Dynamic Branch Prediction

Control Dependences and Branch Prediction

- Modern processors have deep pipelines
  - Pipeline depth for i7 is 14 stages
  - Because of the combination of deep pipelining and multiple issues per clock, the i7 has many instructions in-flight at once (up to 256, and typically at least 30).
  - Branch penalty limits performance of deep pipelines.
- Want to execute instructions beyond a branch even before that branch is resolved → use *speculative execution*
- What to predict?
  - Answer: The address of the next instruction
  - If the fetched instruction is a non-control-flow instruction:
    - Next Fetch PC is the address of the next-sequential instruction
    - Easy to determine if we know the size of the fetched instruction
  - If the instruction that is fetched is a control-flow instruction:
    - How do we determine the next Fetch PC?
    - In fact, how do we even know whether or not the fetched instruction is a control-flow instruction?

Branch Types

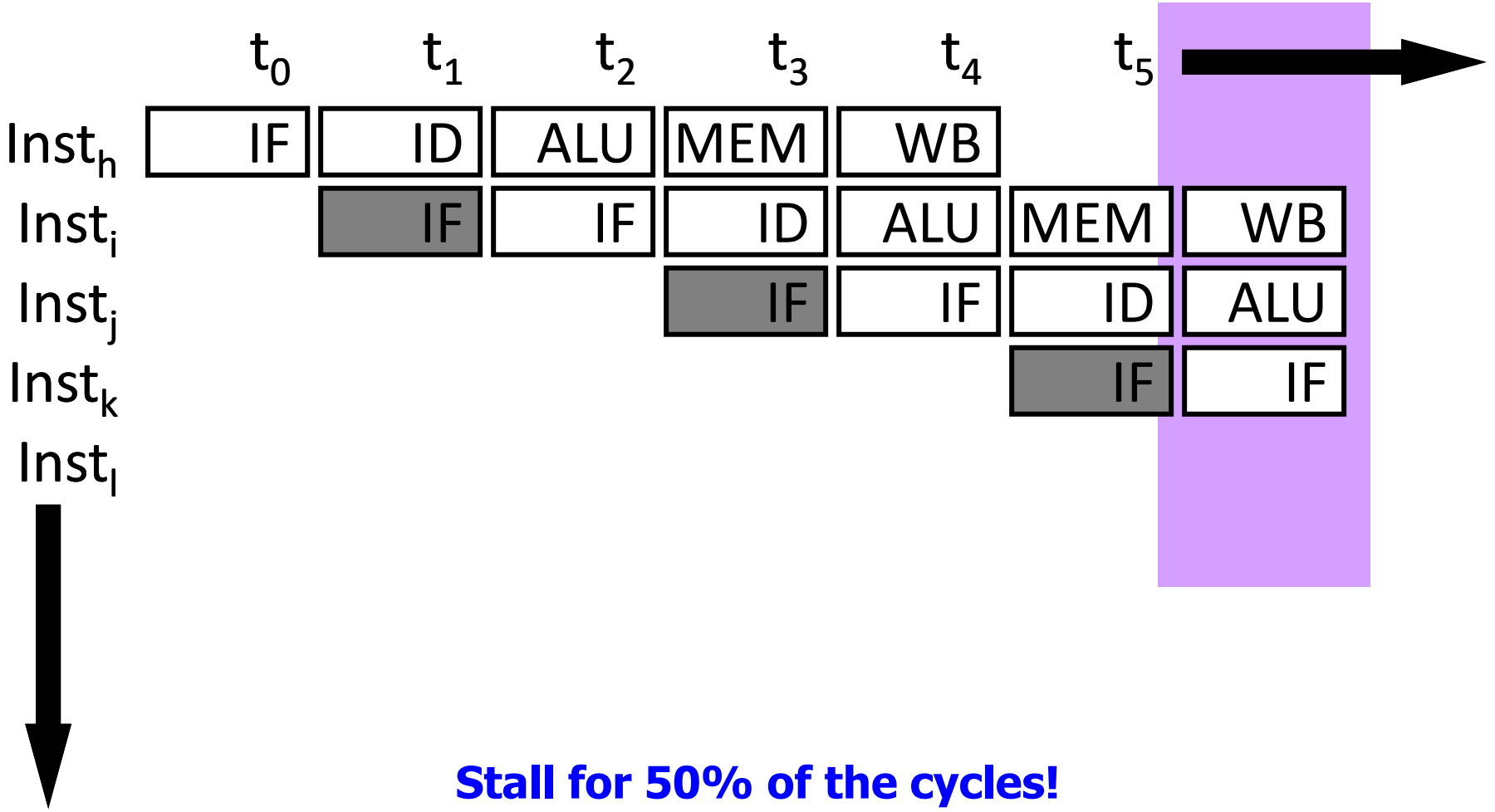
Type	Direction at fetch time	Number of possible next fetch addresses?	When is next fetch address resolved?
Conditional	Unknown	2	Execution (register dependent)
Unconditional	Always taken	1	Decode (PC + offset)
Call	Always taken	1	Decode (PC + offset)
Return	Always taken	Many	Execution (register dependent)
Indirect	Always taken	Many	Execution (register dependent)

Different branch types can be handled differently

How to Handle Control Dependences

- Critical to keep the pipeline full with correct sequence of dynamic instructions.
- Potential solutions if the instruction is a control-flow instruction:
  - Stall the pipeline until we know the next fetch address
  - Always Guess Next PC = PC + 4
  - Reducing the Branch Misprediction Penalty
  - Employ delayed branching (branch delay slot)
  - Eliminate control-flow instructions (predicated execution)
  - Guess the next fetch address (branch prediction)
  - Do something else (fine-grained multithreading)
  - Fetch from both possible paths (if you know the addresses of both possible paths) (multipath execution)

Stall Fetch Until Next PC is Known: Good Idea?



Stall for 50% of the cycles!

This is the case with non-control-flow and unconditional br instructions

Much worse stalling for conditional branch instructions!

Always Guess NextPC = PC + 4

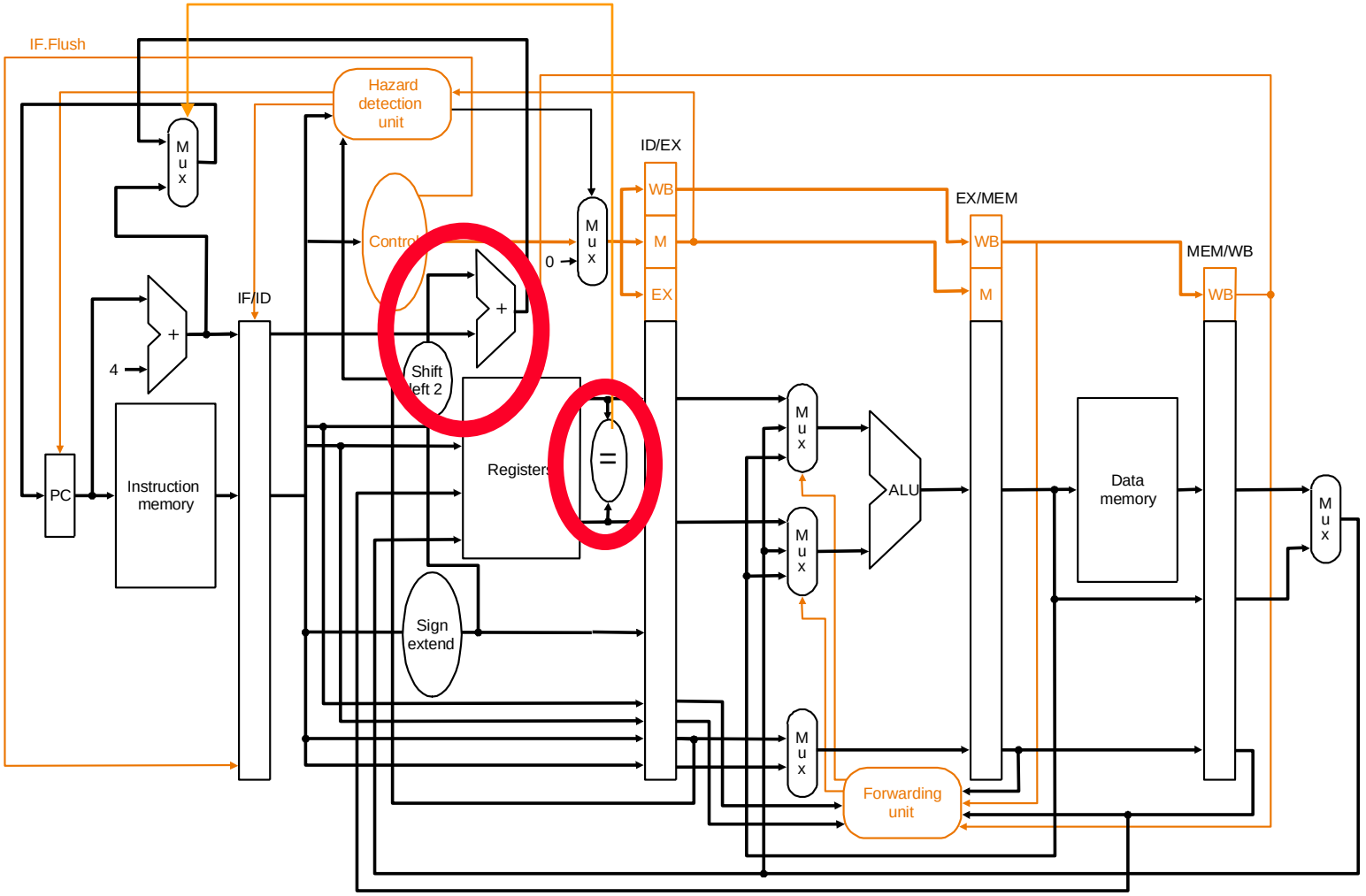
- Always predict the next sequential instruction is the next instruction to be executed
- This is a form of next fetch address prediction (and branch prediction)

- **How can you make this more effective?**

- Idea: Maximize the chances that the next sequential instruction is the next instruction to be executed
  - Software: Lay out the control flow graph such that the "likely next instruction" is on the not-taken path of a branch
    - Profile guided code positioning
  - Hardware: ??? (how can you do this in hardware…)
    - Cache traces of executed instructions → Trace cache

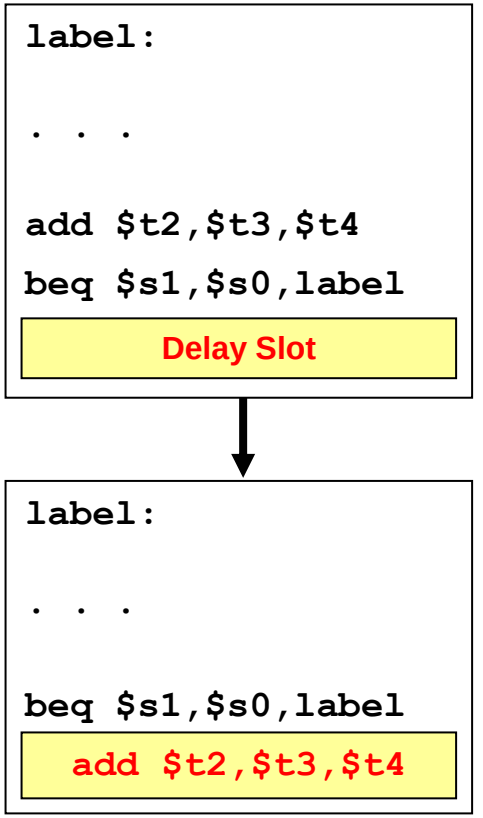
Reducing the Branch Misprediction Penalty

- Resolve branch condition and target address early



Delayed Branch

- Define branch to take place after the next instruction
- MIPS defines one delay slot
  - Reduces branch penalty
- Compiler fills the branch delay slot
  - By selecting an independent instruction from before the branch
  - Must be okay to execute instruction in the delay slot whether branch is taken or not
- If no instruction is found
  - Compiler fills delay slot with a NO-OP



The Branch Problem

- Control flow instructions (branches) are frequent
  - 15-25% of all instructions
- Problem: Next fetch address after a control-flow instruction is not determined after N cycles in a pipelined processor
  - N cycles: (minimum) branch resolution latency
- If we are fetching W instructions per cycle (i.e., if the pipeline is W wide)
  - A branch misprediction leads to **N x W** wasted instruction slots

Branch Prediction (A Bit More Enhanced)

- Idea: **Predict the next fetch address** (to be used in the **next cycle**) → no wasted cycle(s) on correct prediction
- Requires three things to be predicted at fetch stage:
  - Whether the fetched instruction is a branch
  - Direction (1-bit)
    - Single direction for unconditional jumps and calls/returns
    - Binary for conditional branches
  - Target (32-bit or 64-bit addresses)
    - Some are easy
      - One address: uni-directional jumps
      - Two: addresses: fall through (not taken) vs. taken
    - Many: function pointer or indirect jump (e.g. jr r31)
- Observation: Target address remains the same for a conditional direct branch across dynamic instances
  - Idea: Store the target address from previous instance and access it with the PC
  - Called Branch Target Buffer (BTB) or Branch Target Address Cache

Branch Prediction Techniques

- Compile time (static)
  - Always not taken
  - Always taken
  - BTFN (Backward taken, forward not taken)
  - Profile based (likely direction)
  - Program analysis based (likely direction)
- Run time (Dynamic Branch Prediction)
  - Last time prediction (single-bit)
  - Two-bit counter based prediction
  - Two-level prediction (global vs. local)
  - Hybrid

Common disadvantage of compile time methods?

Cannot adapt to dynamic changes in branch behavior

Dynamic Branch Prediction

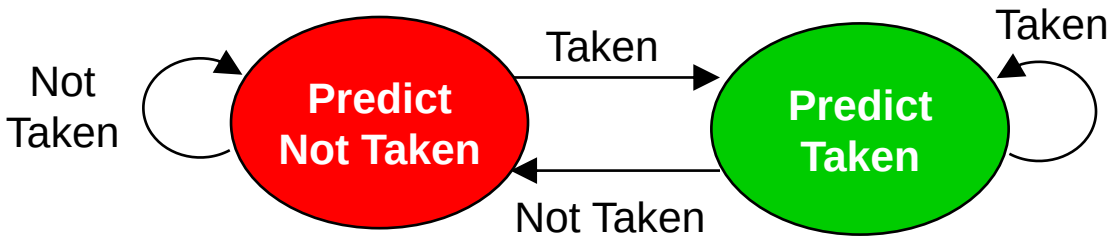
- Idea: Predict branches based on dynamic information (collected at run-time)

- Advantages
  - + Prediction based on history of the execution of branches. It can adapt to dynamic changes in branch behavior
  - + No need for static profiling: input set representativeness problem goes away

- Disadvantages
  - -- More complex (requires additional hardware)

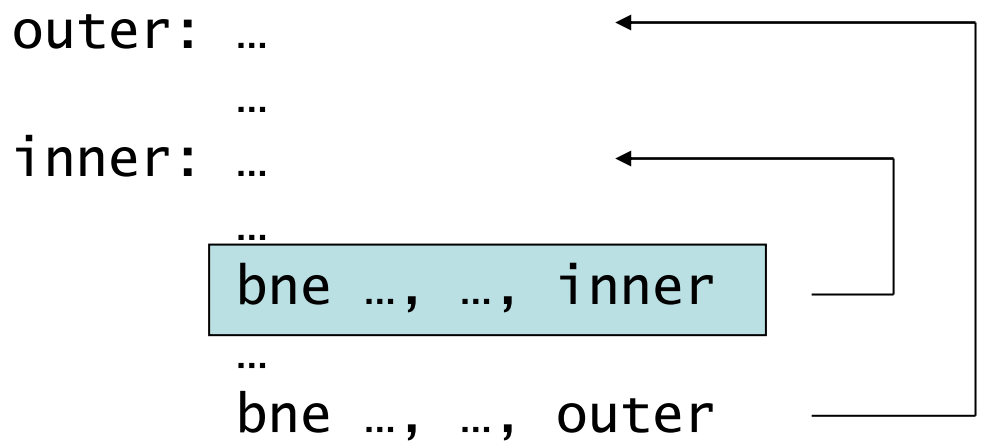
1-bit Dynamic Branch Prediction Scheme

- Prediction is just a hint that is assumed to be correct
- If incorrect then fetched instructions are killed
- 1-bit prediction scheme is simplest to implement
  - 1 bit per branch instruction
  - Record last outcome of a branch instruction (Taken/Not taken)
  - Use last outcome to predict future behavior of a branch



1-Bit Predictor: Shortcoming

- Inner loop branch mispredicted twice!
  - Mispredict as taken on last iteration of inner loop
  - Then mispredict as not taken on first iteration of inner loop next time around



1-Bit Predictor: Example

Assume initially branch is Not Taken

i		0	1	2	3	4
For Br	Pred	NT	T	NT	T	NT
	Act	T	T	T	T	NT
		☠	√	☠	√	√
If Br	Pred	T	T	T	T	
	Act	T	NT	T	NT	
		√	☠	√	☠	

```
For(i=0; a<4; i++) {

    if(a%2==0)
      {.....}
    Else
      {.....}

a++;

}
```

1-Bit Predictor: Example

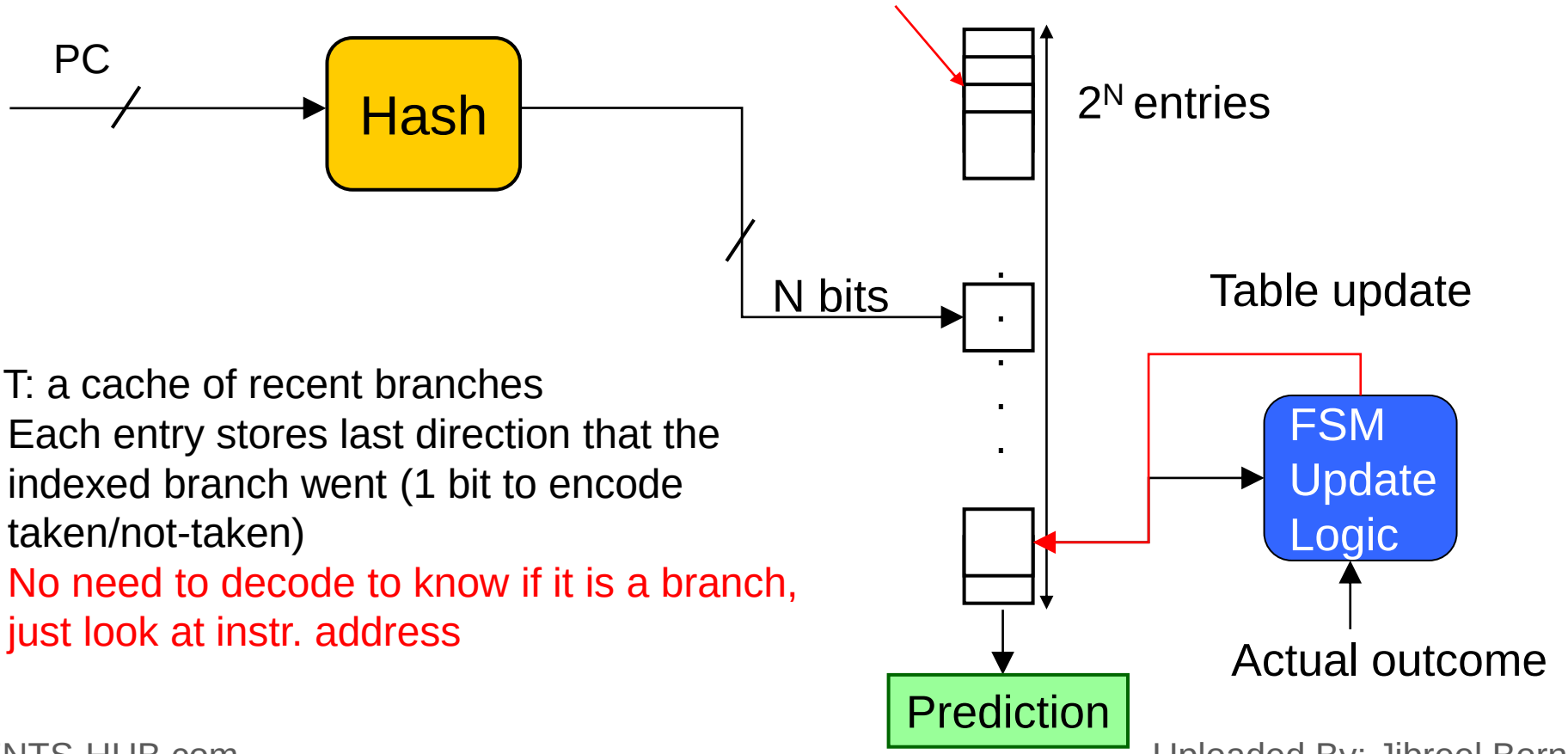
Assume initially branch is Taken

i		0	1	2	3	4
For Br	Pred	T	T	NT	T	NT
	Act	T	T	T	T	NT
		√	√	☠	√	√
If Br	Pred	T	T	T	T	
	Act	T	NT	T	NT	
		√	☠	√	☠	

```
For(i=0; a<4; i++) {

    if(a%2==0)
      {.....}
    Else
      {.....}

a++;

}
```

Dynamic Branch Prediction - One-bit *Branch History Table* (BHT)

- Predict branch based on past history of branch
- One-bit *Branch History Table* (BHT)

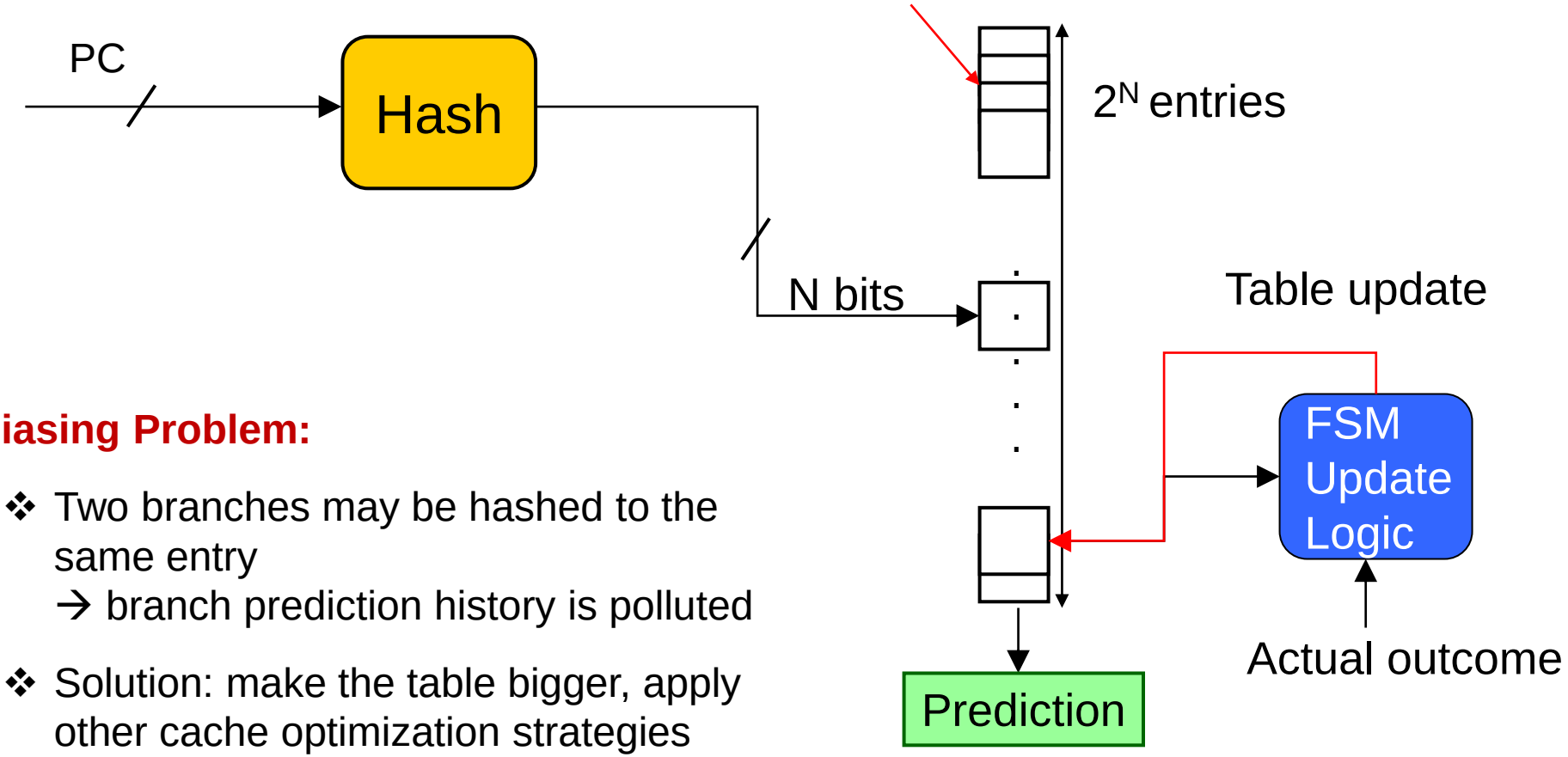


BHT: a cache of recent branches

- Each entry stores last direction that the indexed branch went (1 bit to encode taken/not-taken)
- No need to decode to know if it is a branch, just look at instr. address

Dynamic Branch Prediction - One-bit *Branch History Table* (BHT)

- One-bit *Branch History Table* (BHT)



Aliasing Problem:

- Two branches may be hashed to the same entry
  → branch prediction history is polluted
- Solution: make the table bigger, apply other cache optimization strategies

1-bit *Branch History Table* (BHT) Example

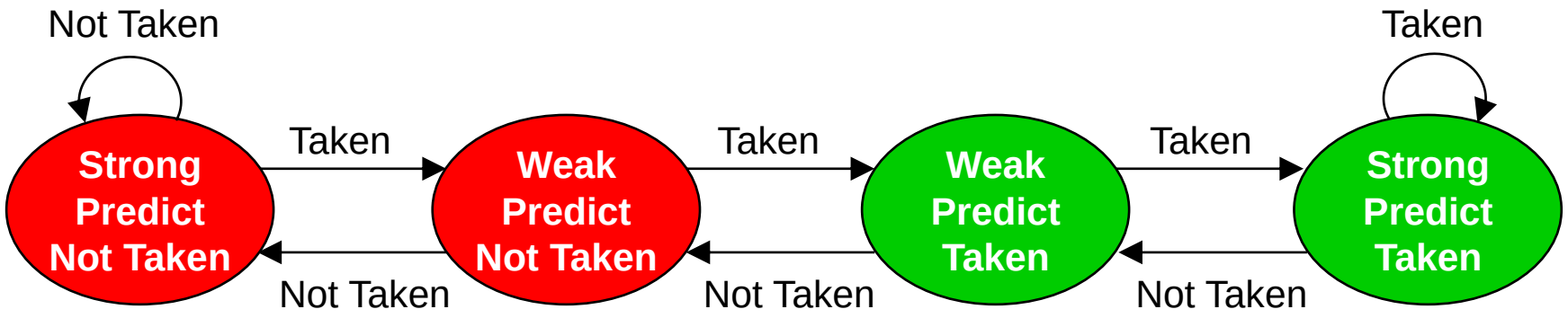
Assume BHT initialized as Taken

i		0	1	2	3	4
For Br	Pred	T	T	T	T	T
	Act	T	T	T	T	NT
		√	√	√	√	☠
If Br	Pred	T	T	NT	T	
	Act	T	NT	T	NT	
		√	☠	☠	☠	

```
For(i=0; a<4; i++) {

    if(a%2==0)
      {.....}
    Else
      {.....}

a++;

}
```

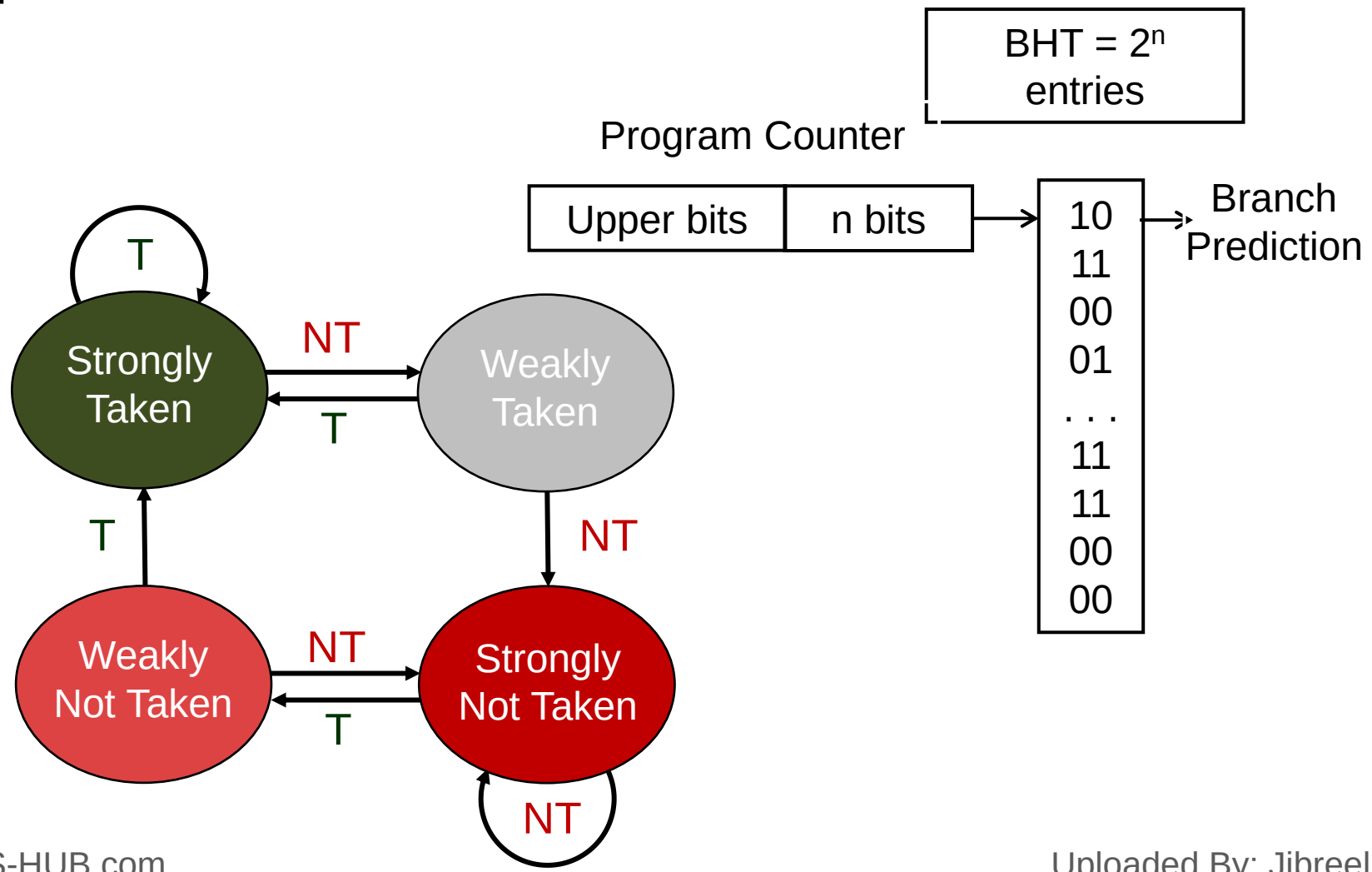
2-bit Prediction Scheme

- 1-bit prediction scheme has a performance shortcoming
- 2-bit prediction scheme works better and is often used
  - 4 states: strong and weak predict taken / predict not taken
- Implemented as a saturating counter
  - Counter is incremented to max=3 when branch outcome is taken
  - Counter is decremented to min=0 when branch is not taken



2-bit Prediction Scheme with BHT

- 2-bit scheme change prediction only if we get two mispredictions



2-bit *Branch History Table* (BHT) Example

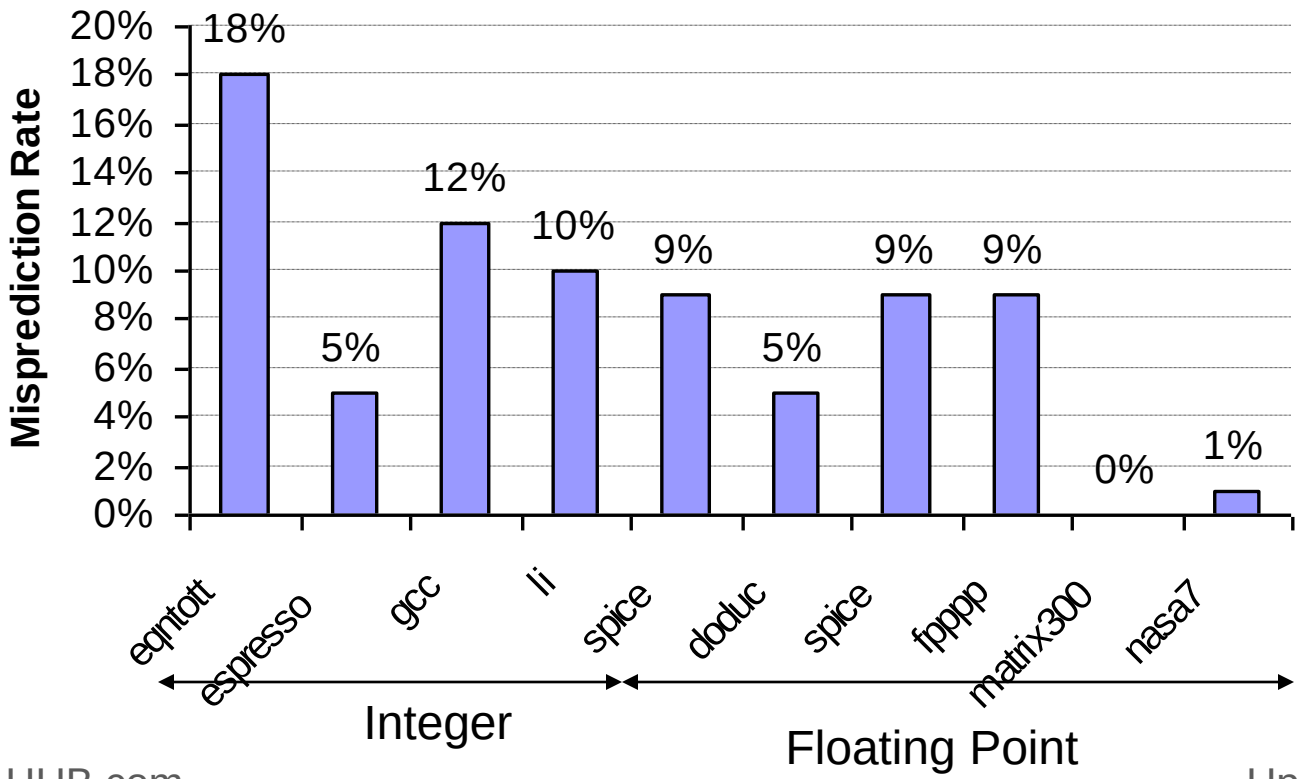
Assume BHT initialized as Weakly Taken (10)

i		0	1	2	3	4
For Br	Pred	(10) T	(11) T	(11) T	(11) T	(11) T
	Act	T	T	T	T	(10) T
		√	√	√	√	☠
If Br	Pred	(10) T	(11) T	(10) T	(11) T	
	Act	T	NT	T	NT	
		√	☠	√	☠	

```
For(i=0; a<4; i++)
{

   if(a%2==0)
     {.....}
   Else
     {.....}

a++;

}
```

2-bit Prediction Scheme with BHT Accuracy

- Mispredict because either:
  - Wrong guess for that branch
  - Got branch history of wrong branch when index the table
- 4096 entry table:



For More Advanced Branch Prediction …

- Hypothesis: recent branches are correlated; that is, behavior of recently executed branches affects prediction of current branch

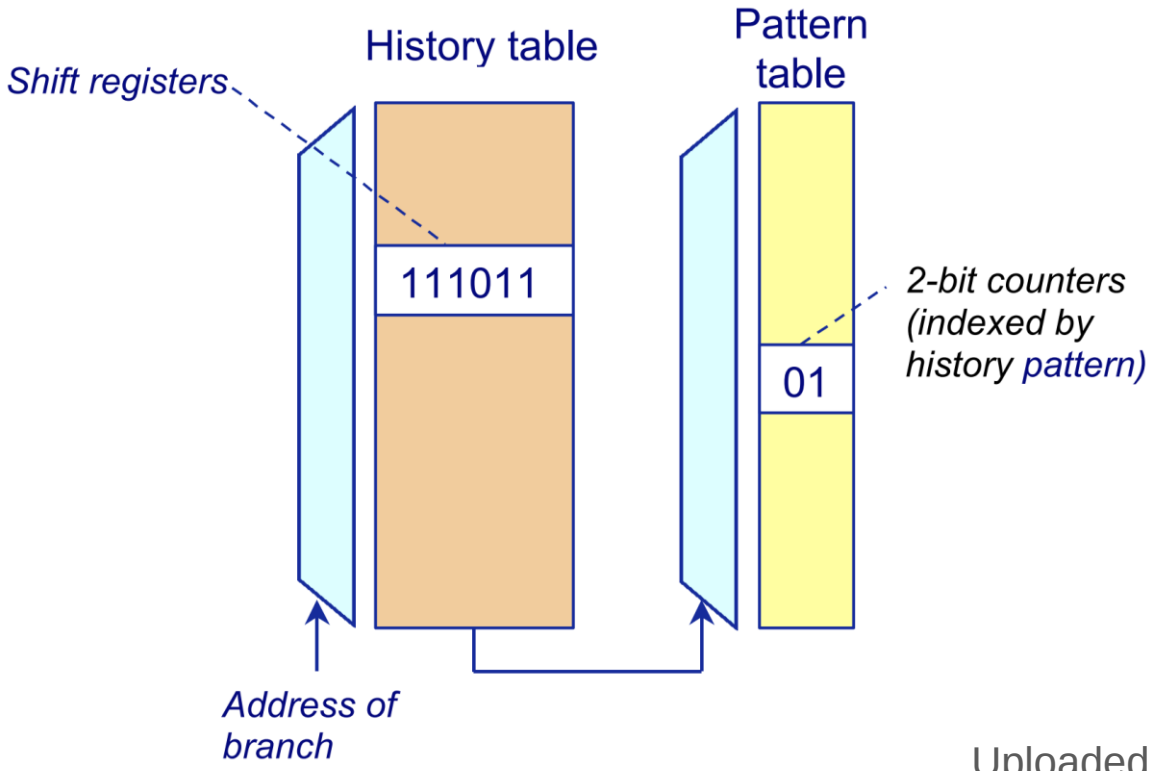
- Two possibilities: current branch depends on
  - Local behavior: Last *m* outcomes of the same branch (*local* branch predictor), e.g., a loop of 3 iterations is executed repetitively

    → a history record of the loop branch of the last 6 iterations should be able to predict the direction of that branch correctly

  - Global behavior: Last *m* most recently executed branches
    → because branches are often *correlated*!

Local Correlation: Yeh-Patt predictor

- It is possible to do quite well considering only information about the current branch (local information).
- We do this by considering, What happened the last time a branch had the same history that the current branch does now?
  - This diagram shows how it operates.

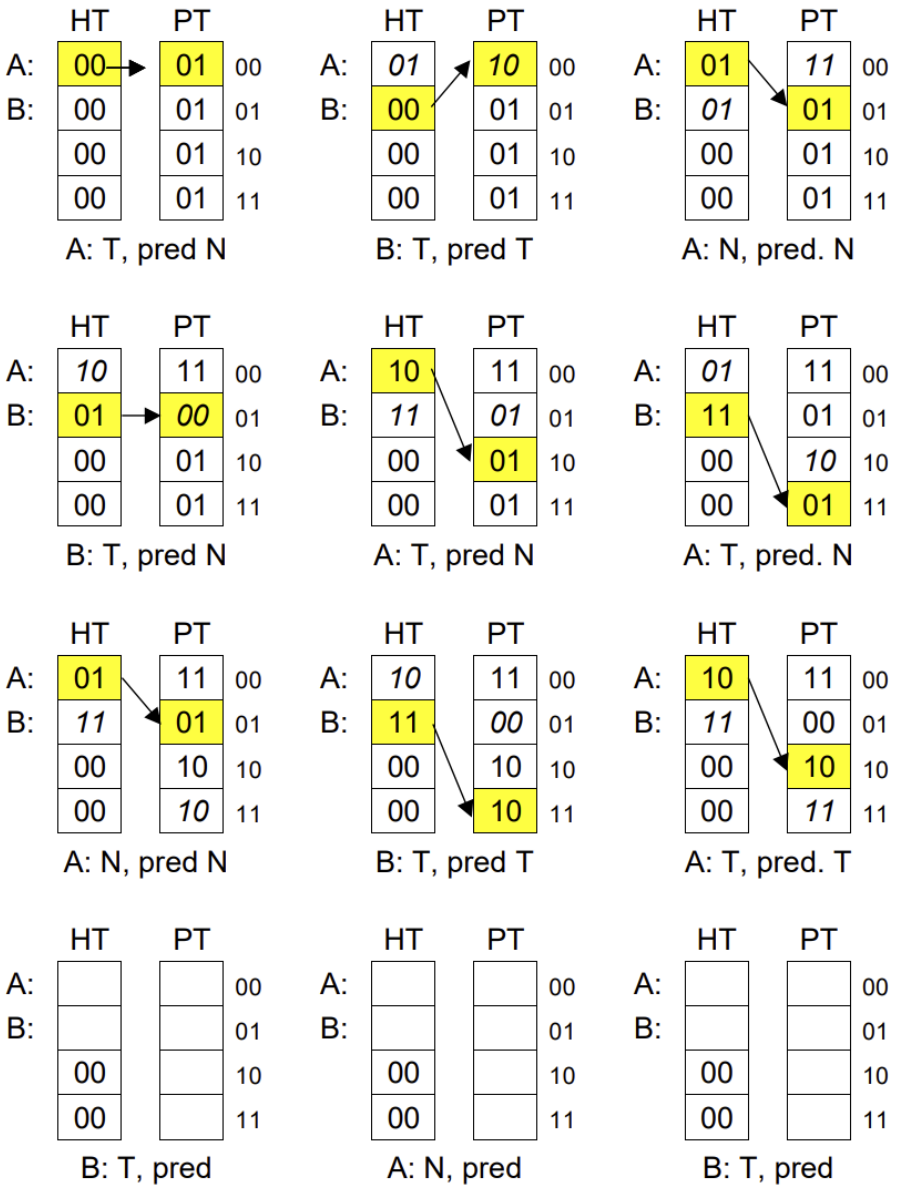


The Yeh-Patt predictor Example

- Let us work out an example, assuming that two branches have this history:

A: T N T N T N T N

B: T T T T T T T T

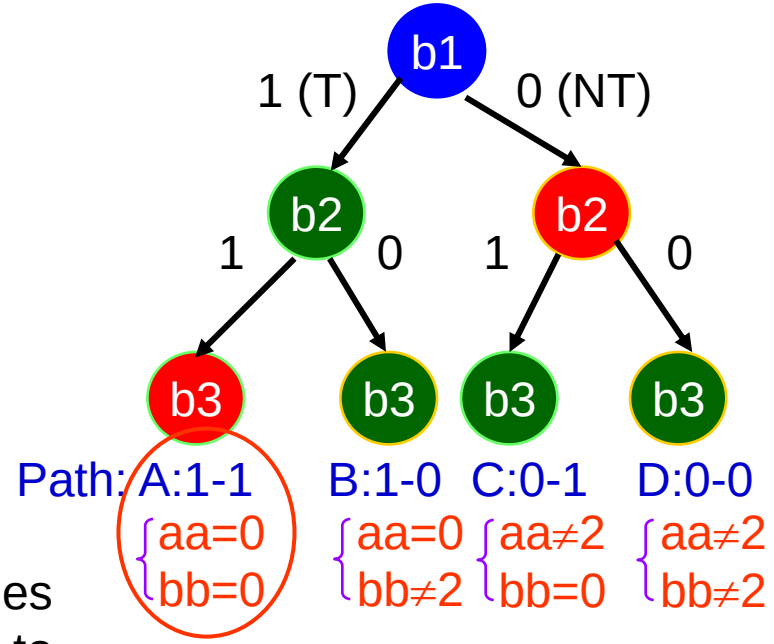


PT entries 01 and 10 are "trained" for A and "trained" for B

Global Correlation

- Branch direction of multiple branches
  - Not independent but correlated to the path taken
- Example: path 1-1 of b3 can be known beforehand

```
if (aa==2)    // b1
   aa = 0;
if (bb==2)    // b2
   bb = 0;
if (aa!=bb) {// b3
        ……
}
```



How to capture global behavior?

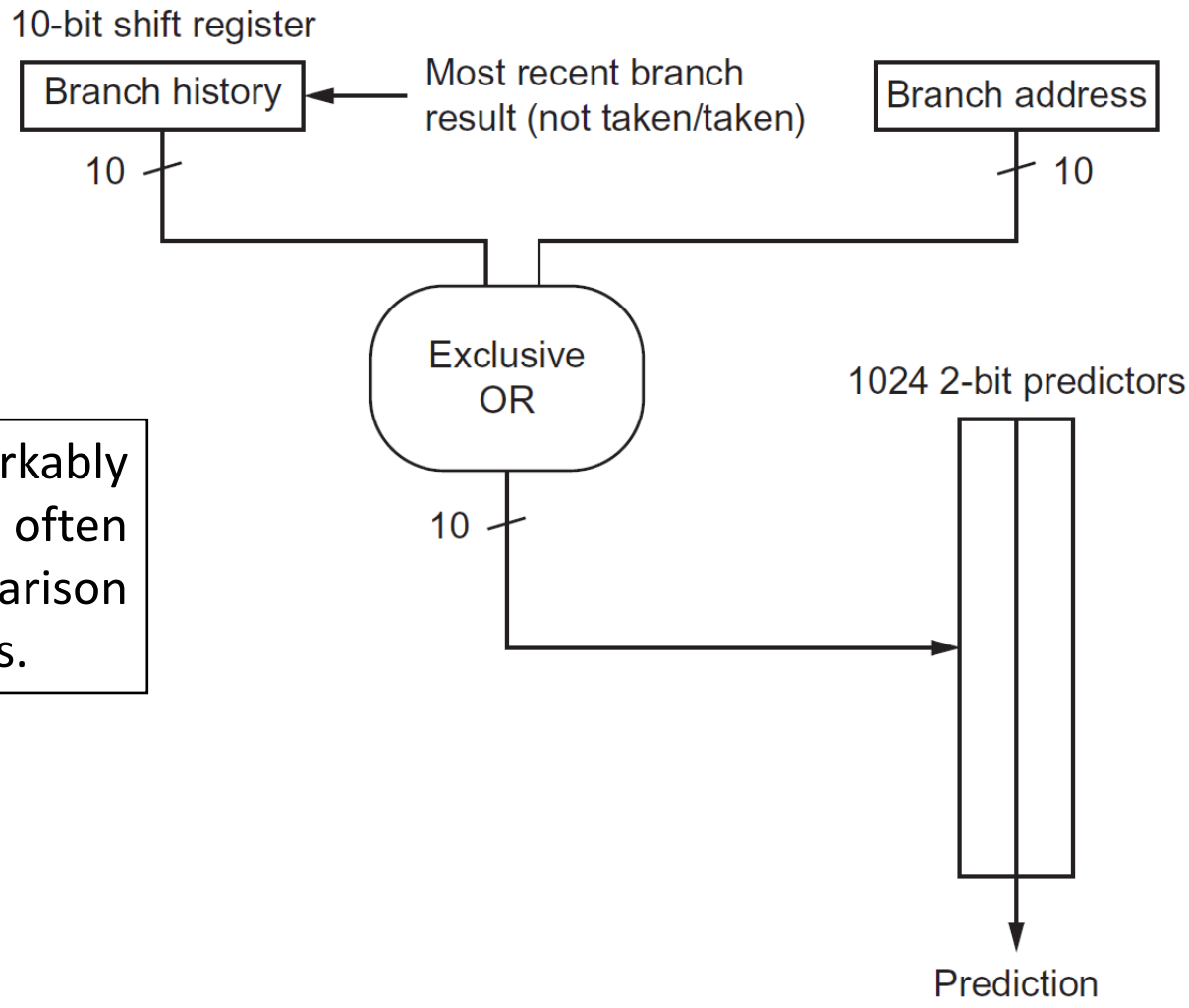
Idea: record *m* most recently executed branches as taken or not taken, and use that pattern to select the proper *n*-bit branch history table

Global Correlation: Gshare Predictor

- The Gshare architecture uses an m-bit global history register to keep track of the direction of the last m executed branches.
- To simplify the implementation, this global history register is xored with the (PC>>2) to create an index into a 2^m-entry pattern history table of n-bit counters.
- The result of this index is the prediction for the current branch.
- The predictor then compares this prediction with the real branch direction to determine if the branch was correctly predicted or not, and updates the prediction statistics.
- The predictor then updates the n-bit counter used to perform the prediction.
- The counter is
  - Incremented if the branch was taken, and
  - Decremented if the branch was not taken.
- Finally, the branch outcome is shifted into the most significant bit of the global history register:

Gshare Predictor Architecture

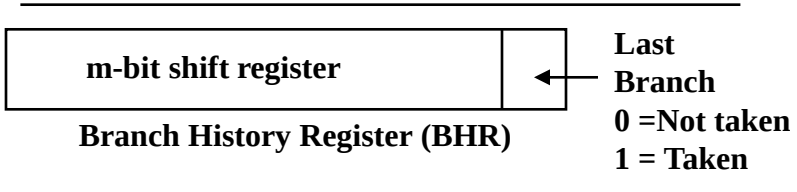
- + Better utilization of PHT
- -- Increases access latency



The Gshare predictor works remarkably well for a simple predictor, and is often used as the baseline for comparison with more sophisticated predictors.

Two-Level Global Adaptive Branch Predictors (GAp)

- Improve branch prediction by looking not only at the history of the branch in question but also at that of other branches using two levels of branch history.
- Uses two levels of branch history:



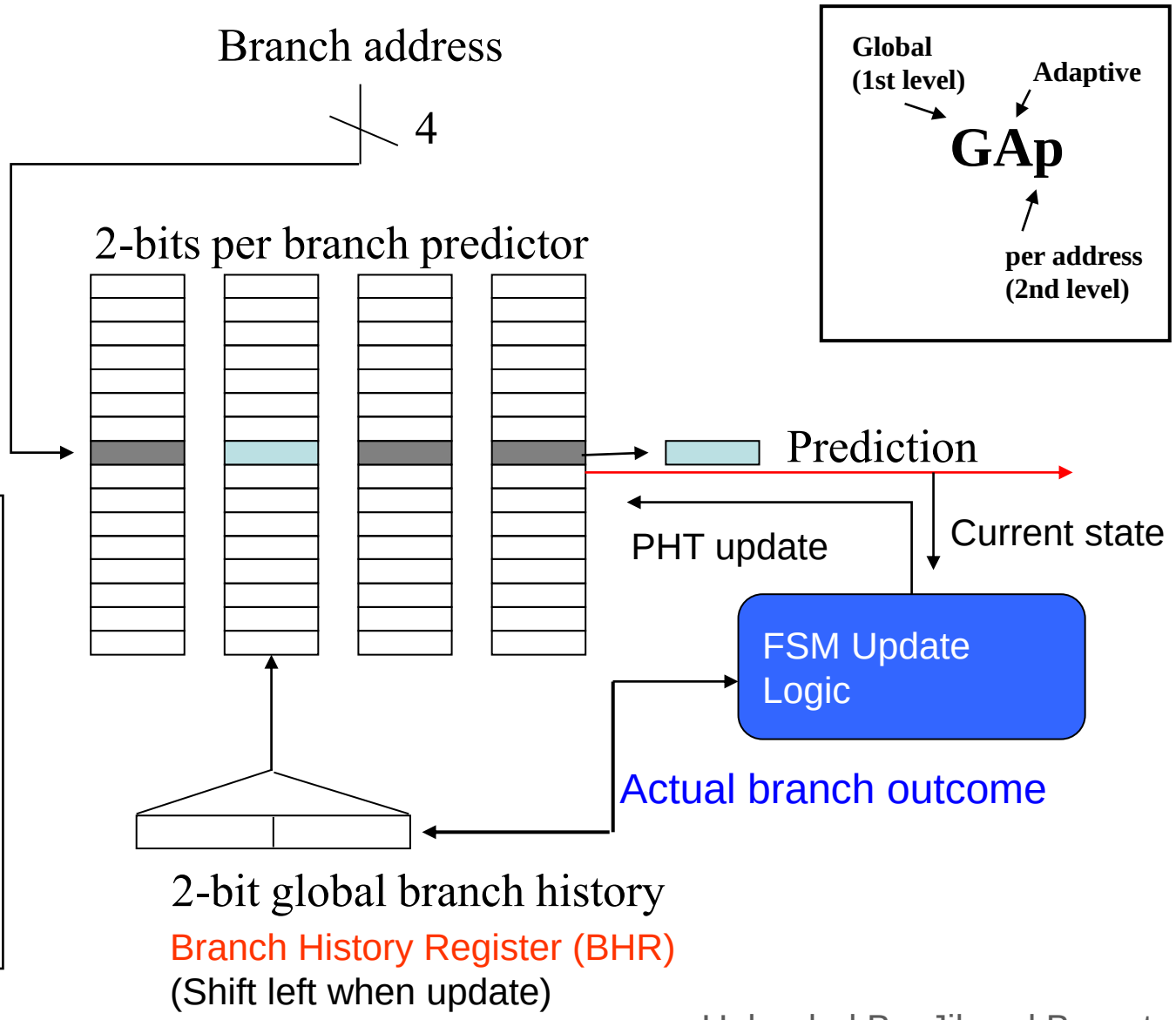
1. **First level (global):**
   - Record the global pattern or history of the m most recently executed branches as taken or not taken. Usually an m-bit shift register.
2. **Second level (per branch address):** Pattern History Tables (PHTs)
   - 2^m prediction tables, each table entry has n bit saturating counter.
   - The branch history pattern from first level is used to select the proper branch prediction table in the second level.
   - The low N bits of the branch address are used to select the correct prediction entry (predictor)within a the selected table, thus each of the 2^m tables has 2^N entries and each entry is 2 bits counter.
   - Total number of bits needed for second level = $2^m \times n \times 2^N$ bits

- In general, the notation: GAp (m,n) predictor means:
  - **Record last m branches to select between 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.

Two-Level Dynamic GAp Architecture Example

GAp (2,2) Predictor

- Behavior of recent branches selects between four predictions of next branch, updating just that prediction



- Two different branches may have the same global branch history but behave differently
- Have a per-branch history register still capture the behavior of the same branch

An Example: GAp(1,1)

```
if (d == 0) d = 1;
if (d == 1) { ... }
        bnez R1, L1        ; b1
        addi R1, R0, #1
L1:     subi R3, R1, #1
        bnez R3, L2        ; b2
        ...
L2:     ...
```

Initial value of d	d==0?	b1	Value of d before b2	d==1?	b2
0	Yes	NT	1	Yes	NT
1	No	T	1	Yes	NT
2	No	T	2	No	T

=> if b1 is NT, then b2 is NT

Behavior of one-bit Standard Predictor initialized to not taken; d alternates between 0 and 2.

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

=> All branches are mispredicted

An Example: GAp(1,1)

- Introduce one bit of correlation
  - Each branch has two separate prediction bits: one prediction assuming the last branch executed was not taken, and another prediction assuming it was taken

Prediction bits	Prediction if last branch NT	Prediction if last branch T
NT/NT	NT	NT
NT/T	NT	T
T/NT	T	NT
T/T	T	T

Behavior of one-bit predictor with one bit of correlation initialized to NT/NT; Assume last branch NT

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

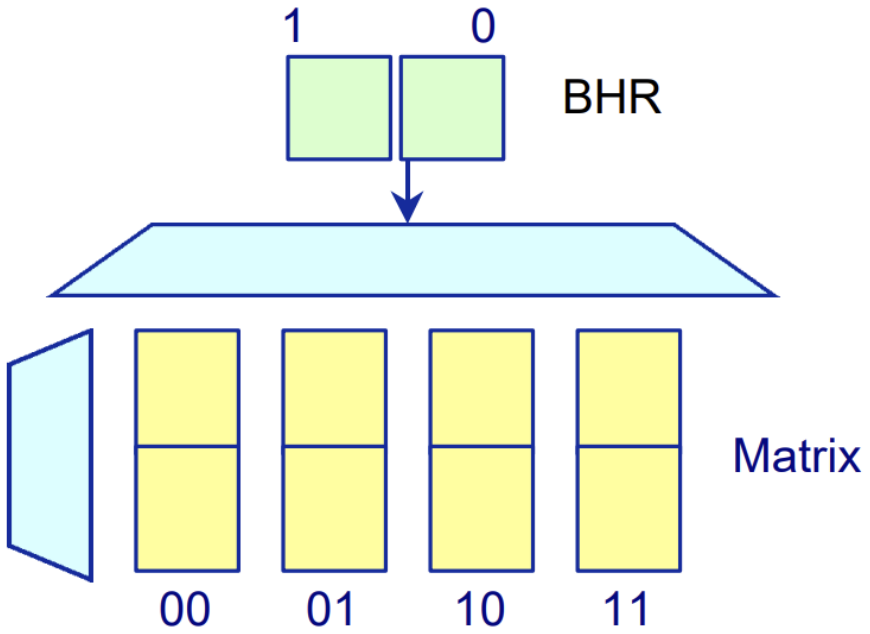
?	NT
b1	T
b2	T
b1	NT
b2	NT
b1	T
b2	T
b1	NT
b2	NT

=> Only misprediction is on the first iteration

An Example: GAp(2,2)

Suppose that we have this code.

```
     Initially, r1 = 0
A:   BEQZ   R1, D
     …
D:   BEQZ   R1, F
     …
F:   NOT    R1, R1
G:   JUMP   A
```



- All entries in the matrix are initialized to 01. In the tables next slide, matrix entries are italicized if they were updated due to the execution of the previous branch. The entry with the yellow background is the entry used for the prediction.

An Example: GAp(2,2)

```
     Initially, r1 = 0
A:   BEQZ    R1, D
     …
D:   BEQZ    R1, F
     …
F:   NOT     R1, R1
G:   JUMP    A
```

BHR = 00

	00	01	10	11
A:	01	01	01	01
D:	01	01	01	01

A: T, pred N
BHR → 01

BHR = 01

	00	01	10	11
A:	*10*	01	01	01
D:	01	01	01	01

D: T, pred N
BHR → 11

BHR = 11

	00	01	10	11
A:	10	01	01	01
D:	01	*10*	01	01

A: N, pred N
BHR → 10

BHR = 10

	00	01	10	11
A:	10	01	01	
D:	01	10	01	01

D: N, pred N
BHR → 00

BHR = 00

	00	01	10	11
A:	10	01	01	00
D:	01	10		01

A: T, pred T
BHR → 01

BHR = 01

	00	01	10	11
A:				
D:	01	10	00	01

D:
BHR →

BHR = 11

	00	01	10	11
A:	10	01	01	00
D:				

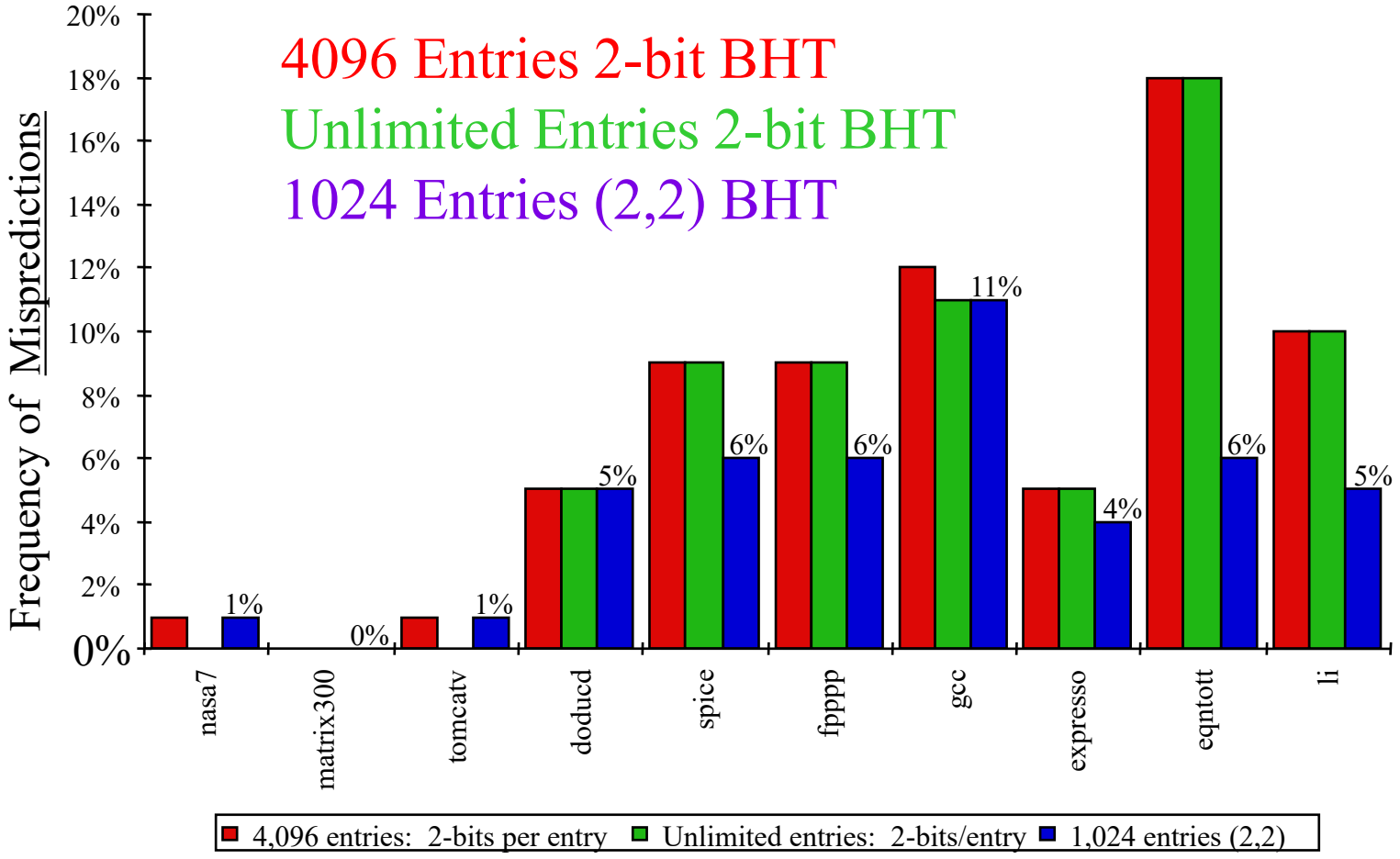
A: N, pred N
BHR → 10

BHR = 10

	00	01	10	11
A:	*11*	01	01	*00*
D:	01	10	00	01

D:
BHR →

Accuracy of Different Schemes



Hybrid Branch Predictors

- Idea: Use more than one type of predictor (i.e., multiple algorithms) and select the "best" prediction
  - E.g., hybrid of 2-bit counters and global predictor

- Advantages:

  + Better accuracy: different predictors are better for different branches

  + Reduced warmup time (faster-warmup predictor used until the slower-warmup predictor warms up)

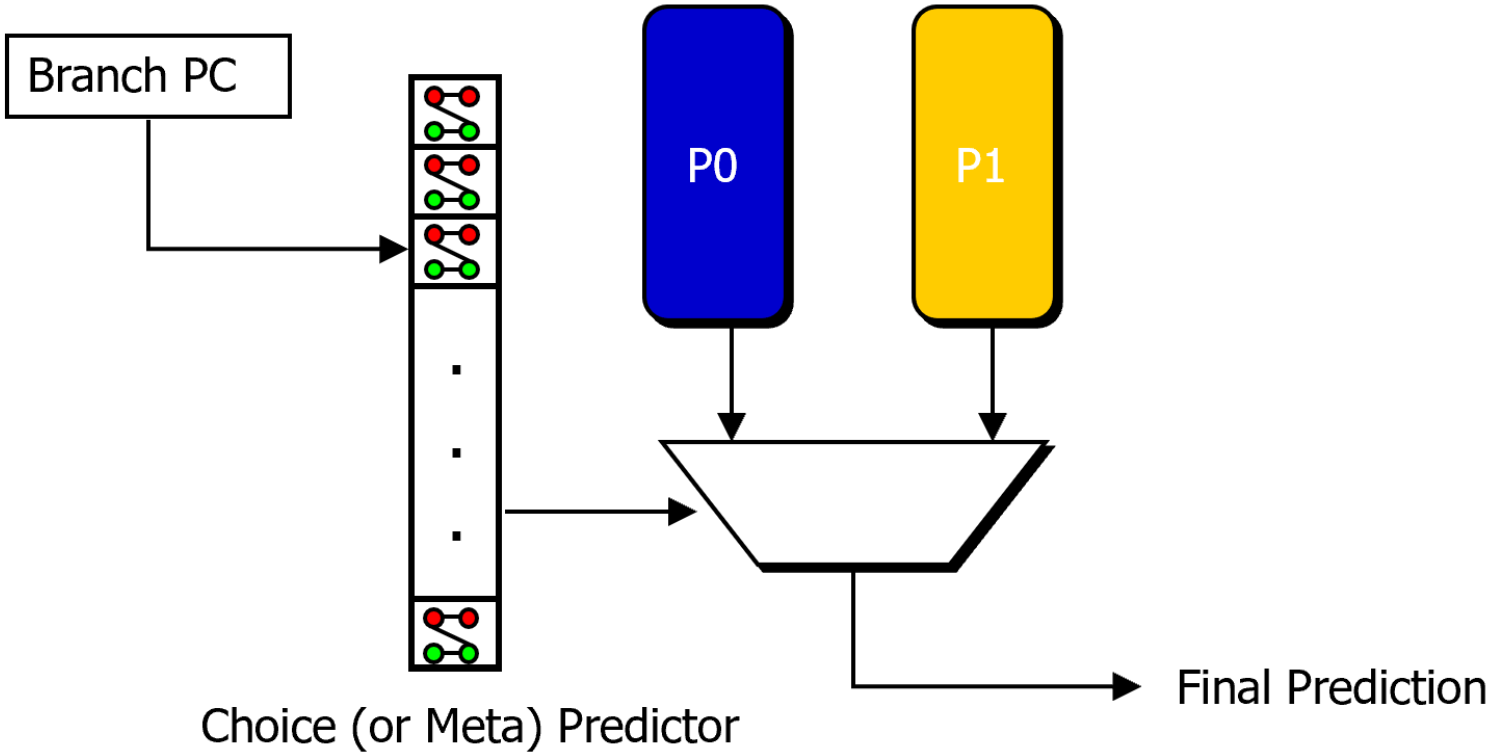
- Disadvantages:

  -- Need "meta-predictor" or "selector" to decide which predictor to use

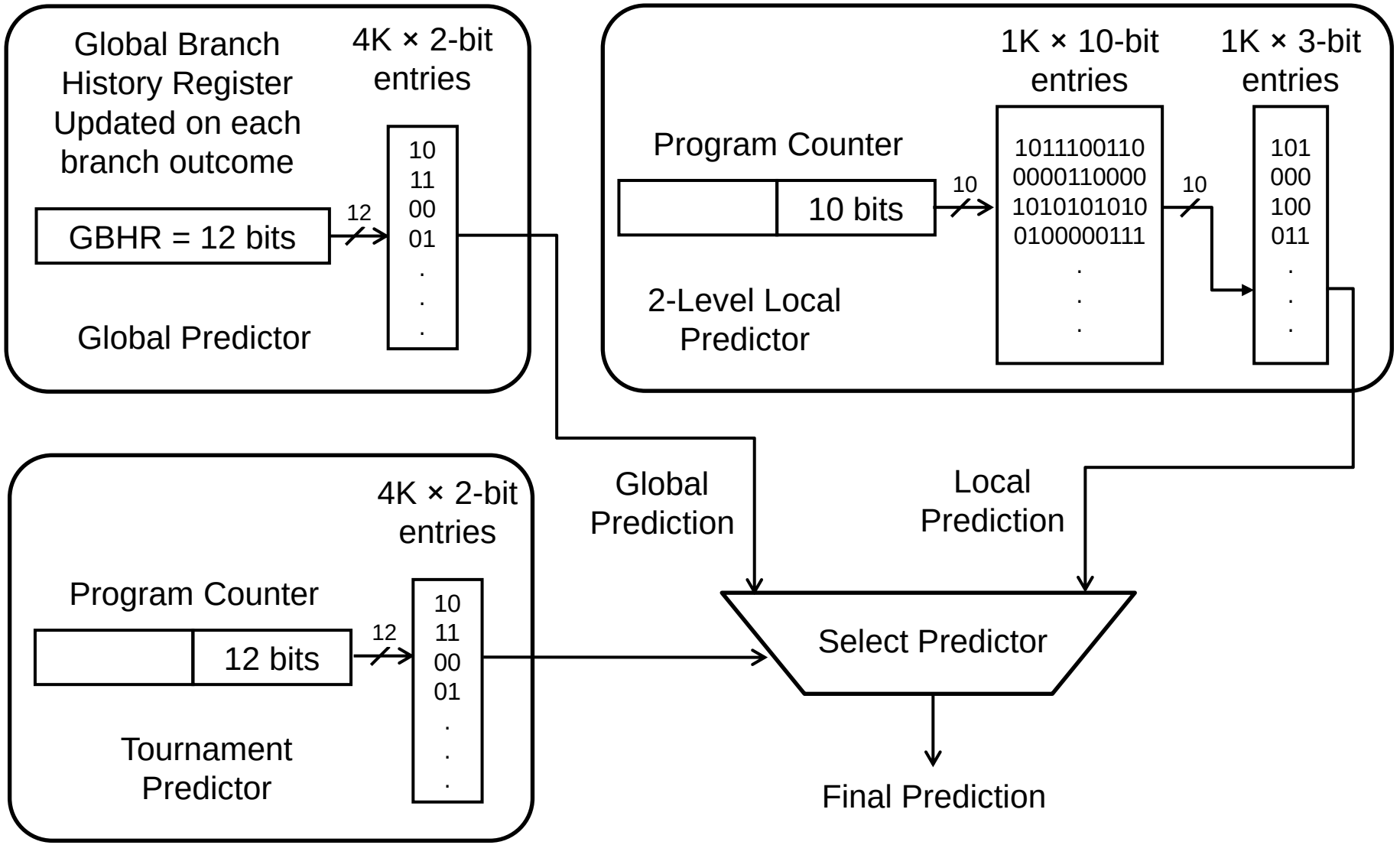
  -- Longer access latency

Hybrid Branch Predictor – Basic Schema

- Some branches correlated to global history, some correlated to local history
  - Use more than one type of predictors and select “best”

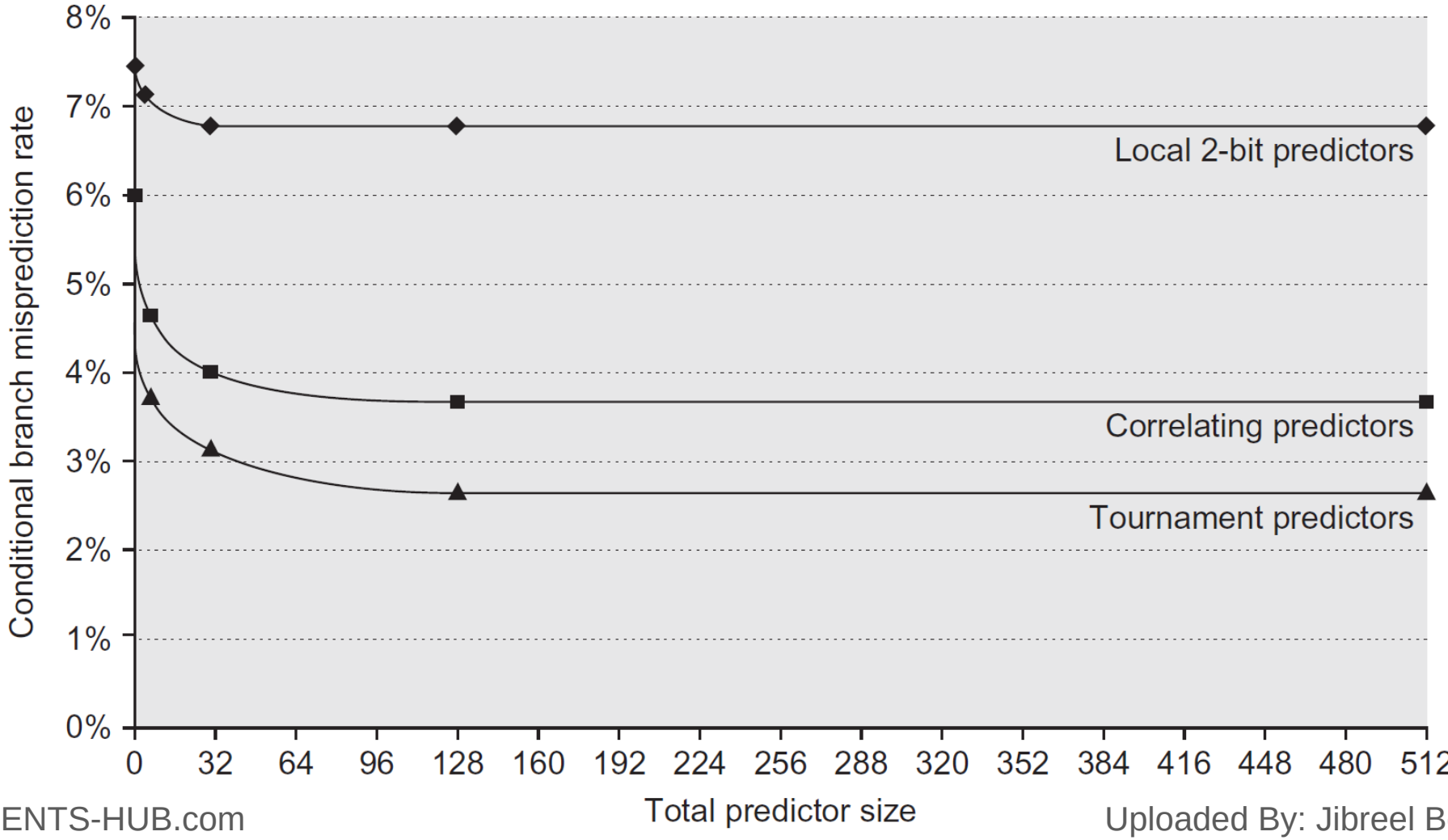


Example: Tournament Predictor



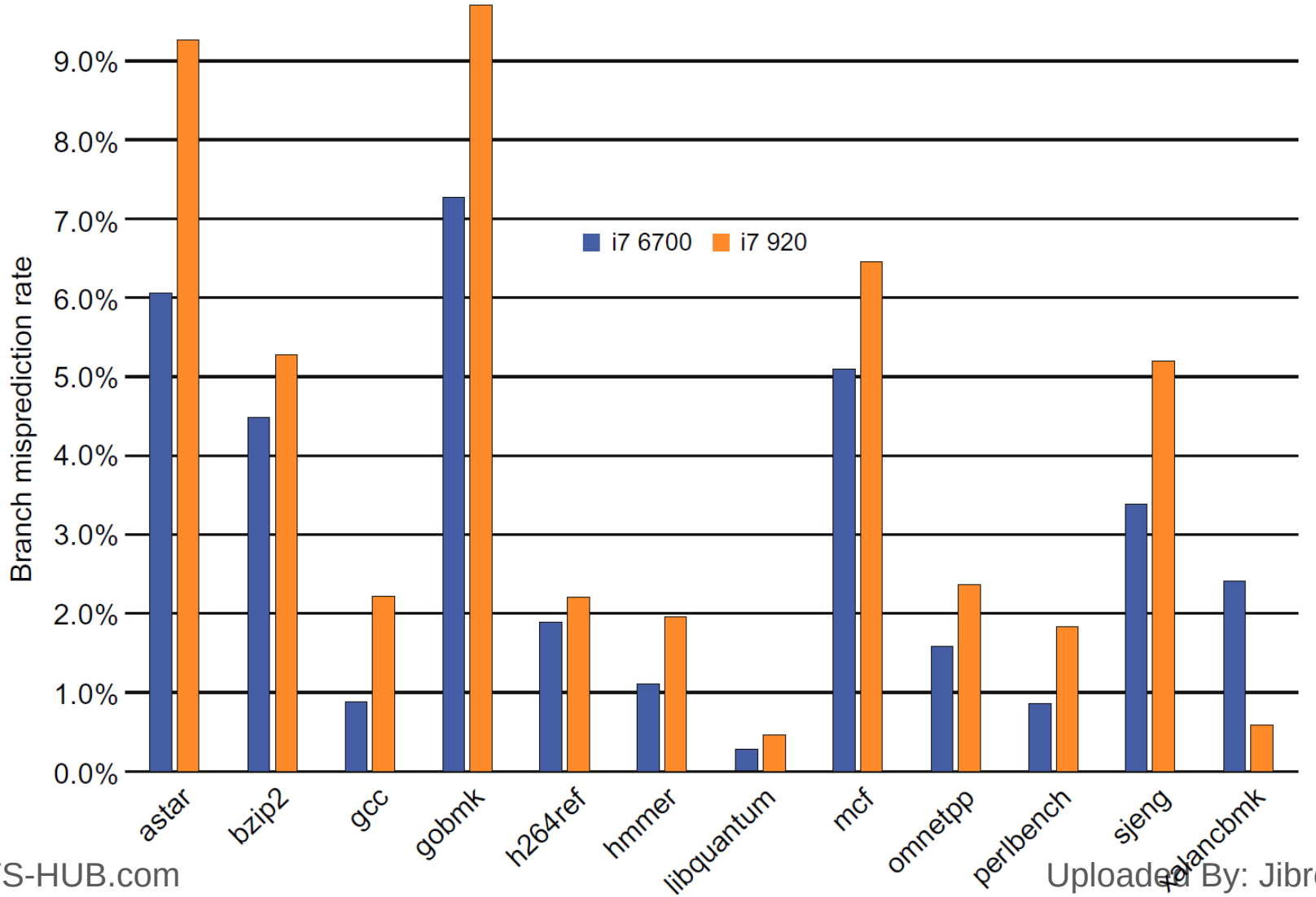
Comparing Accuracy of Different Predictors

The misprediction rate for three different predictors on SPEC89 versus the size of the predictor in kilobits.



Performance of Core i7 920 and core i7 6700

The misprediction rate for the integer SPECCPU2006 benchmarks on the Intel Core i7 920 and 6700.



Zero-Delayed Branching

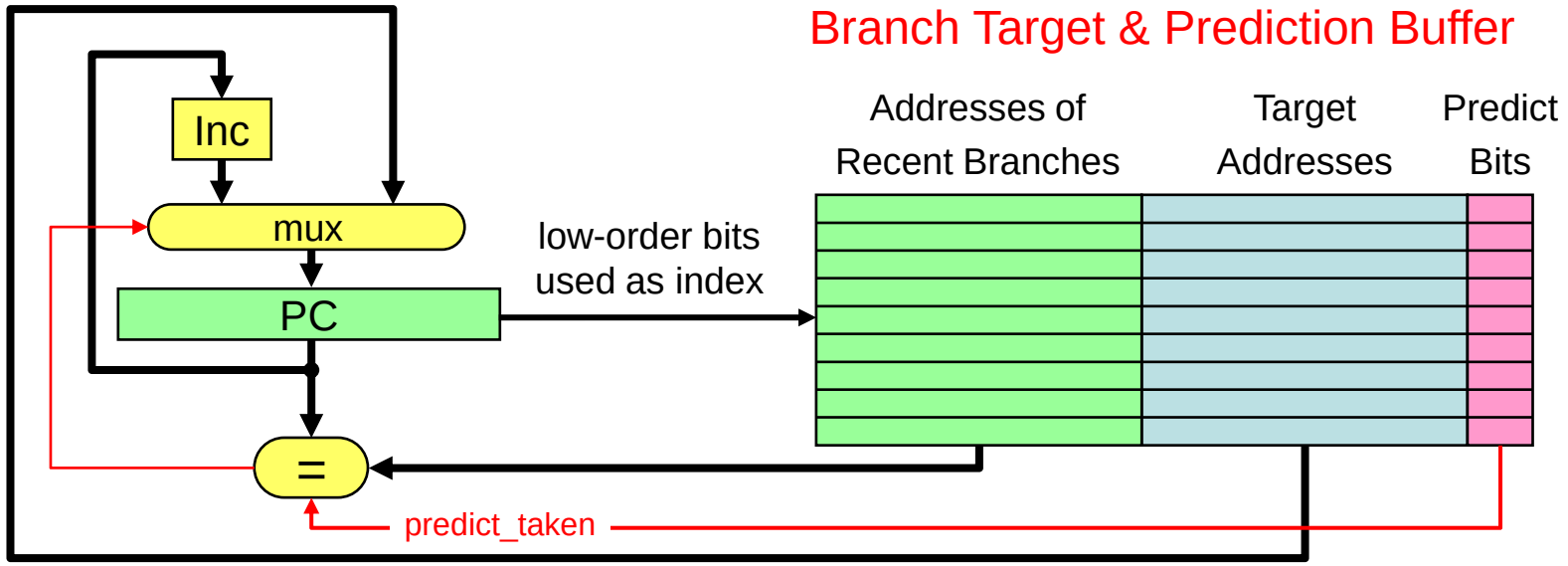
- How to achieve zero delay for a jump or a taken branch?
  - Jump or branch target address is computed in the ID stage
  - Next instruction has already been fetched in the IF stage

Solution

- Introduce a Branch Target Buffer (BTB) in the IF stage
  - Store the target address of recent branch and jump instructions
- Use the lower bits of the PC to index the BTB
  - Each BTB entry stores Branch/Jump address & Target Address
  - Check the PC to see if the instruction being fetched is a branch
  - Update the PC using the target address stored in the BTB

Branch Target Buffer (IF Stage)

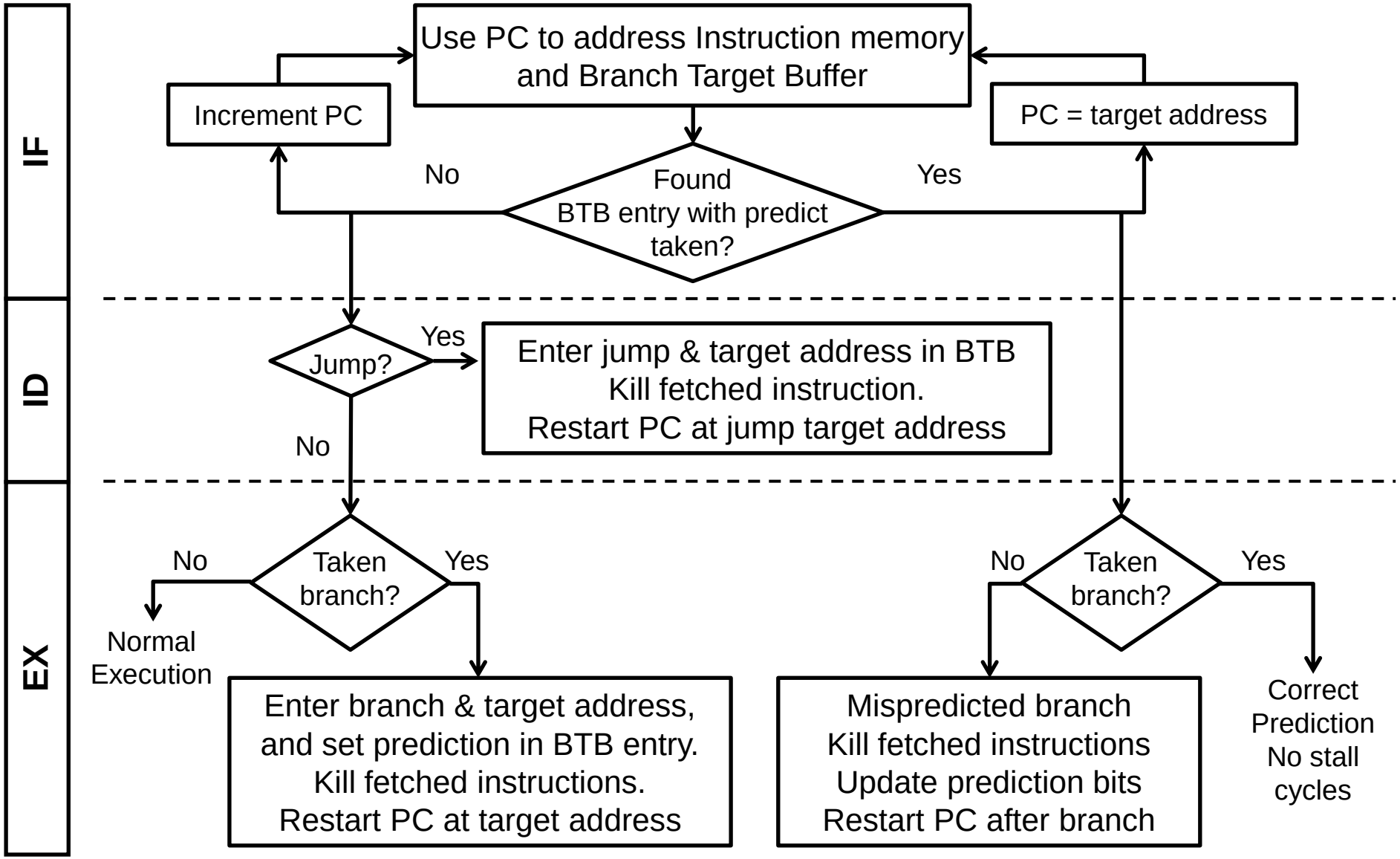
- The branch target buffer is implemented as a small cache
  - Stores the target address of recent branches and jumps
- We must also have prediction bits
  - To predict whether branches are taken or not taken
  - The prediction bits are determined by the hardware at runtime



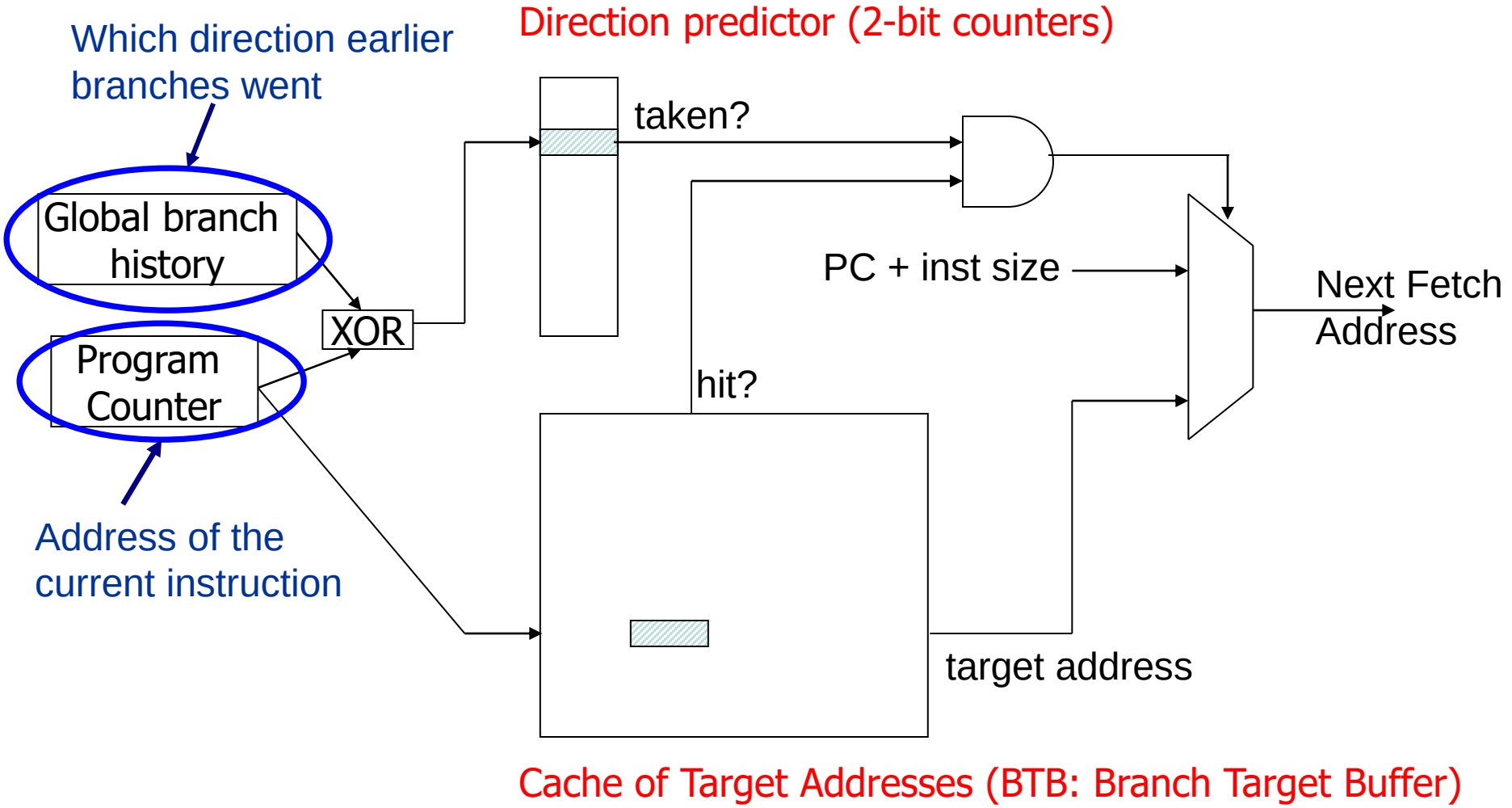
Branch Target Buffer – cont'd

- Each Branch Target Buffer (BTB) entry stores:
  - Address of a recent jump or branch instruction
  - Target address of jump or branch
  - Prediction bits for a conditional branch (Taken or Not Taken)
  - To predict jump/branch target address and branch outcome before instruction is decoded and branch outcome is computed
- Use the lower bits of the PC to index the BTB
  - Check if the PC matches an entry in the BTB (jump or branch)
  - If there is a match and the branch is predicted to be Taken then Update the PC using the target address stored in the BTB
- The BTB entries are updated by the hardware at runtime

Dynamic Branch Prediction with BTB

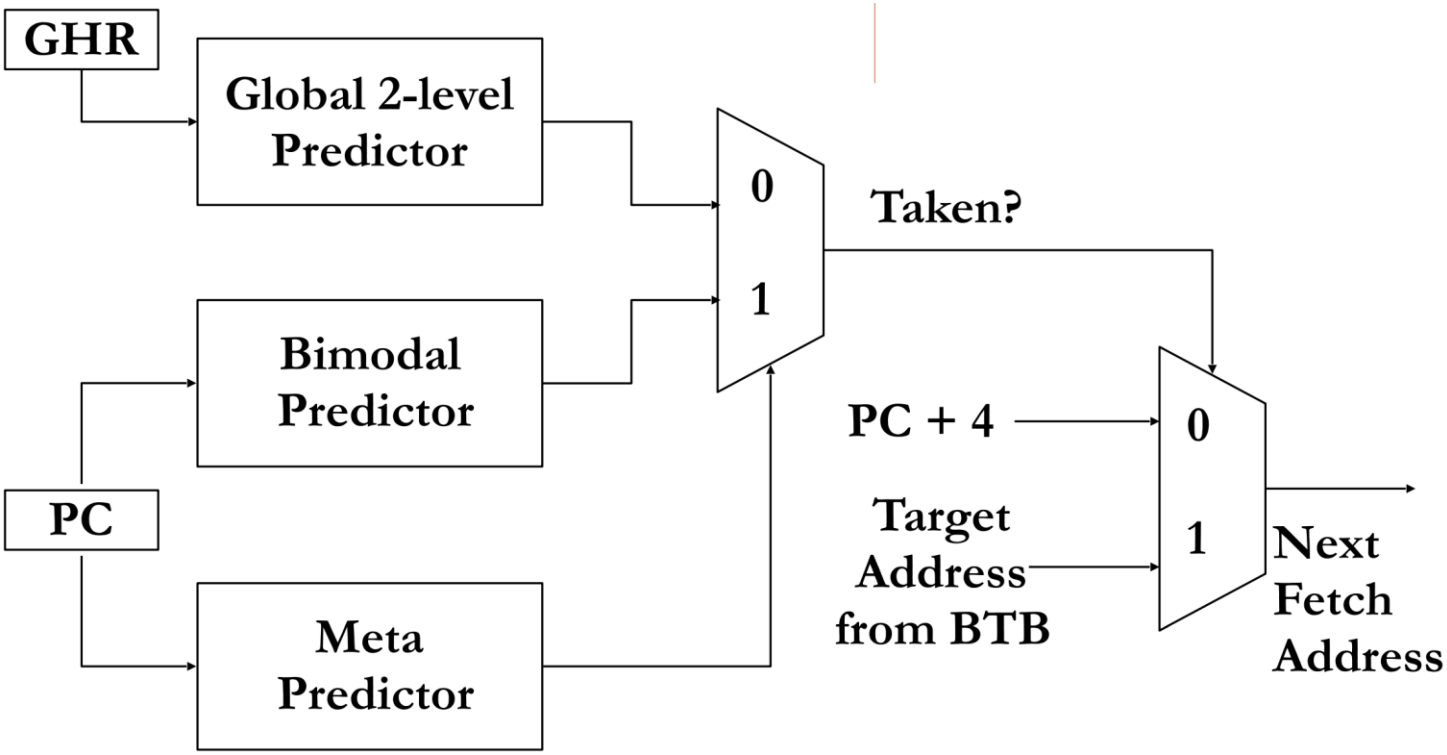


Gshare Predictor with BTB



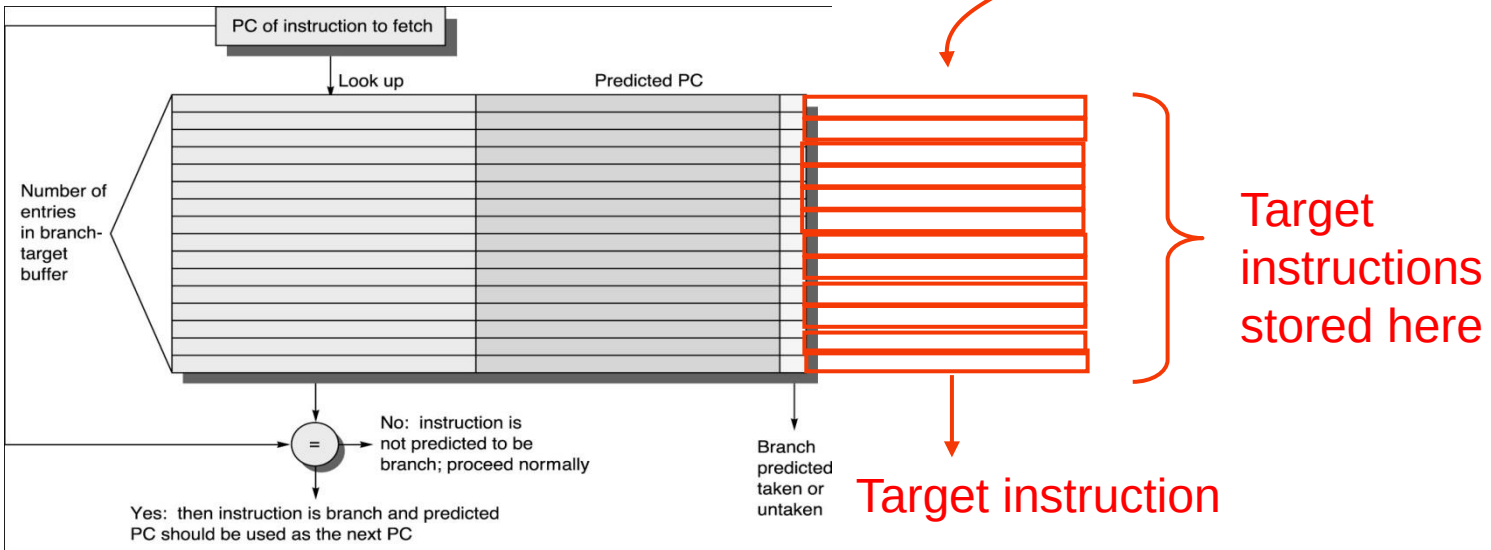
Tournament Predictor with BTB

- Bimodal/Local (2-bit) Branch Predictor
  - + Good for biased branches
  - + No interference
  - - Cannot discern patterns
- Global 2-level Branch Predictor
  - + Leverages correlated branches
  - + Identifies patterns
  - - Cannot always take advantage of biased branches - Interference



Branch Target Cache

- Similar to BTB, but we also want to get the *target instruction*!
  - Prediction returns not just the target address, but also the instruction stored there
  - Allows zero-cycle *unconditional* branches (branch-folding)
    - Send target-instruction to ID rather than branch
    - Branch is not even sent into pipe
    - For conditional branches? Read



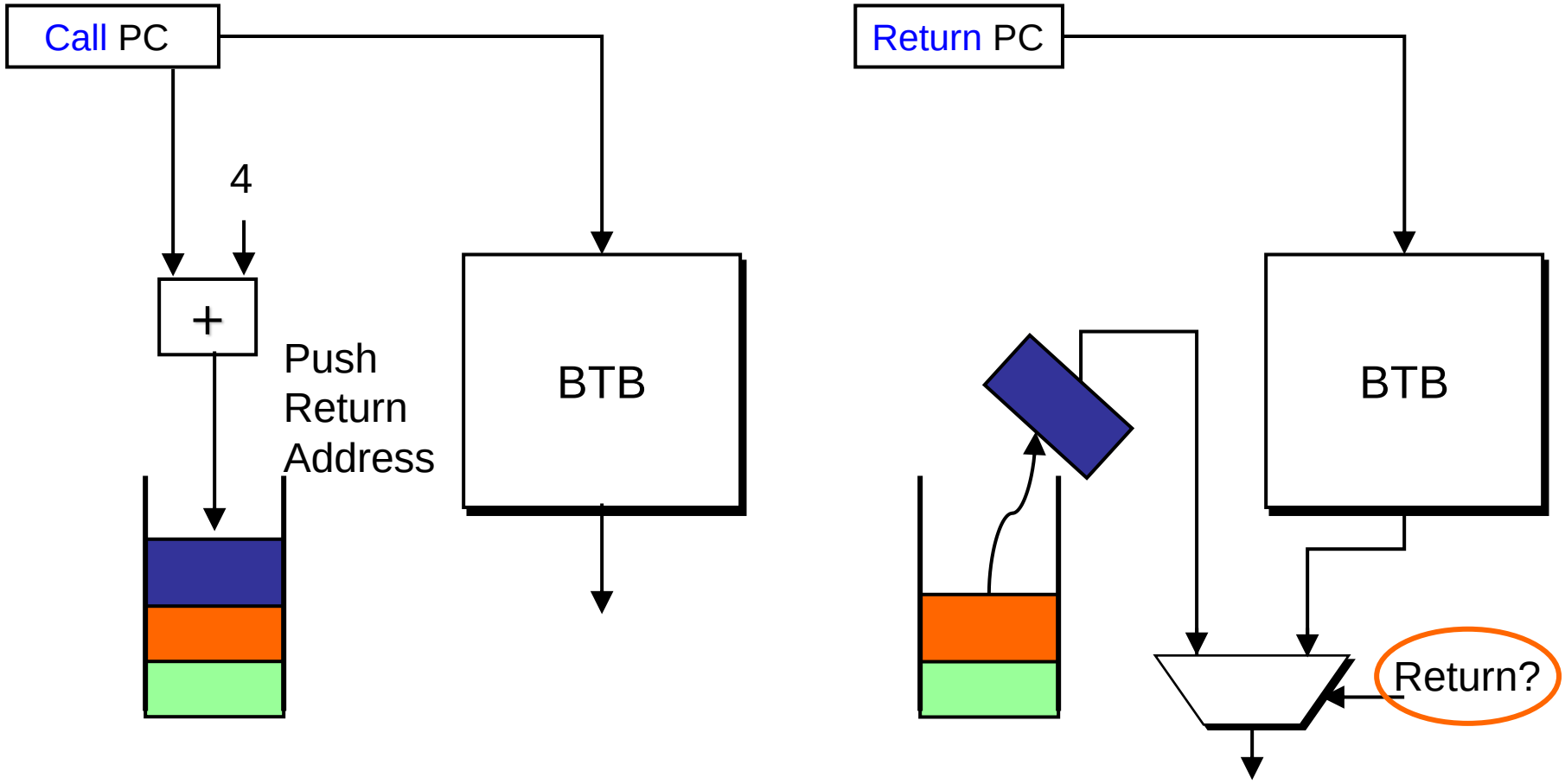
How about Subroutine Returns?

- Different call sites make return address hard to predict
  - printf() may be called by many callers
  - Target of “return” instruction in printf() is a moving target

- But return address is actually easy to predict
  - It is the address after the last call instruction that have not returned from yet
  - Can use a *Return Address Stack* (RAS)

- RAS:
  - Call will push return address on the stack
  - Return uses the prediction of top-of-stack

Return Address Stack



- May not know if it is a return instruction prior to decoding
  - Rely on BTB for speculation
  - Fix once recognize Return