + 1 = 2 cycles

branch target buffer에서는 taken branch만 저장되어 있다.
not taken branch는 저장하지 않고, branch가 아닌 것처럼 진행 시킨다. (sequential하게 다음 address 실행)

target pc

branch target buffer에서 insert/delete해야 하기 때문에

Branch Penalty Cycles Using A Branch-Target Buffer (BTB)

Base Pipeline Taken Branch Penalty = 1 cycle

No **Not Taken** **Not Taken** **0**

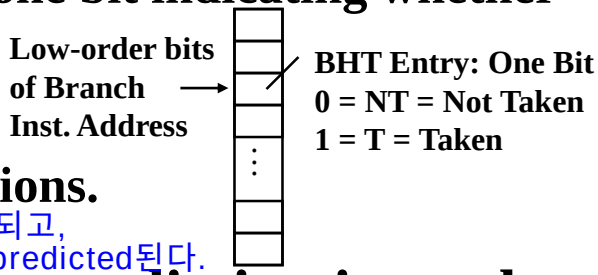
Instruction in buffer	Prediction	Actual branch	Penalty cycles
Yes	Taken	Taken	0
Yes	Taken	Not taken	2
No	**Not Taken**	Taken	2

Assuming one more stall cycle to update BTB
Penalty = 1 + 1 = 2 cycles

Penalties for all possible combinations of whether the branch is in the buffer and what it actually does, assuming we store only taken branches in the buffer.

Basic Dynamic Branch Prediction

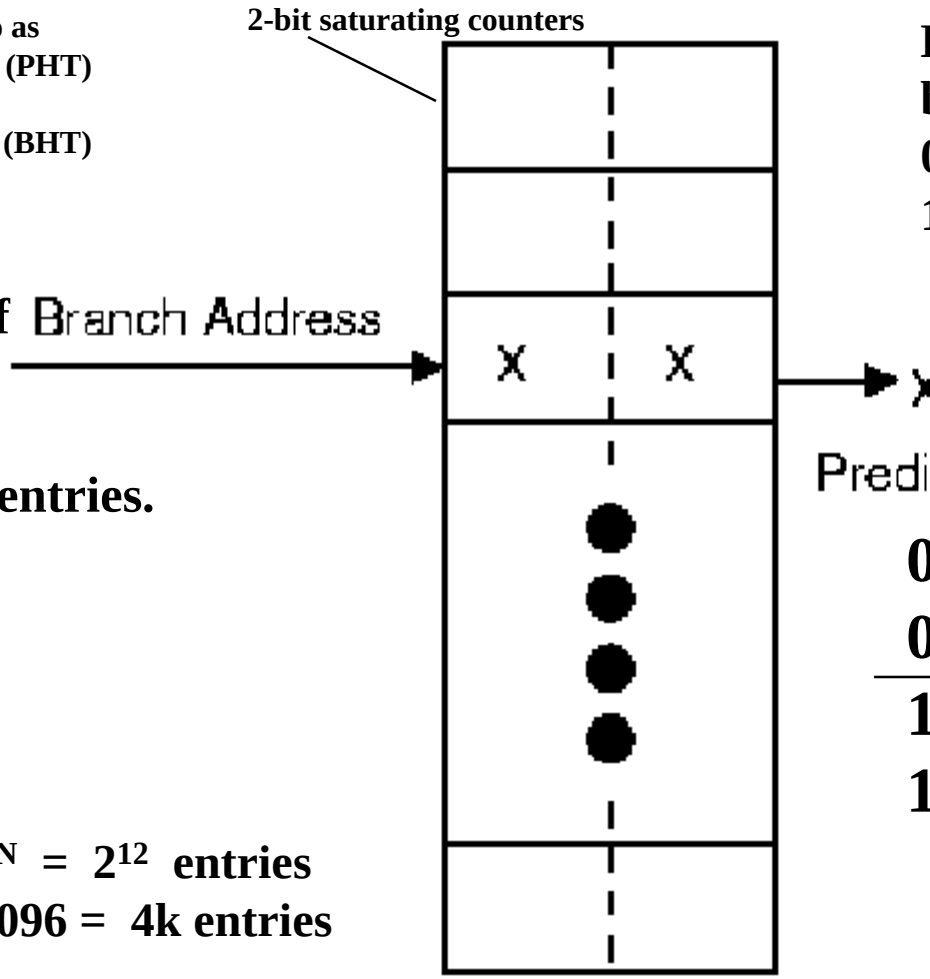
- **Simplest method: (One-Level or Bimodal)**
  - **A branch prediction buffer or Branch History Table (BHT) indexed by low-order address bits of the branch instruction.**
  - **Each buffer location (or BHT entry) contains one bit indicating whether the corresponding branch was recently taken**
    - e.g 0 = not taken , 1 =taken
  - **Always mispredicts in first and last loop iterations.**
    initial은 not taken이기 때문에, first에서는 loop가 mispredicted되고,
    last loop에서는 바로 전 과거가 taken인데, not taken이라서 mispredicted된다.



- **To improve prediction accuracy, two-bit prediction is used:**
  - **Prediction must miss twice before it is changed.**
  - **Two-bit prediction is a specific case of n-bit saturating counter incremented when the branch is taken and decremented when the branch is not taken.**
  - **Two-bit prediction counters are usually used based on observations that the performance of two-bit BHT prediction is comparable to that of n-bit predictors.**

One-Level Bimodal Branch Predictors
Decode History Table (DHT)

Sometimes referred to as
Pattern History Table (PHT)
or
Branch History Table (BHT)



High bit determines branch prediction
0 = NT = Not Taken
1 = T = Taken

0 0		Not Taken (NT)
0 1		
1 0		Taken (T)
1 1		

Table has 2^N entries.

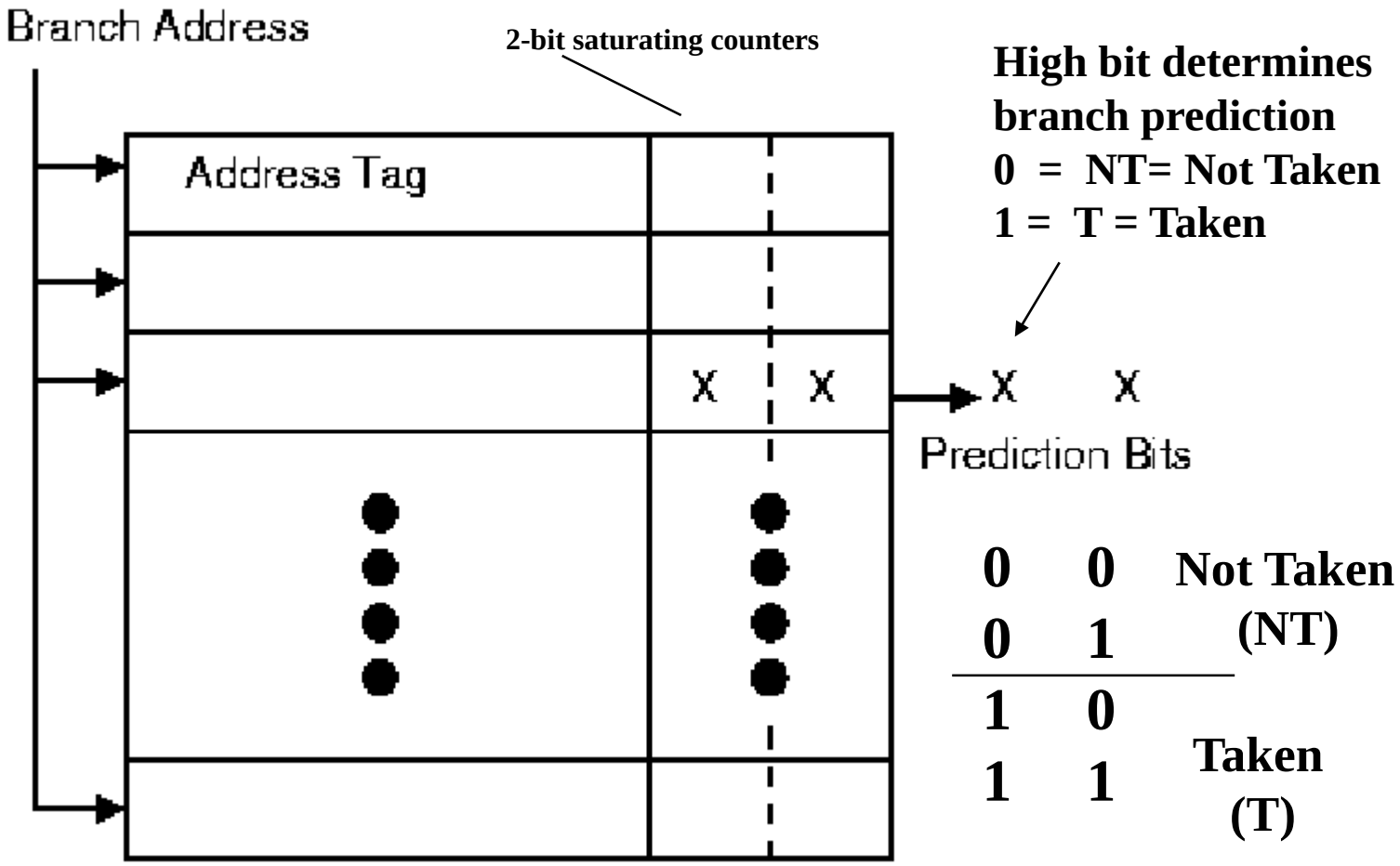
Example:

For N =12
Table has $2^N = 2^{12}$ entries
= 4096 = 4k entries

Number of bits needed = 2 x 4k = 8k bits

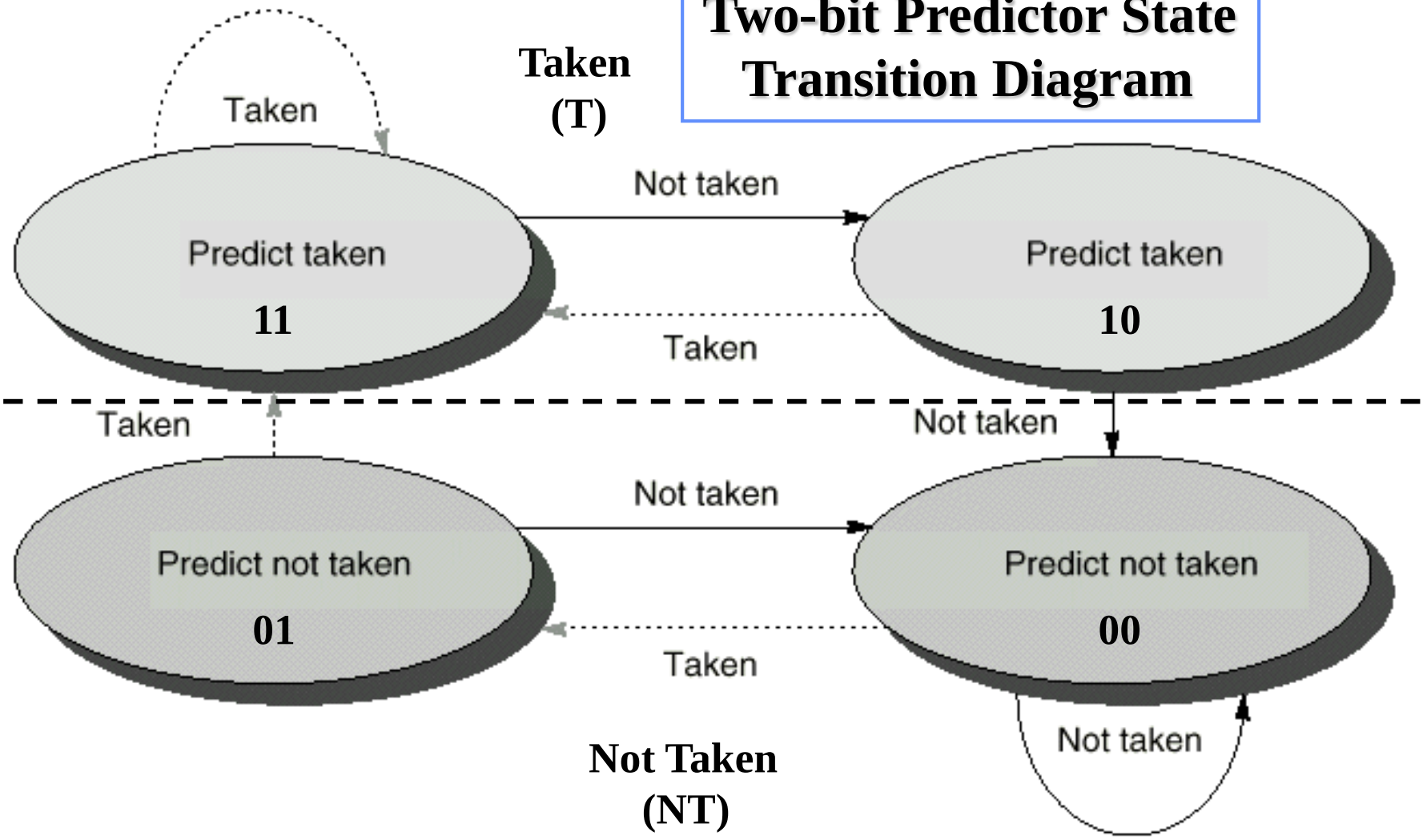
Common one-level implementation

One-Level Bimodal Branch Predictors
Branch History Table (BHT)



Not a common one-level implementation

Basic Dynamic Two-Bit Branch Prediction:



The states in a two-bit prediction scheme.

predict을 바꿀려면 연속적으로 두번의 사건이 일어나야 가능하다.
predict not taken -> predict taken (taken, taken)
predict taken -> predict not taken (not taken, not taken)

Dynamic Branch Prediction: Example

```
if (d==0)
    d=1;
if (d==1)
```

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

Possible execution sequences for a code fragment.

Initial value of d	d==0?	b1	Value of 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

Behavior of a one-bit predictor initialized to not taken.

d=?	b1 prediction	b1 action	New b1 prediction	b2 prediction	b2 action	New b2 prediction
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT

8 miss predict

One Level with one-bit table entries : NT = 0 = Not Taken, T = 1 = Taken

One level (0,1)

Correlating Branches

Recent branches are possibly correlated: The behavior of recently executed branches affects prediction of current branch.

Example:

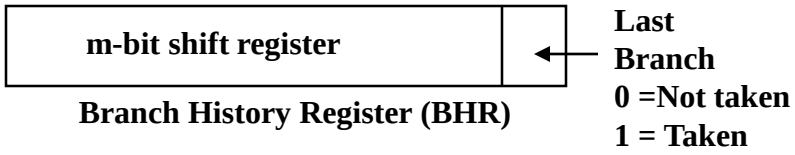
```
B1      if (aa==2)
           aa=0;   (not taken)
B2      if (bb==2)
           bb=0;   (not taken)
B3      if (aa!==bb){
```

```
        DSUBUI   R3, R1, #2
        BNEZ     R3, L1      ;  b1   (aa!=2)
        DADD     R1, R0, R0  ;  aa==0
L1:     DSUBUI   R3, R2, #2
        BNEZ     R3, L2      ; b2  (bb!=2)
        DADD     R2, R0, R0  ; bb==0
L2:     DSUBU    R3, R1, R2  ; R3=aa-bb
        BEQZ     R3, L3      ; b3   (aa==bb)
```

Branch B3 is correlated with branches B1, B2. If B1, B2 are both not taken, then B3 will be taken. Using only the behavior of one branch cannot detect this behavior.

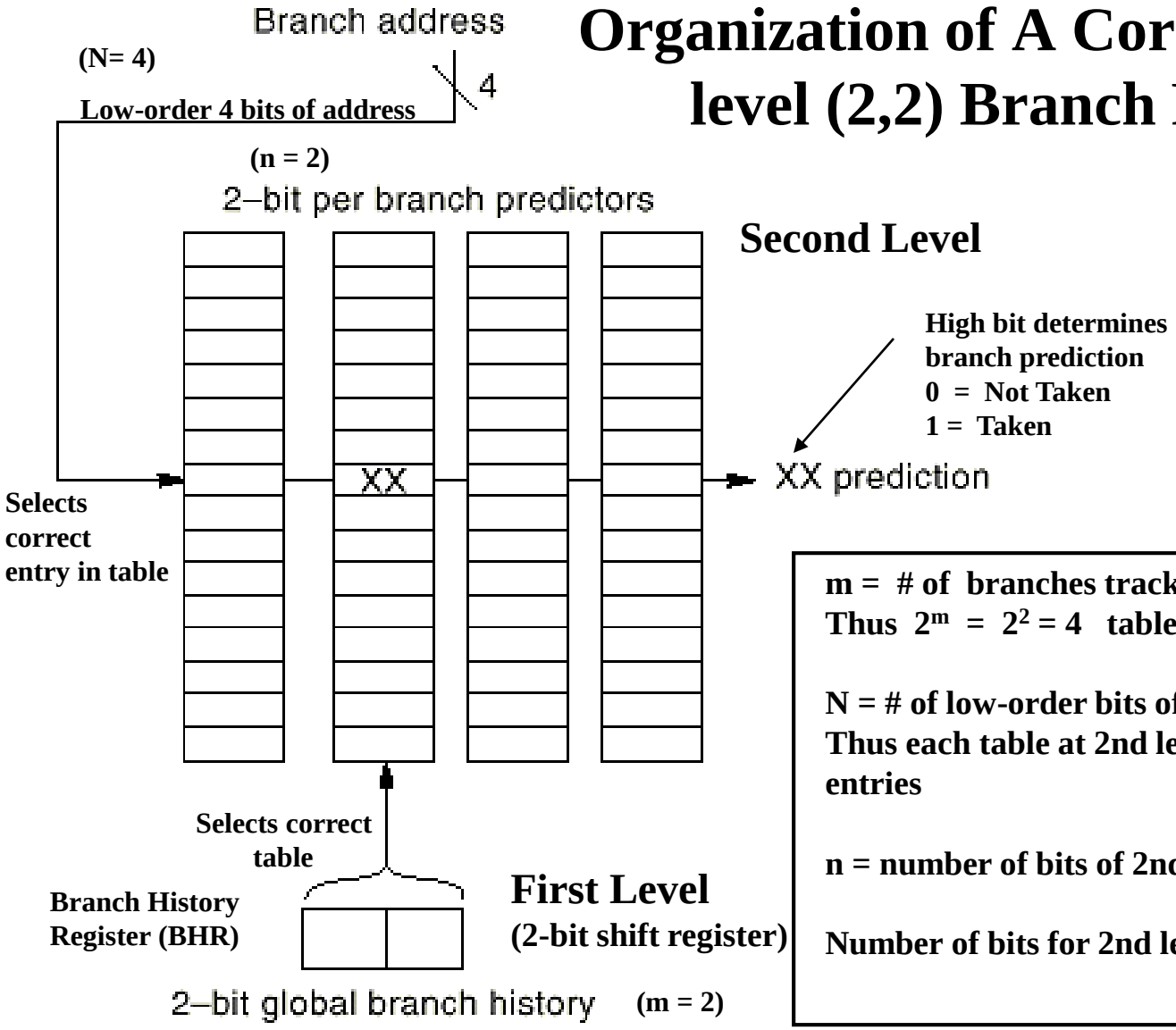
Correlating Two-Level Dynamic Branch Predictors

- Improve branch prediction by looking not only at the history of the branch in question but also at those of other branches.



- Uses two levels of branch history:
  - First level (global):
    - Record the global pattern or history of the m most recently executed branches as taken or not taken. Usually an m-bit shift register.
  - Second level (per branch address):
    - 2^m prediction tables, each table entry has n bit saturating counter.
    - The branch history pattern from first level is used to select the proper branch prediction table at the second level.
    - The low N bits of the branch instruction address are used to select the correct prediction entry within the selected table, thus each of the 2^m tables has 2^N entries and each entry is a n-bit counter.
    - Total number of bits needed for second level = $2^m \times 2^N \times$ n bits
- In general, the notation: (m,n) predictor means:
  - Record last m branches to select from 2^m history tables.
  - Each second level table uses n-bit counters (each table entry has n bits).
- Basic two-bit single-level Bimodal BHT is then a (0,2) predictor.

Organization of A Correlating Two-level (2,2) Branch Predictor



m = # of branches tracked at first level = 2
Thus $2^m = 2^2 = 4$ tables at second level

N = # of low-order bits of branch address = 4
Thus each table at 2nd level has $2^N = 2^4 = 16$ entries

n = number of bits of 2nd level table entry = 2

Number of bits for 2nd level = $2^m \times n \times 2^N$
$= 4 \times 2 \times 16 = 128$ bits

A (2,2) branch-prediction buffer uses a two-bit global history to choose from among four predictors for each branch address.

Dynamic Branch Prediction: Example (continued)

```
if (d==0)
    d=1;
if (d==1)
```

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

Combinations and meaning of the taken/not taken prediction bits.

Initial value of d	d==0?	b1	Value of 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

The action of the one-bit predictor with one bit of correlation, initialized to not taken/not taken.

d=?	b1 prediction	b1 action	New b1 prediction	b2 prediction	b2 action	New b2 prediction
2	**NT**/NT	T	T/NT	NT/**NT**	T	NT/T
0	T/**NT**	NT	T/NT	**NT**/T	NT	NT/T
2	**T**/NT	T	T/NT	NT/**T**	T	NT/T
0	T/**NT**	NT	T/NT	**NT**/T	NT	NT/T

Two level (1,1)