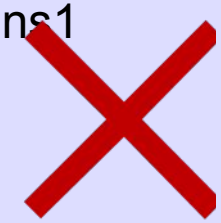
High Performance Computer Architecture (CS60003)

Dept. of Computer Science & Engineering
Indian Institute of Technology Kharagpur

Spring 2020

Branch Prediction

Processor Pipeline Stalls

F	D	A	M	W
JUMP	Add R3,R2,1			
Ins1	JUMP (here processor know it is a jump, but not the target)			
Ins2 	Ins1 	JUMP (here we figure out where it is going)		
Next Correct Instruction-Inst3			JUMP	

This increases the CPI (cycles per instruction)
Note, as the branch is decided on the 3rd stage, 2 extra cycles are spent.

Replaced by pipeline bubbles: pipeline flush

Control Dependency

- Add R1, R1, R2
- BEQ R1, R3, label
- Add R2, R3, R4
- Sub R5, R6, R8
- …
- label: MUL R5, R6, R8

Taken Branch

All statements after a branch have this control dependency, as the processor does not know whether they should be executed.

Quiz 1

- 20% of instructions are branch/jump
- Say 50% of the branch instructions are taken
- Thus, for about 10% of the instructions we would be fetching wrong instructions.
- Assuming, the 5 stage pipeline where the branch/jump is resolved in the 3rd stage along with the target next address, we spend 2 cycles for pipeline stall.
- Thus, CPI=1 + 0.1 $\times$ 2 = 1.2
- For a deeper pipeline, the stage at which we know the branch outcome will be lesser, and hence the branch penalty would be higher

Branch Prediction

- Tries to reduce the factor 0.1 by predicting better the branch outcome.
- We need better predictors more so for modern processors with deeper pipelines.
- Ideally, this will enable to achieve CPI to be closer to 1 for the classical pipelined processor.

Quiz 2

- 25% of all instructions are taken branch/jump
- 10 stage pipeline
- Correct target for branch/jump computed in 6th stage.
- Everything else is flowing through the pipeline stages without stalls/bubbles.
- What is $CPI_{effective}$?

Quiz 2

- 25% of all instructions are taken branch/jump
- 10 stage pipeline
- Correct target for branch/jump computed in 6th stage.
- Everything else is flowing through the pipeline stages without stalls/bubbles.
- $CPI_{effective} = 1 + 0.25 \times 5 = 2.25.$

- **Why do we predict branches at all?**
  - If we do not, then the penalty will always be there. So, we rather have the penalty sometime. It does not help us to not fetch anything after a branch.
- **When do we predict a branch?**
  - When we fetch. We just have obtained the instruction word but have not decoded.
  - So, we don't even know at the fetch stage whether current instruction is a branch.
  - But have to predict whether it is a taken branch!
  - In the next cycle, we have to fetch some instruction based on the address of the branch.

Taken Branch

- Thus, Branch Prediction works on:
  - PC of instruction
    - We need to decide on the PC of the next instruction to be fetched.
  - What does branch prediction mean:
    - Is this a branch?
    - Is it taken?
    - If taken what is the target PC?
  - Both, a non-branch and not taken branch is not an issue.
  - The main objective is to correctly predict the taken branches.

Branch Prediction Accuracy and Pipeline

Accuracy of Predictions

Roughly depends on the stage where we detect misprediction

- $$CPI = 1 + \frac{\#MISPREDICTIONS}{\#INSTRUCTION} \times \frac{Clock\ Cycles\ (PENALTY)}{\#MISPREDICTION}$$

Cycles on average per instruction that we add because of misprediction.

Branch Prediction Accuracy and Pipeline

20% of all instructions are branches.

Accuracy	Resolve Branch Prediction in 3rd Stage	Resolve Branch Prediction in 10th Stage
50% for Branch Prediction		
90% for Branch Prediction		
SpeedUp?		

Fill up the CPIs and the SpeedUps?

Branch Prediction Accuracy and Pipeline

20% of all instructions are branches.

Accuracy	Resolve Branch Prediction in 3rd Stage	Resolve Branch Prediction in 10th Stage
50% for Branch Prediction	1+0.2x0.5x2=1.2	1+0.2x0.50x9=1.9
90% for Branch Prediction	1+0.2x0.1x2=1.04	1+0.1x0.2x9
SpeedUp?	1.15	1.6

- Better branch prediction helps to improve CPI and reduce the penalty because of a branch miss.
- However, speedup depends on the number of pipeline stages.
- Motivates study for better branch predictors, due to increase in pipeline stages in modern processors.

Quiz 3

- 5 stage pipeline
- Branch resolved in 3rd stage
- Fetch nothing until sure what to fetch.
- Execute many iterations of:

```
LOOP: ADDI R1, R1, -1
      ADD R2, R2, R2
      BNEZ R1, LOOP
```

Compute, the number of cycles.

Quiz 3

```
LOOP: ADDI R1, R1, -1
      ADD R2, R2, R2
      BNEZ R1, LOOP
```

ADDI: When ADDI is fetched we do not know if it is a branch.
It is decoded in the 2nd stage.
So, 2 cycles are needed after which the next instruction is fetched.

ADD: Similarly, 2 cycles

BNEZ: The target address and what to fetch next is decided only after 3rd stage. So, we need 3 cycles.

Hence, overall 7 cycles are needed per loop to complete the fetches.

F	R	A	D	W
ADDI				
	ADDI			
ADD				
	ADD			
BNEZ				
	BNEZ			
		BNEZ		

Quiz 4

- 5 stage pipeline
- Branch resolved in 3rd stage
- Consider a perfect predictor: For every address we know which instruction is to be executed next.
- Execute many iterations of:

```
LOOP: ADDI R1, R1, -1
      ADD R2, R2, R2
      BNEZ R1, LOOP
```

Compute, the number of cycles.

Quiz 4

```
LOOP: ADDI R1, R1, -1
      ADD R2, R2, R2
      BNEZ R1, LOOP
```

With a perfect predictor we know next instructions magically!

Hence, overall 3 cycles are needed per loop to complete the fetches.

SpeedUp=7/3=2.33

F	R	A	D	W
ADDI				
ADD	ADDI			
BNEZ	ADD	ADDI		

Not Taken Prediction

- Simply fetch the next instruction as if nothing is a taken branch!
- Can you compare the performance with this simple prediction with the “Refuse to predict strategy”?

Refuse to predict strategy	Predict Not-taken
Branch: 3 cycles Non-branch: 2 cycles	Branch: 1 (Not Taken), 3 (Taken) Non-branch: 1

Thus, Predict Not-taken always is better strategy!
All processors will thus have some form of branch prediction, even if it as simple as Not-taken Prediction.
It just means increment PC, which the processor has to do anyway…

Quiz 5

```
BNE R1, R2, LABELA (T)
BNE R1, R3, LABELB (T)
A
B
C
LABEL A:
X
Y
LABEL B:
Z
```

Using Not-taken predictor how many cycles are wasted on mispredictions until we get to Y?

Quiz 5

```
BNE R1, R2, LABELA (T)
BNE R1, R3, LABELB (T)
A
B
C
LABEL A:
X
Y
LABEL B:
Z
```

Using Not-taken predictor how many cycles are wasted on mispredictions until we get to Y?

We mispredicted 2 branches, but the 2nd branch was in the shadow of the first.
We donot pay extra cycles for the 2nd branch which was itself wrongly fetched.

Quiz 5

BNE R1, R2, LABELA (T)
BNE R1, R3, LABELB (T)
A
B
C
LABEL A:
X
Y
LABEL B:
Z

Using Not-taken predictor how many cycles are wasted on mispredictions until we get to Y?

F	R	A	D	W
BNE1				
BNE2	BNE1			
A	BNE2	BNE1 (taken!)		
X				
Y	X			

So, extra 2 cycles are needed.

Branch Prediction (Contd.)

25.1.2021

Branch Prediction in Computers (Recap)

Simple Branch Instruction:

BEQ: R1, R2, Imm

R1!=R2, PC++

R1=R2, PC+4+Imm

Fetch	Read	ALU	MEM	WRITE BACK
BEQ				
??	BEQ			
??	??	BEQ		
✔	x	x	BEQ	

2 cycles for mis-prediction!
Branch prediction is based on the address only!

Predict Not-taken

- Implementation requires only to increment the PC
- 20% of instructions are branches
- 60% of instructions are taken=>40% are not taken
- So, % of correct prediction:

  80+20x0.40=88%
- Incorrect %=12
- CPI=1+0.12xPENALTY=1+0.12x2=1.24

Quiz 6

- Fill up the table assuming the same statistics:

	Not Taken	Better (99% accurate)	SpeedUp
5 Stage Pipeline (3rd Stage)			
14 Stage Pipeline (11th stage)			
14 Stage Pipeline (11th stage) Wider pipeline, 4 inst/cycle			

Quiz 6

- Fill up the table assuming the same statistics:

	Not Taken	Better (99% accurate)	SpeedUp
5 Stage Pipeline (3rd Stage)	1+0.12x2 =1.24	1+0.01x2 =1.02	1.22
14 Stage Pipeline (11th stage)	1+0.12x10 =2.2	1+0.01x10 =1.1	2
14 Stage Pipeline (11th stage) Wider pipeline, 4 inst/cycle	0.25+0.12x10 =1.45	0.25+0.01x10= 0.35	4.14

So, if we have deeper pipelines or we can execute multiple instructions in a cycle, there is more incentive for better branch predictions.

- Pentium “4” Prescott (extremely deep-pipeline):
  - Fetch…29 stages, resolve branches (thus 30 stages to resolve the branch)
  - Branch Predictions
  - Supports Multiple Instructions/Cycle
- Program consists of:
  - 20% of instructions are branches.
  - 1% of branches are mispredicted.
  - CPI=0.5
- Question: If 2% of branches are mispredicted, what would be the impact on CPI?

- Pentium “4” Prescott (extremely deep-pipeline):
  - Fetch…29 stages, resolve branches (thus 30 stages to resolve the branch)
  - Branch Predictions
  - Supports Multiple Instructions/Cycle
- Program consists of:
  - 20% of instructions are branches.
  - 1% of branches are mispredicted.
  - CPI=0.5
- Question: If 2% of branches are mispredicted, what would be the impact on CPI?

CPI=0.5=X (ideal CPI)+0.2x0.01x30=>X=0.44
If 2% of branches are mispredicted,
CPI=0.44+0.2x0.02x30=0.56

Why do we need better branch predictions?

- CPI gets better
- The other fact is it reduces the number of wasted instructions, wastage of power consumption:

5 stage (3rd stage BR)

14 stage (11th stage BR)

x	x	x	x	x	x	x	x	x	BR

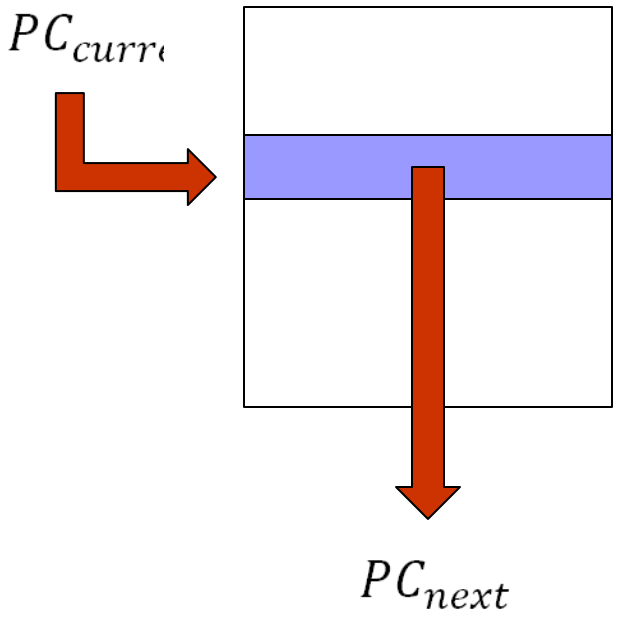
14 stage (11th stage BR, 4 inst/cycle)

x	x	x	x	x	x	x	x	x	BR
x	x	x	x	x	x	x	x	x	
x	x	x	x	x	x	x	x	x	
x	x	x	x	x	x	x	x	x	

Better Prediction

- We have seen the ”Predict not taken” predictor updates the next PC as a function of the current PC by just incrementing the PC: $PC_{next} = f(PC_{current})$
- Following information can help in the prediction:
  - Is the instruction a branch?
  - Is it taken?
  - What is the offset?
- However, we need to perform the prediction at the ‘fetch’ stage
  - At this stage, we have only the current PC value.
- However, History of $PC_{current}$ can help!
  - We know how a ‘branch’ statement had behaved in the past.
  - So, we assume implicitly, that the branches tend to behave the same over and over again.

Branch Target Buffer



Next time, for the same branch the correct PC_{next} is obtained from the BTB, assuming similarity of the branches!

- Predicting whether a branch will be taken is one part of the task.
  - We also need to predict the branch target address.
  - Has to wait till the completion of address computation.
  - We maintain a cache like structure, that records the addresses of the branches and the target addresses associated.
- In a pipelined processor, we carry the $PC_{current}$ and the predicted PC through the pipeline.
- Later in the pipeline, we have the actual PC_{next}.

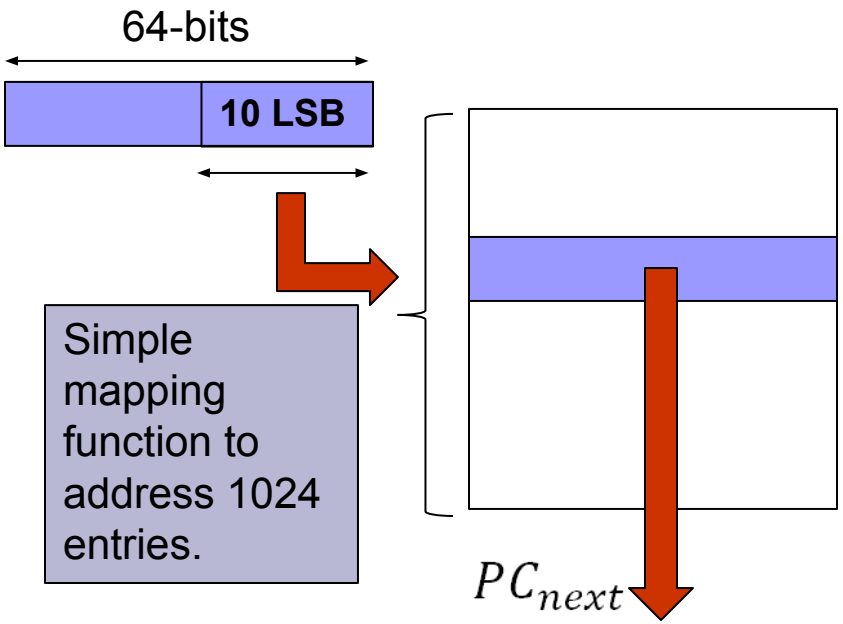
Anatomy of the BTB: Big or Small?

- We want to obtain PC_{next} given $PC_{current}$ in one cycle.
  - For a single cycle latency, the BTB should be small.
- On the other hand, we need an entry for each PC value:
  - If the program to be executed is large, then the BTB should also be big.
  - This implies, we cannot have one entry for every PC value.
- Then, how do we develop a realistic BTB?

Realistic BTB

- We need to have an entry for each PC value.
- However, the BTB has to be small and addressable in 1 cycle.
- So, the idea is - it is enough if we have an entry for all instructions that are going to execute soon!
- For example, if there is a program with say 100 instructions, we really need around 100 entries in the BTB.
- So, we design a realistic BTB with the knowledge on how many entries in the BTB can be accessed in 1 cycle latency.
  - Say 1024 entries.

Realistic BTB



- Why do we take LSBs?
- Consider a program:

0x243C: ADD

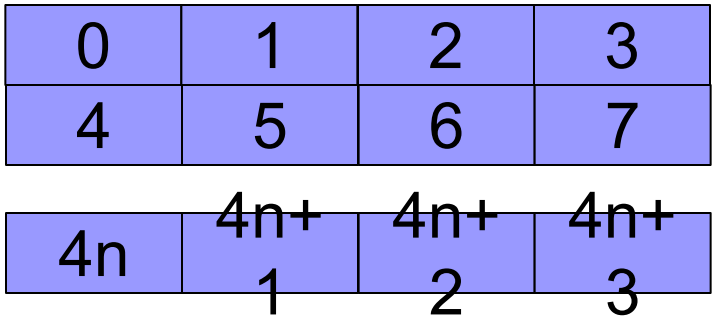
0X2430: MUL

...

If we take the MSBs to index, all instructions in the same part of the program, would get mapped to the same entry. Thus, they will evict each other.
Using LSBs are therefore better.

Quiz 7

- The BTB has 1024 entries (0…1023)
- Fixed-size 4 Byte word aligned instructions
  - Instructions need to begin at addresses divisible by 4. Like 0, 4, …4n,…
- Program Counter has 32-bit addresses.



Which BTB entry is used for PC=0x0000AB0C?

Quiz 7

- The BTB has 1024 entries (0…1023)
- Fixed-size 4 Byte word aligned instructions
  - Instructions need to begin at addresses divisible by 4. Like 0, 4, …4n,…
- Program Counter has 32-bit addresses.

0	1	2	3
4	5	6	7
4n	4n+1	4n+2	4n+3

Which BTB entry is used for PC=0x0000AB0C?

0	Valid
1	Invalid
2	Invalid
3	Invalid
4	Valid

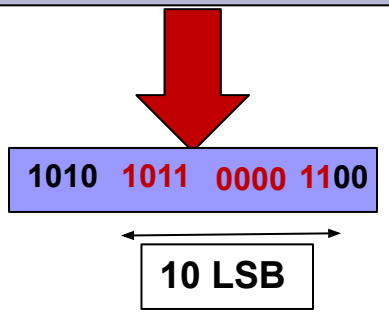
Using 10 LSBs should be good? No! Not from bit 0. Huge wastage, as around ¼ of the BTB entries are invalid, as they are corresponding to not word-aligned addresses.

Quiz 7

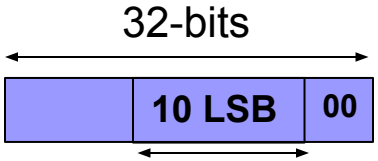
- The BTB has 1024 entries (0…1023)
- Fixed-size 4 Byte word aligned instructions
  - Instructions need to begin at addresses divisible by 4. Like 0, 4, …4n,…
- Program Counter has 32-bit addresses.

0	1	2	3
4	5	6	7
4n	4n+1	4n+2	4n+3

Which BTB entry is used for PC=0x0000AB0C?



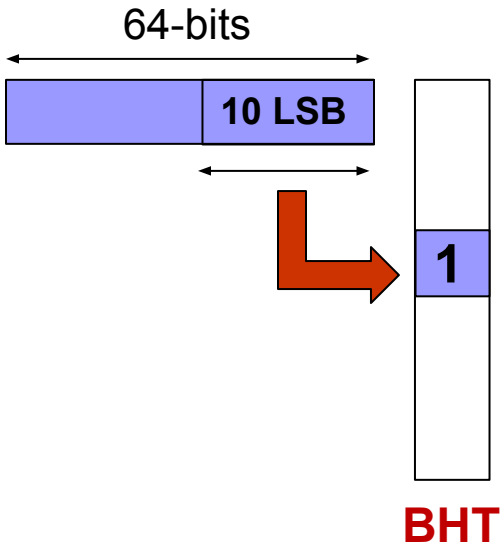
Hence, we will not use 10 LSBs, but use 10 bits leaving the last two bits, ie,



Hence, index in BTB is (10 1100 0011)=2C3

Direction Predictors: Branch History Table

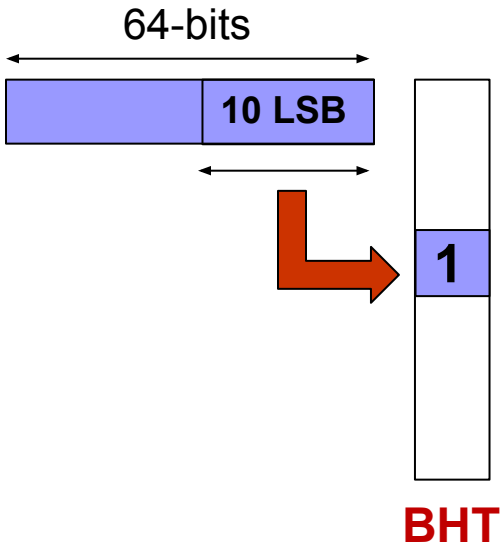
- Rather than using the BTB for all instructions, we use it only for “taken branches”
- We use an auxiliary table of local history registers, called Branch History Table (BHT)



$$\begin{cases} b = 0, & \text{Not a taken branch/not taken branch} \\ b = 1, & \text{A taken branch. Use BTB.} \end{cases}$$

Direction Predictors: Branch History Table

- Just like the BTB, we can update BHT once we resolve the branch.
- If we resolve to a taken, we put 1 and write destination address to BTB.
- If we resolve to 0, we write to BHT and do not update the BTB.
- Because, a BHT entry is 1-bit, BHT can be large.
  - Thus, we can have lots of instructions whose history is maintained without conflicts and use the BTB only for taken branches.



$$\begin{cases} b = 0, & \text{Not a branch/not taken branch} \\ b = 1, & \text{A taken branch. Use BTB.} \end{cases}$$

Quiz 8

			#BHT Accesses	BHT Index
0xC000	MOV	R2, 100		
0xC004	MOV	R1, 0		
0xC008	LOOP: BEQ	R1,R2,DONE		
0xC00C	ADD	R4,R3,R1		
0xC010	LW	R4,0(R4)		
0xC014	ADD	R5,R5,R4		
0xC018	ADD	R1,R1,1		
0xC01C	B	LOOP		
0xC020	DONE:…			

Assume, the BHT has 16 entries, and uses PERFECT prediction, while the BTB has 4 entries, and also uses PERFECT prediction.
How many times to we access the BHT for each instruction?
Hint: Because of perfect prediction, there is no misprediction and no wrong fetches happen!

Quiz 8

			#BHT Accesses	BHT Index
0xC000	MOV	R2, 100	1	0
0xC004	MOV	R1, 0	1	1
0xC008	LOOP: BEQ	R1,R2,DONE	101	2
0xC00C	ADD	R4,R3,R1	100	3
0xC010	LW	R4,0(R4)	100	4
0xC014	ADD	R5,R5,R4	100	5
0xC018	ADD	R1,R1,1	100	6
0xC01C	B	LOOP	100	7
0xC020	DONE:…		1	8

Assume, the BHT has 16 entries, and uses PERFECT prediction, while the BTB has 4 entries, and also uses PERFECT prediction.
How many times to we access the BHT for each instruction?
Hint: Because of perfect prediction, there is no misprediction and no wrong fetches happen!

Quiz 9

			#BTB Accesses	BTB Index
0xC000	MOV	R2, 100		
0xC004	MOV	R1, 0		
0xC008	LOOP: BEQ	R1,R2,DONE		
0xC00C	ADD	R4,R3,R1		
0xC010	LW	R4,0(R4)		
0xC014	ADD	R5,R5,R4		
0xC018	ADD	R1,R1,1		
0xC01C	B	LOOP		
0xC020	DONE:…			

How many times do we access the BTB for each instruction?

Quiz 9

			#BTB Accesses	BTB Index
0xC000	MOV	R2, 100	0	
0xC004	MOV	R1, 0	0	
0xC008	LOOP: BEQ	R1,R2,DONE	1	2
0xC00C	ADD	R4,R3,R1	0	
0xC010	LW	R4,0(R4)	0	
0xC014	ADD	R5,R5,R4	0	
0xC018	ADD	R1,R1,1	0	
0xC01C	B	LOOP	100	3
0xC020	DONE:…			

How many times do we access the BTB for each instruction?

Quiz 10

			#Mispredictions
0xC000	MOV	R2, 100	
0xC004	MOV	R1, 0	
0xC008	LOOP: BEQ	R1,R2,DONE	
0xC00C	ADD	R4,R3,R1	
0xC010	LW	R4,0(R4)	
0xC014	ADD	R5,R5,R4	
0xC018	ADD	R1,R1,1	
0xC01C	B	LOOP	
0xC020	DONE:…		

Assume, the BHT has 16 entries, and uses 1-bit prediction, while the BTB has 4 entries, and also uses PERFECT prediction. The 1-bit predictor is initialized to Not Taken (NT)

How many mispredictions do we have for each instruction?

Hint: A 1-bit prediction means the prediction is the outcome of the previous acccess.

Quiz 10

			#Mispredictions
0xC000	MOV	R2, 100	0
0xC004	MOV	R1, 0	0
0xC008	LOOP: BEQ	R1,R2,DONE	1
0xC00C	ADD	R4,R3,R1	0
0xC010	LW	R4,0(R4)	0
0xC014	ADD	R5,R5,R4	0
0xC018	ADD	R1,R1,1	0
0xC01C	B	LOOP	1
0xC020	DONE:…		

Assume, the BHT has 16 entries, and uses 1-bit prediction, while the BTB has 4 entries, and also uses PERFECT prediction. The 1-bit predictor is initialized to Not Taken (NT)

How many mispredictions do we have for each instruction?

Hint: A 1-bit prediction means the prediction is the outcome of the previous acccess.

Problems with 1 Bit Prediction

- Predicts well:
  - Always Taken (May make a mistake first time, but after that predicts correctly)
  - Always Not Taken
  - Taken>>> Not Taken
  - Not Taken>>>Taken
- Problem occurs when we have:

  T T T…. T **NT** T … T
- Each such anomaly will cause two mispredictions.
- So, the 1-bit prediction is not good for T, NT sequences where the number of NT is say almost of similar number compared to T branches.

Quiz 11

- Consider a 1 bit predictor initialized to T.
- Consider, a branch instruction with the following sequence:

  a) T T …T (100 times) NT (5 times) TT … T (100 times)

  b) T…T (20 times) NT (5 times) T…T (20 times)

Compute the % mispredictions in both cases.

  - Case a, shows where Taken>>> Not Taken
  - Case b, shows where Taken>Not Taken

Quiz 11

- Consider a 1 bit predictor initialized to T.
- Consider, a branch instruction with the following sequence:

  a) T T …T (100 times) NT (5 times) TT … T (100 times)

  b) T…T (20 times) NT (5 times) T…T (20 times)

Compute the % mispredictions in both cases.

a) T T …T (100 times) (**NT** NT NT NT NT NT)**T**T … T (100 times): % of misprediction = 2/(200+5)=1%

b) T T …T (20 times) (**NT** NT NT NT NT NT)**T**T … T (20 times): % of misprediction = 2/(40+5)=11.11%

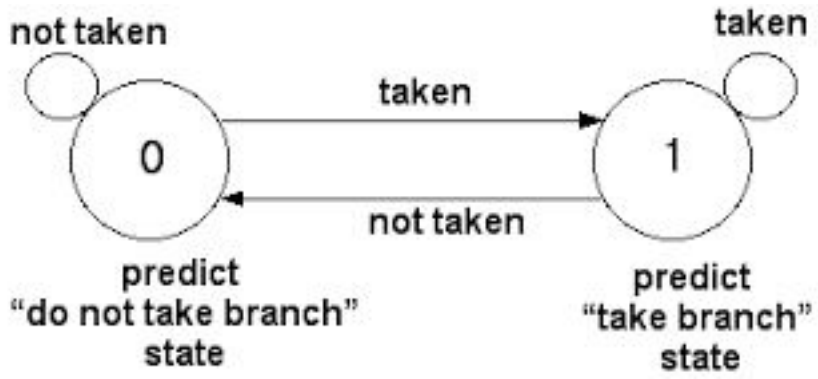
Why we need hysteresis in prediction?

- Main motivation is in the prediction of branches at the end of the loops.

```
LOOPOUT: BEQ R1,R2, DONEOUT
LOOPIN: BEQ R3, R4, LOOPOUT
…
DONEOUT: …
```

- In the case, where the prediction is just the direction that the branch took last, the prediction for the branch terminating the inner loop (ie. LOOPIN) will be a NT, resulting in one misprediction (as it is a taken!)
- Nothing can be much done about this exit misprediction.
- However, in case of nested loops, when the outer loop executes the inner loop afresh, the first prediction for LOOPIN will be a T, resulting in another misprediction.
- We can eliminate this by enforcing that two wrong predictions are needed in a row for a change in prediction.

1-bit Predictor



Predicts well:
- **Always Taken**
- **Always Not Taken**
- **Taken>>>Not Taken**
- **Not Taken>>>Always Taken**

T	T	T	T	T	NT	T	T	T	T
✔	✔	✔	✔	✔	x	x	✔	✔	✔

Does not Predict so well:
- Taken>Not Taken
- Not Taken>Always Taken
- Short Loops (loops with an exit condition, one which stays in the loop, and goes back)
- Work bad for cases where Taken ≈ Not Taken

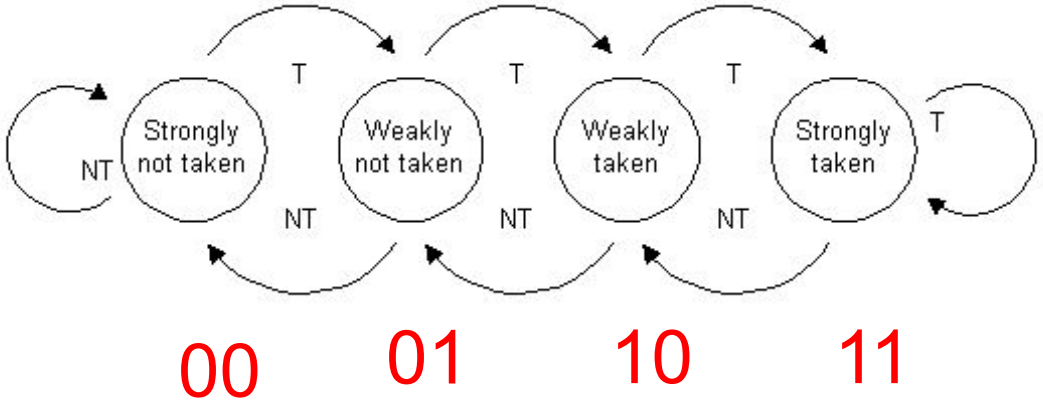
Each anomaly there are two mis-predictions

2-bit Predictor

0	0
0	1
1	0
1	1

Prediction bit

Hysteresis (conviction) bit

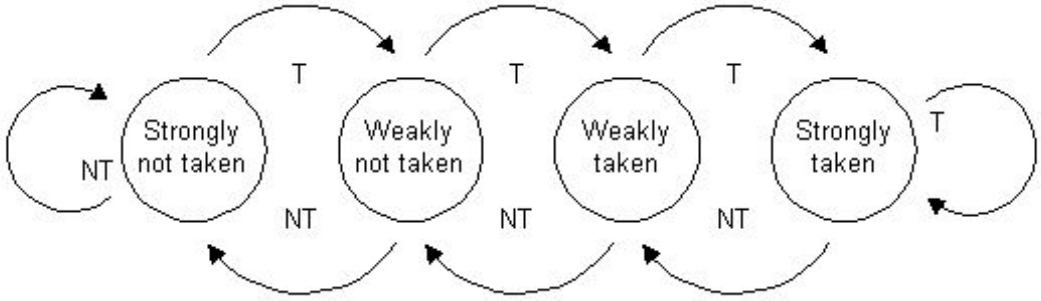


T	T	T	T	T	NT	T	T	T	T
✔	✔	✔	✔	✔	x	✔	✔	✔	✔

These predictors are good when anomaly comes in between!

- A single anomaly will cause 1 misprediction.
- A switch in behavior will cause 2 mispredictions.

2-Bit Saturating Predictor Initialization



- Start in a strong state:

00	00	00
NT	NT	NT

00	01	10	11
T	T	T	T

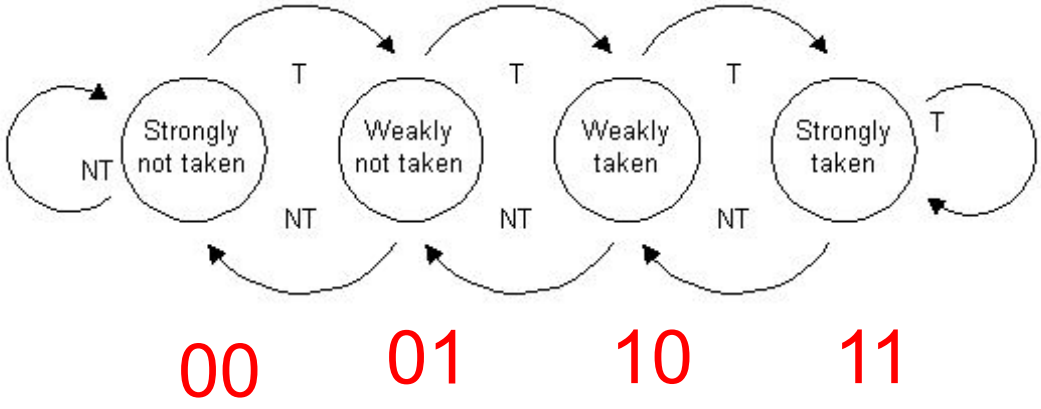
- Start in a weak state:

01	00	00
NT	NT	NT

01	10	10	11
T	T	T	T

If you are wrong about predicting the dominant behavior, we have one less misprediction if we start in the weak state.

2-Bit Saturating Predictor Initialization



- Consider the sequence when initialized to NT:

00	01	00	01
T	NT	T	NT

- Start in a weak state:

01	10	01	10
T	NT	T	NT

This behavior is bad for initialization at the weak state.

A pathological sequence with strong initialization

00	01	10	01	10	01	10
T	T	NT	T	NT	T	NT

- So, in reality there is no as such to choose between the two initializations
- Every predictor has a worst case scenario where 100% mispredictions occur.
- Having, a 3-bit predictor can be good when there are 2 NT in between.
- Likewise, we can move to 4-bit predictors, but the cost increases exponentially…Further, the chance of such sequences in real programs is limited.
- So, in real life we have either 2-bit or 3-bit predictors.

How do we improve? :History vs Majority

NT T NT T NT T NT T …
NT NT T NT NT T NT NT…

100 % predictable
But not with n-bit predictors.

With 2-bit predictors you will have significant number of mis-predictions!

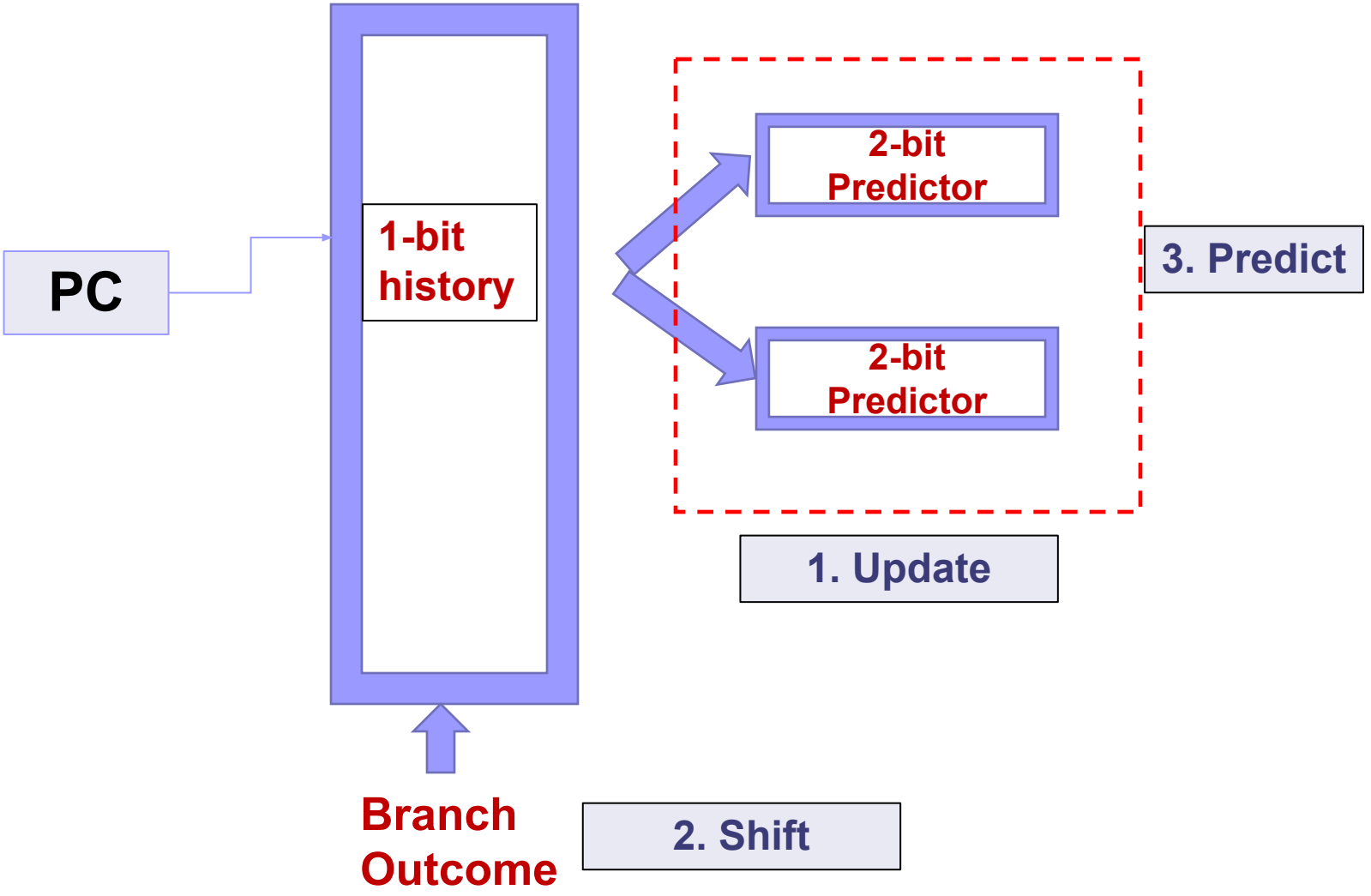
History:	NT	T
Prediction:	T	NT

History:	NT,NT	NT,T	T,NT
Prediction:	T	NT	NT

From history we can learn the pattern! Then this sequence is 100% predictable!

From history we can learn the pattern! In this case, we need 2-bits of history.

1-bit history with 2-bit counters



Update-Shift-Predict (USP)

State	Pred	Outcome	Correct
(0,SN,SN)		T	
		N	
		T	
		N	
		T	
		N	
		T	

Update-Shift-Predict (USP)

State	Pred	Outcome	Correct
(0,SN,SN)	N	T	x
(1,WN,SN)	N	N	✔
(0,WN,SN)	N	T	x
(1,WT,SN)	N	N	✔
(0,WT,SN)	T	T	✔
(1,ST,SN)	N	N	✔
(0,ST,SN)	T	T	✔

If the previous outcome is 0, it predicts T, else N. So, we have perfect prediction.

Quiz 12: Predict the sequence (NNT)*

State	Pred	Outcome	Correct
(0,SN,SN)		N	
		N	
		T	
		N	
		N	
		T	
		N	
		N	
		T	

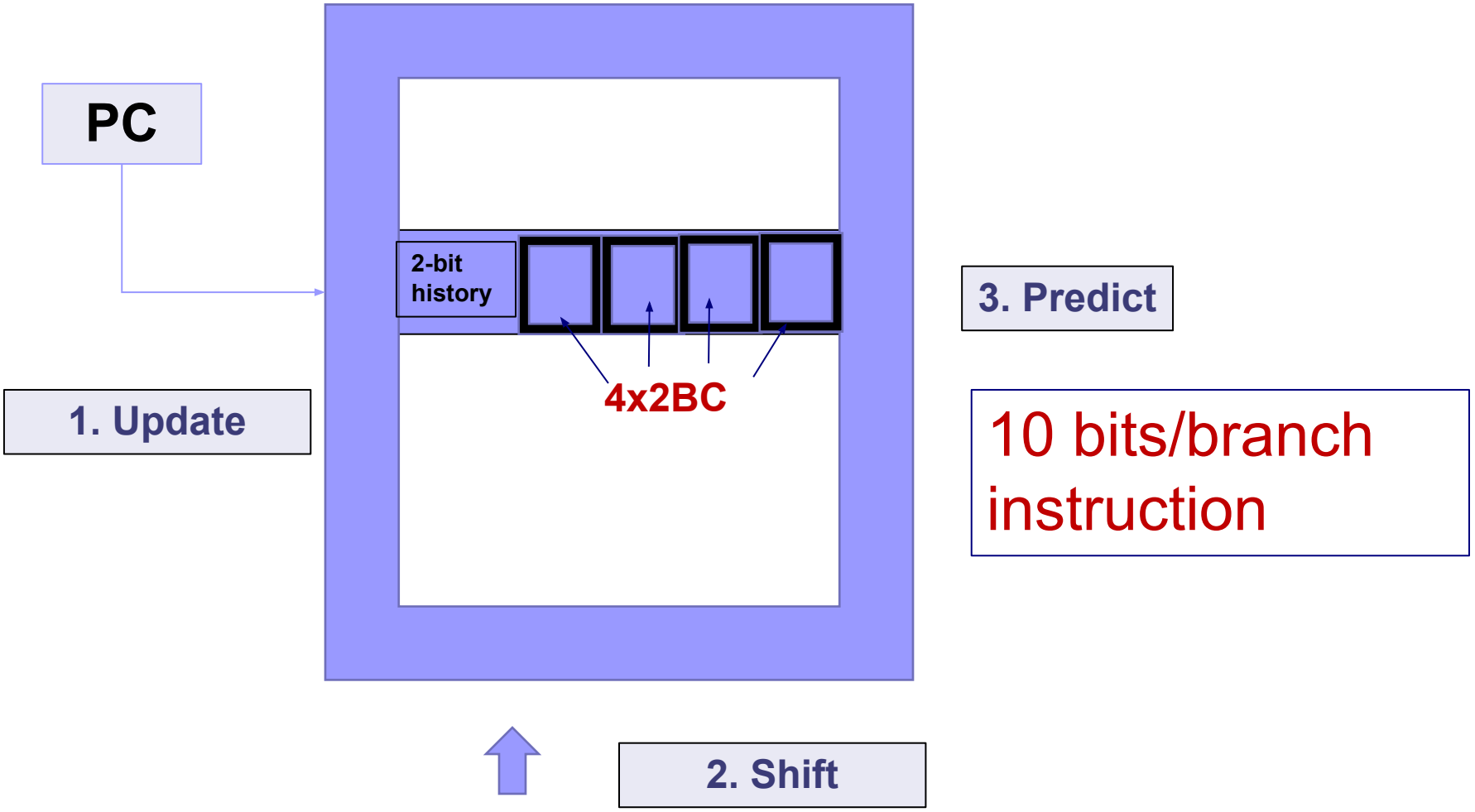
After 100 repetitions, no of mispreditions is ?

Quiz 12: Predict the sequence (NNT)*

State	Pred	Outcome	Correct
(0,SN,SN)	N	N	
(0,SN,SN)	N	N	
(0,SN,SN)	N	T	
(1,WN,SN)	N	N	
(0,WN,SN)	N	N	
(0,SN,SN)	N	T	
(1,WN,SN)	N	N	
(0,WN,SN)	N	N	
(0,SN,SN)	N	T	

After 100 repetitions, no of mispreditions is ?
In every round, there is one misprediction. So, in total there are 100 mispredictions.

2-bit History Predictor



10 bits/branch instruction

Prediction of (NNT)*

	N	N	T	N	N	T
History bits=>	….	’00’	’01’	’10’	’00’	
Counter Directions:		C0⇧	C1⇩	C2⇩	C0⇧	
Final States:			ST	SN	SN	ST

- 2-bit history predictor can accurately learn (NNT)* accurately.
- After an initial warm up period, will learn the pattern with 100% accuracy.
- But we waste 1 2-BC.

Prediction of (NT)*

	N	T	N	T	N
History bits=>	….		’01’	’10’	’01’
Counter Directions:			C1⇩	C2⇧	C1⇩
Final States:			SN		ST

- Can predict the pattern (NT)* with 100% accuracy, after the initial warm up period.
- But it is wasting 2 x 2BC.

N-bit History Predictor

- In general N-bit history predictor can learn all patterns of length$\leq N + 1$
  - Single bit history can learn patterns of length 2 and also 3.
  - 2 bit history can learn pattern of length 2 and also 3.
  - Both can learn patterns of length 1, ie. always N or T.
- Cost per entry: $N + 2^N \times 2$
  - For 10 bit history size required is more than 1kilobit of information for a single branch.
  - Most 2BCs are wasted.
  - With 10 bit history 1024 counters can be indexed, but in the USP technique only 11 histories are of relevance.
  - Lots of histories are not used.
- With increasing N, we can predict longer patterns, but cost and wastage both increase.

Quiz 13

- N-bit history with 2 BCs
- Need 1024 entries for each branch to have an entry.

	Cost (bits)	(NNNT)*	#of 2 BCs used for (NT)*
N=1			
N=4			
N=8			
N=16			

Quiz 13

- N-bit history with 2 BCs
- Need 1024 entries for each branch to have an entry.

	Cost (bits)	(NNNT)*	#of 2 BCs used for (NT)*
N=1	5x1024	No	2
N=4		Yes	2(out of 16)
N=8		Yes	2(out of 256)
N=16		Yes	2(out of around 65000)

Quiz 14

```
for(i=0;i!=8;i++)
 for(j=0;j!=8;j++)
     {
        …(no branches)
      }
```

- At least _____ entries.
- Each entry should have at least _____ -bit history
- Number of counters = _____

Quiz 14

```
for(i=0;i!=8;i++)
 for(j=0;j!=8;j++)
     {
        …(no branches)
      }
```

- At least ___4____ entries.
- Each entry should have at least ___8___ -bit counters
- Number of counters = ___256_____

```
i=0
Loop1: if(i==8) B Done
          j=0
Loop2: if(j==8) B D1
       //inner loop operations
   D1: B Loop2
   B Loop1
Done:…
```

Pattern: (N N…NT)* Is of length 9, so we need 8-bits of history.

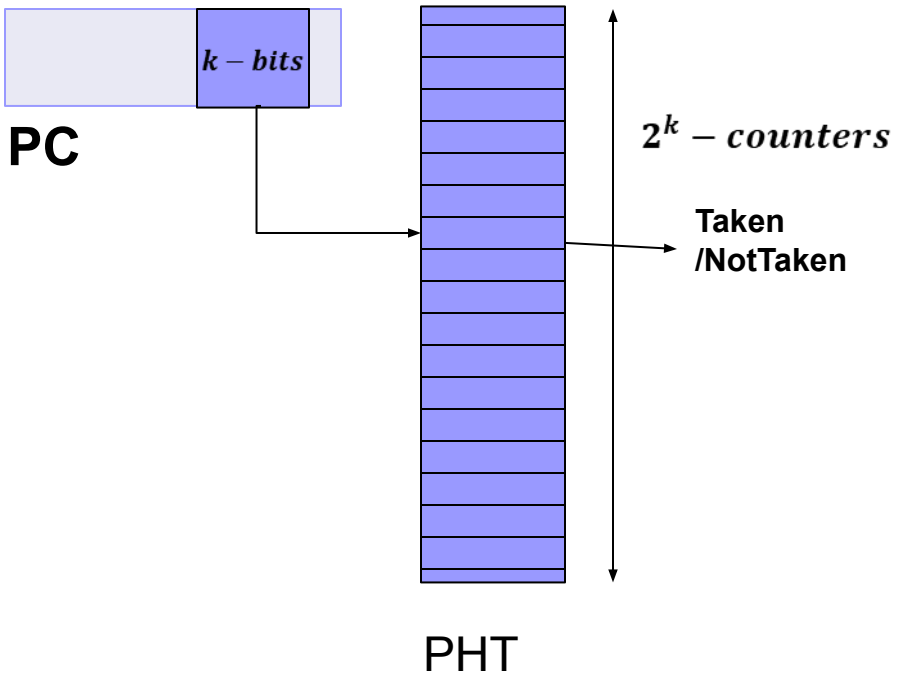
Loop back branches, are always taken

Only 9, 2BCs will be used.

Sharing Counters

- N-bit history can index 2^N counters per branch.
  - However only $O(N)$ counters are used.
- Can we share the counters, so that some branches with short history can use smaller number of counters, while some with longer history can use more number of counters?
  - Idea: We shall have a pool of 2 BCs and share them between the branch entries.
  - Cons: There is a possibility that different entries corresponding to different PC values and different history end up in indexing the same 2BC.

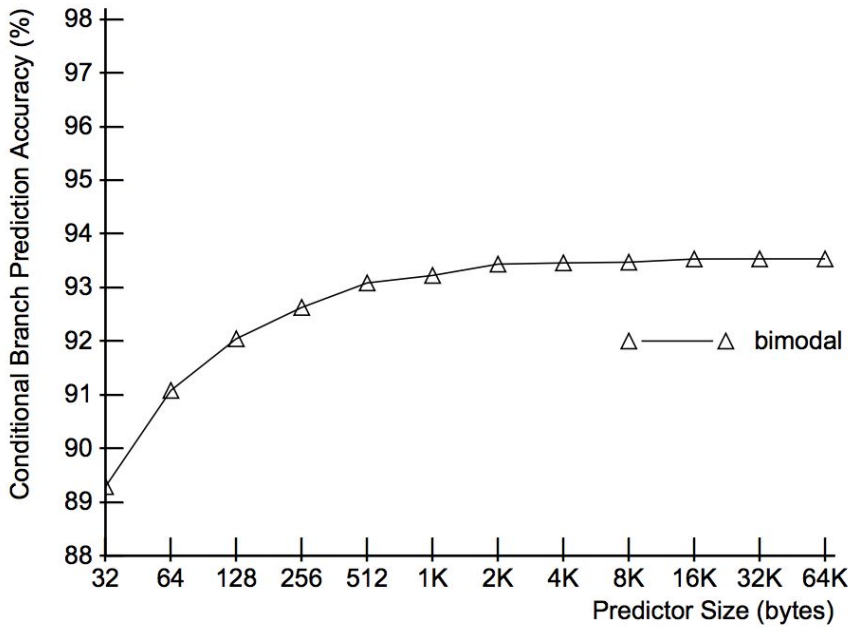
BiModal Predictor



- Each entry in the PHT are 2-bits for a 2-BC.
- The PHT is indexed by select PC bits.
- Aliasing happens when two different branches are assigned the same 2-BC.
  - This can be compensated by increasing the PHT, which can be achieved as every entry has only 2 bits.

Scott McFarling, “Combining Branch Predictors”, 1993

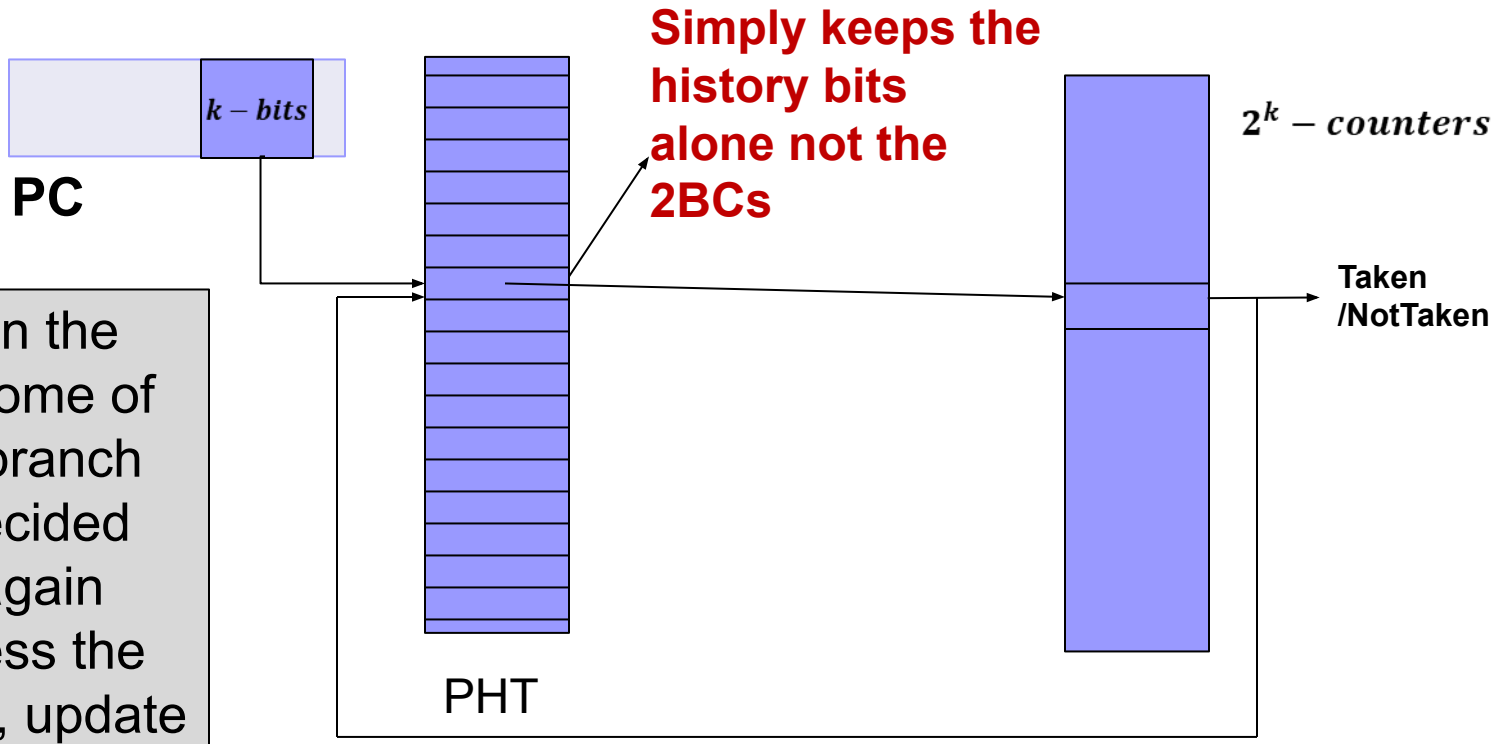
Accuracy of bimodal predictor as function of PHT Size



Average accuracy for conditional branches in first 10 million instructions of SPEC89 traces reached 93.5% for PHT of size 2KB (ie, 8K counters)

But we need to incorporate history into the prediction!

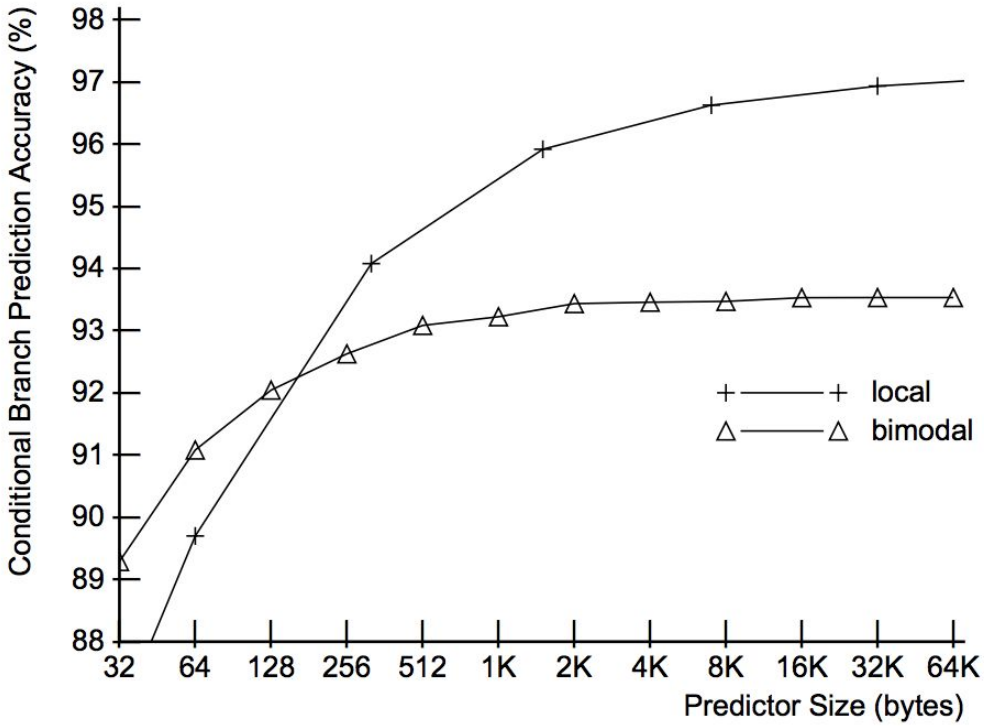
Local History Predictor Structure



When the outcome of the branch is decided we again access the 2BC, update its state and shift in the branch outcome into the PHT.

- For every PC, we have a small number of possible histories.
  - Hence, number of 2BCs used are small for a branch.
  - Thus, if we have a large number of 2BCs, possibility of collisions is low.

Performance

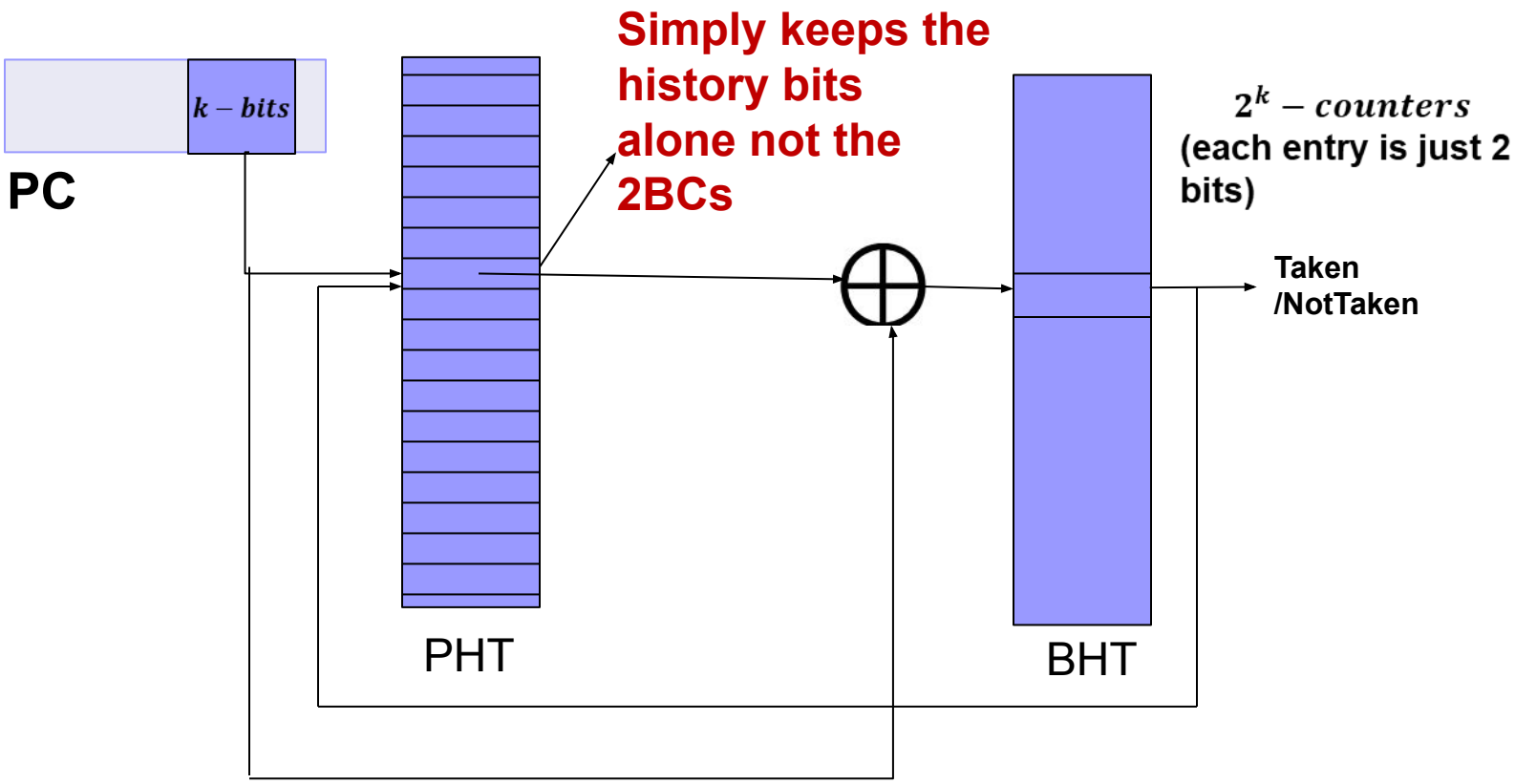


For very small predictors there is excessive contention for history entries, then storing this history is of no value.

However, above about 128 bytes, the local predictor has significantly better performance.

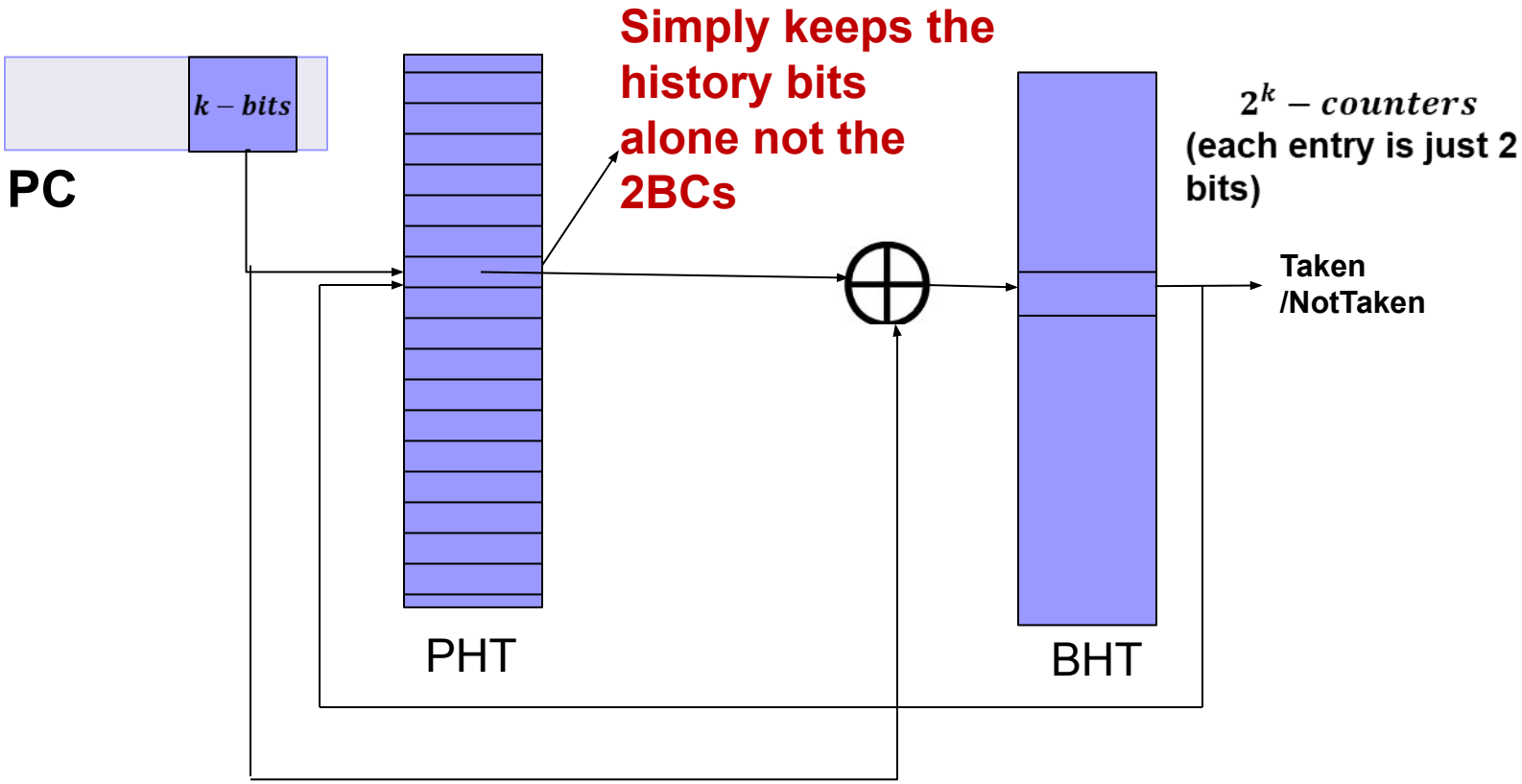
For large predictors, the accuracy approaches 97.1%!

The Pshare Predictor



- The indexing of the 2BCs is not only a function of the history but also of the PC.
- For a collision: $g(PC_1, H_1) = g(PC_2, H_2)$. Chances are lesser assuming that the. BHT can be large.

The Pshare Predictor: Can Increase history bits



Consider, 11 bits of PC with 11 bits of history (need not be same).
Size of PHT Table=$2^{11} \times 11$
Size of BHT Table=$2^{11} \times 2$
Total Size=$2^{11} \times 13 = 26kBits.$
Compare with the previous design!

Number of Counters depending on History

- Consider a branch that is always Taken:
  - TTTT : In that case, the PC value of the branch gets combined with the same history (it is a constant), and results in accessing the same 2BC. Thus, only 1 counter is used.
  - NNNN: Like before only 1 counter is used.
  - NTNT: History is either 0101 or 1010, ie. 2 counters are used.
    - Most branches would be like these with short patterns. Thus very few counters are needed leaving room for branches with longer history.
  - Say, a long pattern with 16 history bits, we need around 17 counters.
  - Thus, keeping the BHT larger (the PHT is more consuming), as per entry is only 2 bits, we can reduce the chance of the overlap.

Nested For loops and Corresponding Assembly showing the Branches

```
main()
{
int i, j;
int sum=0;

for(i=0;i<10;i++)
   for(j=0;j<8;j++)
     {
      sum=sum+i+j;
     }
}
```

```
        movl    $0, -4(%rbp)
        movl    $0, -16(%rbp)
        movl    $0, -8(%rbp)
LBB0_1:                                 ## =>This Loop Header: Depth=1
                                        ##     Child Loop BB0_3 Depth 2
        cmpl    $10, -8(%rbp)
        jge     LBB0_8
## %bb.2:                               ##   in Loop: Header=BB0_1 Depth=1
        movl    $0, -12(%rbp)
LBB0_3:                                 ##   Parent Loop BB0_1 Depth=1
                                        ## =>  This Inner Loop Header: Depth=2
        cmpl    $8, -12(%rbp)
        jge     LBB0_6
## %bb.4:                               ##   in Loop: Header=BB0_3 Depth=2
        movl    -16(%rbp), %eax
        addl    -8(%rbp), %eax
        addl    -12(%rbp), %eax
        movl    %eax, -16(%rbp)
## %bb.5:                               ##   in Loop: Header=BB0_3 Depth=2
        movl    -12(%rbp), %eax
        addl    $1, %eax
        movl    %eax, -12(%rbp)
        jmp     LBB0_3
LBB0_6:                                 ##   in Loop: Header=BB0_1 Depth=1
        jmp     LBB0_7
LBB0_7:                                 ##   in Loop: Header=BB0_1 Depth=1
        movl    -8(%rbp), %eax
        addl    $1, %eax
        movl    %eax, -8(%rbp)
        jmp     LBB0_1
LBB0_8:
```

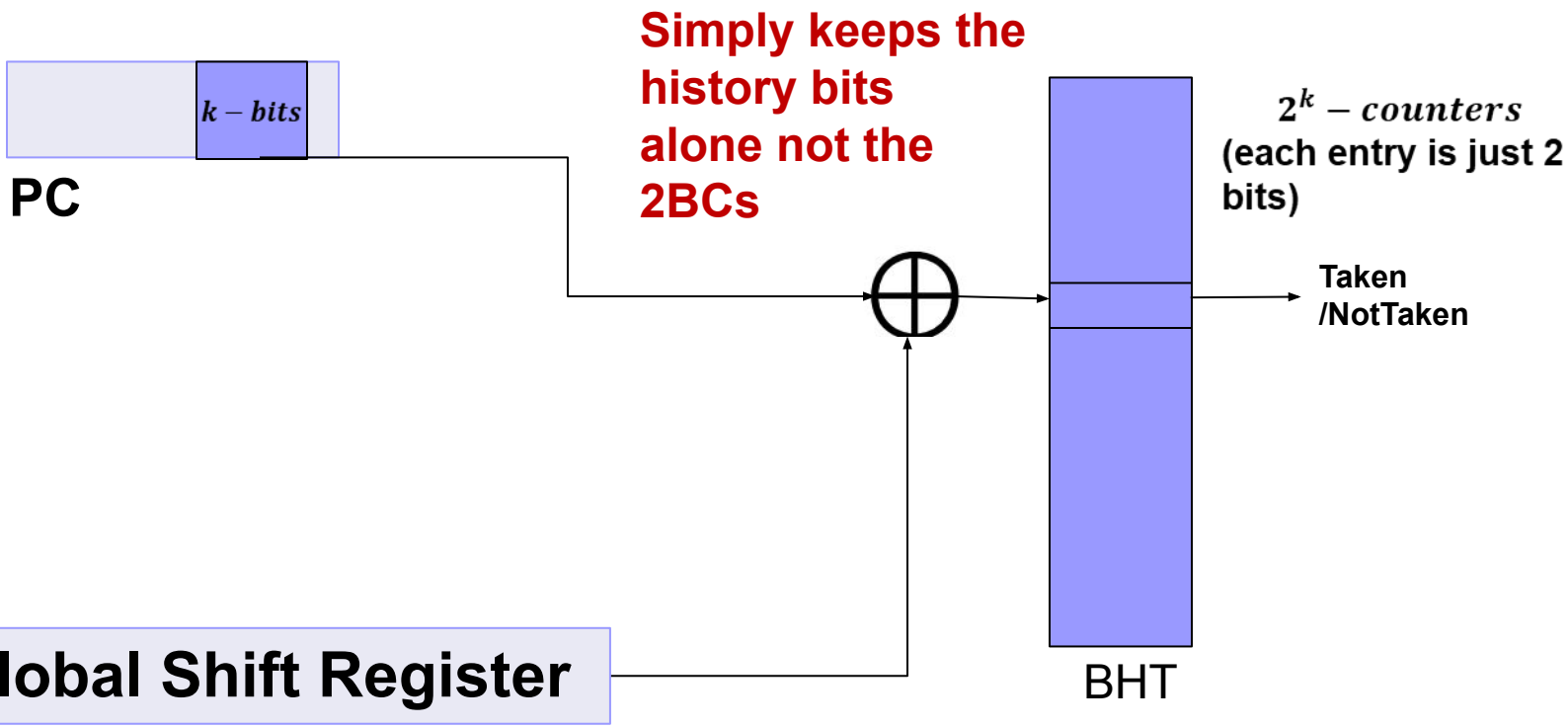
Global Predictors

- There exists a weakness in the prediction accuracy while using local histories.
  - The decision is based on usually some number of history bits of the branch whose outcome is predicted.
  - Consider, the following snippet with *correlated branches*:

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

If both branches b_1 and b_2 are taken, branch b_3 will not be taken.
Local predictors will fail.
So, we use a global history register.

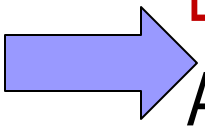
GShare



Single register that stores the history for all the branches.
Good for correlated branches.

Quiz 15

```
for(int i=1000;i!=0;i--)
    if(i%2) n+=i;
```



```
LOOP:
BEQ R1, zero, Exit
AND R2, R1, 1
BEQ R2, zero, Even
ADD R3, R3, R1
Even: ADD R1, R1, -1
B LOOP
```

B1 each time not taken, except Once
A single 2BC will predict accurately.

Good accuracy on all branches.

History should be:

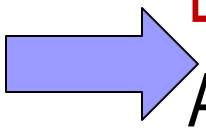
- For Pshare?
- For Gshare?

Easily predictable.

For the 2nd branch: BEQ R2, zero, Even,
the pattern is (NT T NT T…). Thus, for PShare we need history of length 1.

Quiz 15

```
for(int i=1000;i!=0;i--)
    if(i%2) n+=I;
```



```
LOOP:
BEQ R1, zero, Exit
AND R2, R1, 1
BEQ R2, zero, Even
ADD R3, R3, R1
Even: ADD R1, R1, -1
B LOOP
```

B1 each time not taken, except Once
A single 2BC will predict accurately.

Easily predictable.

Good accuracy on all branches.

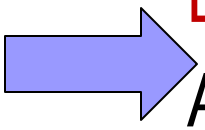
History should be:

- For Pshare?
- For Gshare?

For Pshare history should be 1 bit.

Quiz 15

```
for(int i=1000;i!=0;i--)
    if(i%2) n+=I;
```



```
LOOP:
BEQ R1, zero, Exit
AND R2, R1, 1
BEQ R2, zero, Even
ADD R3, R3, R1
Even: ADD R1, R1, -1
B LOOP
```

B1 each time not taken, except Once
A single 2BC will predict accurately.

Good accuracy on all branches.

History should be:

- For Pshare?
- For Gshare?

Easily predictable.

Global History: 011|001|011|001…
For GShare, only 2 history sequences possible:
011|001|011|001
011|001|011|001 …Thus, 3 bits of history are needed.

Pshare vs GShare

- GShare can predict correlated branches, which PShare cannot.
- But GShare requires bigger history.
- So, a tournament predictor tries to use both.
- Predictors are sensitive to data.

```
Consider:
for(i=0;i<N;i++){
   if(A[i]<50)
     do_something();
   if(A[i]>50)
     do_something_else();
}
```

Assume data is randomly distributed in [1…100].
Then PShare may fail.
But GShare will easily learn that second branch depends on first.
If first branch is taken second is not.
If first is not taken then it is very likely second branch will be taken.

Note when A[i]=50, there is a 1% chance for the second branch to be mispredicted.

Pshare vs GShare

- GShare can predict correlated branches, which PShare cannot.
- But GShare requires bigger history.
- So, a tournament predictor tries to use both.
- Predictors are sensitive to data.

Consider:

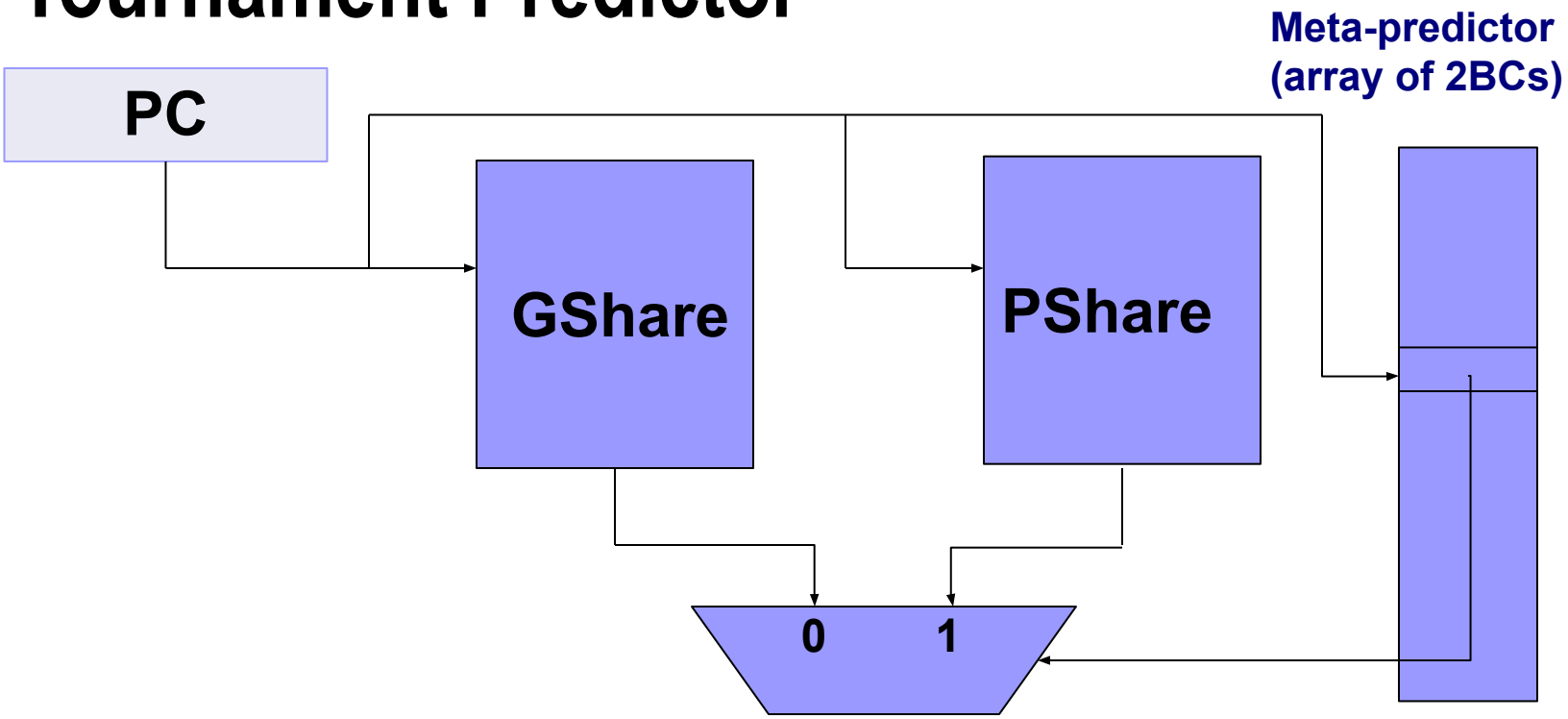
```
for(i=0;i<N;i++){
   if(A[i]<50)
     do_something();
   if(A[i]>50)
     do_something_else();
}
```

Now assume data is sorted.

Sequence: NN…NTT…T

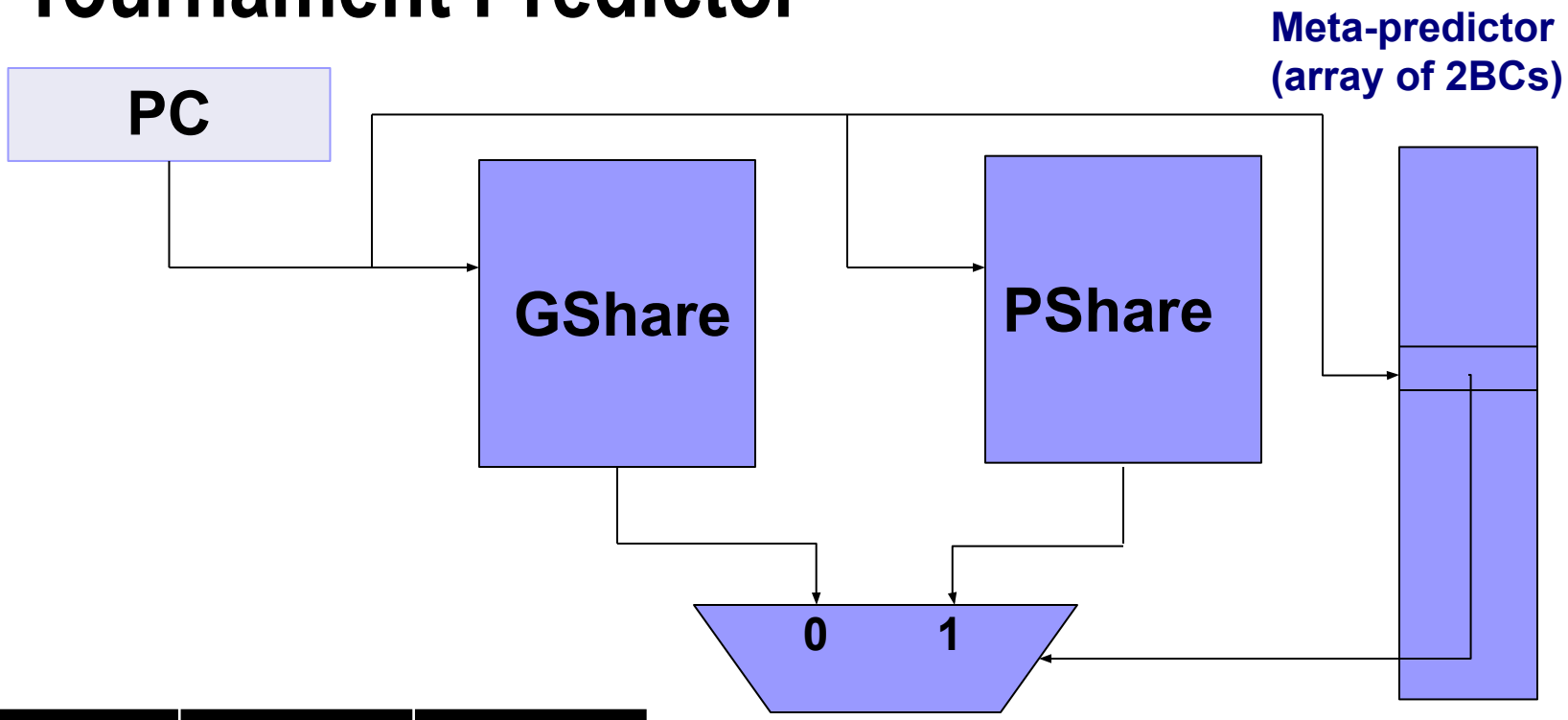
In that case PShare (or a simple bimodal predictor) will win and perform better than GShare.

Tournament Predictor



- Meta-predictor does not predict the branch, but tells which one is more accurate between the GShare and PShare.
- Note, the 2BCs in the meta-predictor do not count up and down as usual on a T and NT branch.
- Rather they are trained on the success and failure of the GShare and PShare predictors.

Tournament Predictor



GShare	PShare	

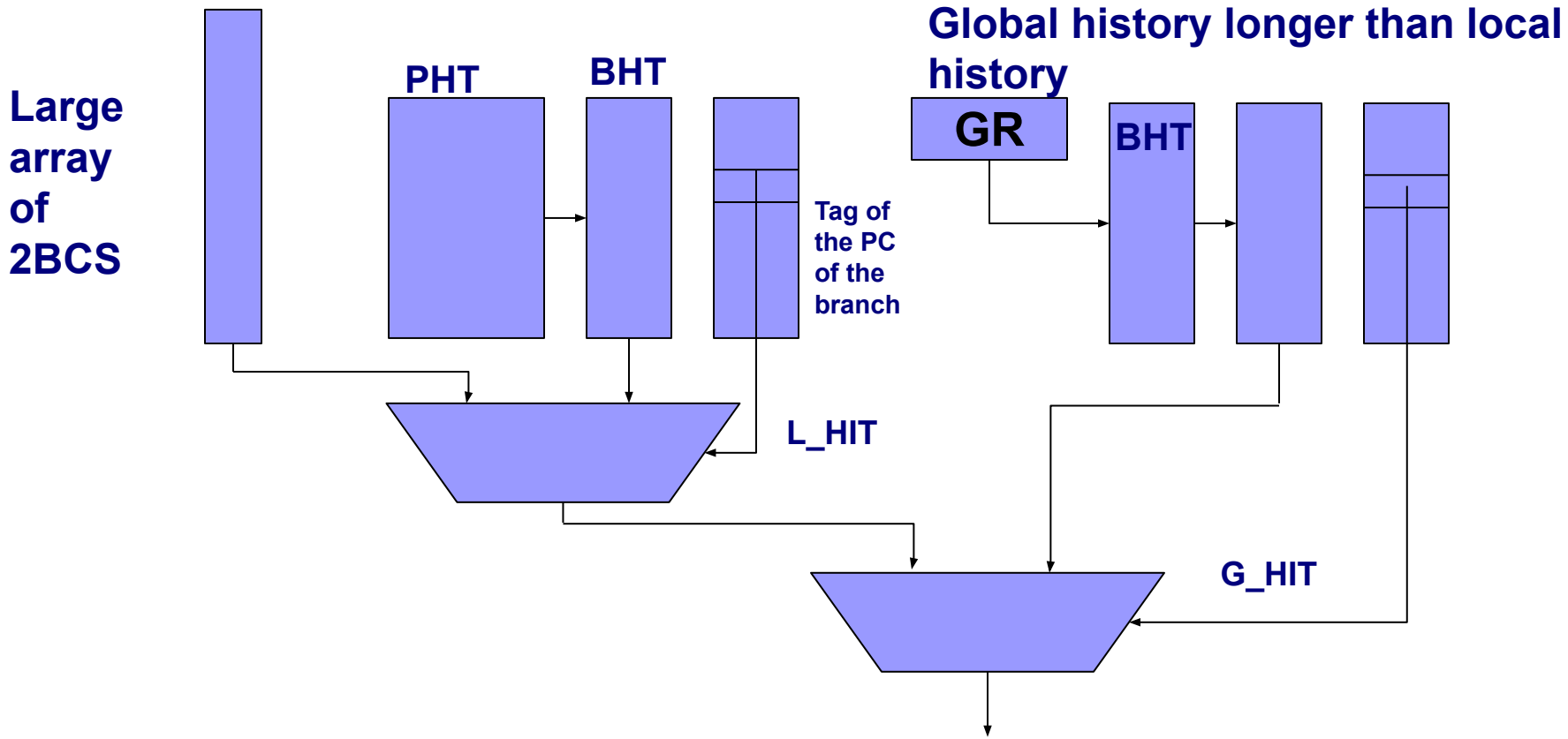
The hysteresis in the 2BC helps to ensure that one off prediction from the two predictors does not change the selection.

Hierarchical Predictors

- Tournament: Selection between 2 Good Predictors.
  - It is expensive.
  - But at a time we are using only one of them.
- Hierarchical: 1 Good, 1 OK predictor
  - We use the OK predictor in cases when the branch is easy to predict.
- In tournament predictors, we need to update both on the outcome of a branch which is again costly.
- In hierarchical predictors, we update the OK predictor on each decision, we update the Good predictor only if the OK prediction is not good.

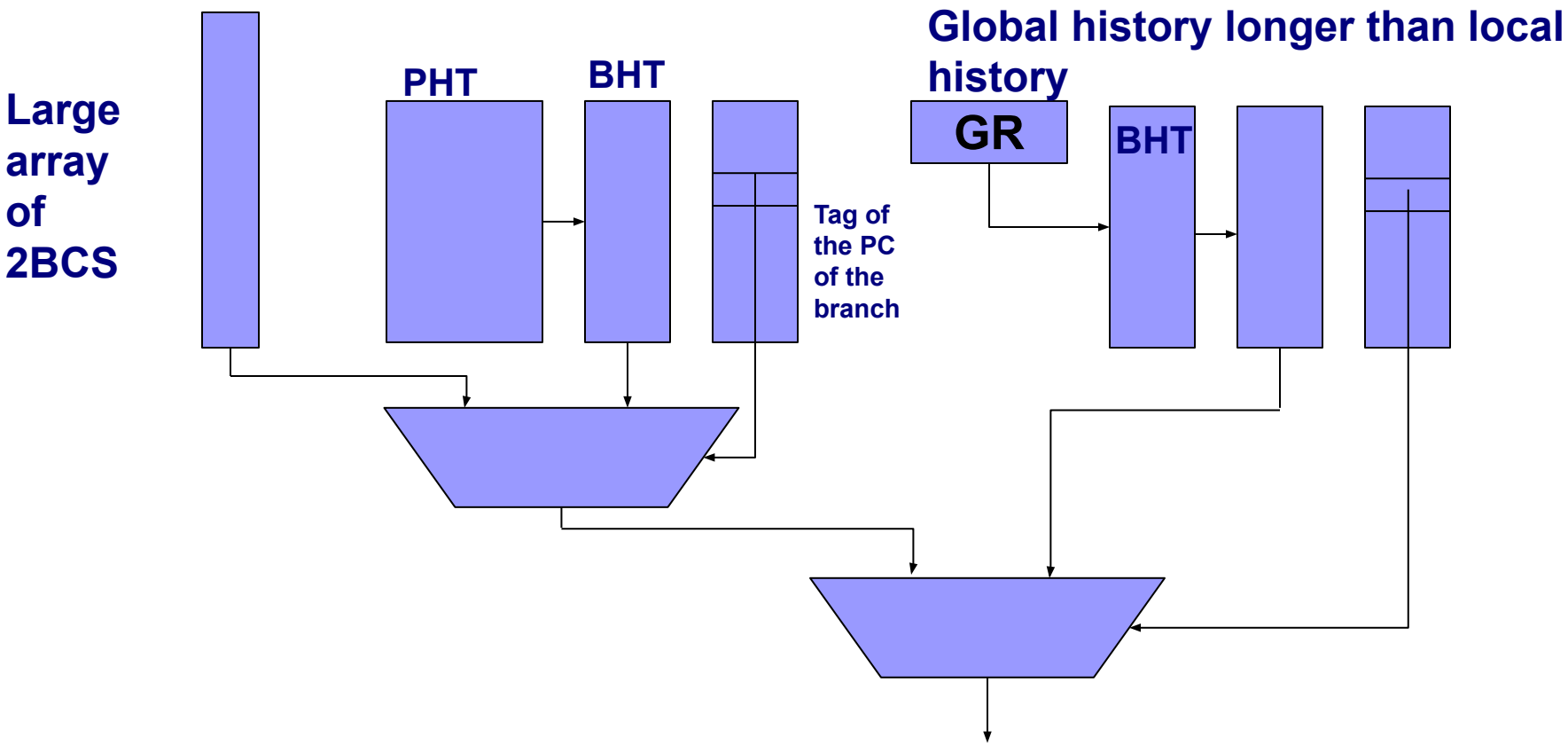
Hierarchical Predictor Example

- Pentium M uses:
  - A Cheap predictor with 2BCs
  - Predictor with local history
  - Predictor with global history



Hierarchical Predictor Example

- We update the 2BCs on a branch outcome.
- We update the local/global history predictor if the branch is there inside them.
- Else, we update them only if the previous predictor is not working well!
  - So, this way the 2BCs take care of a majority of branches, like always T, or NT, dominant branches.
  - We can keep few histories of those branches needed in the local or global predictors.

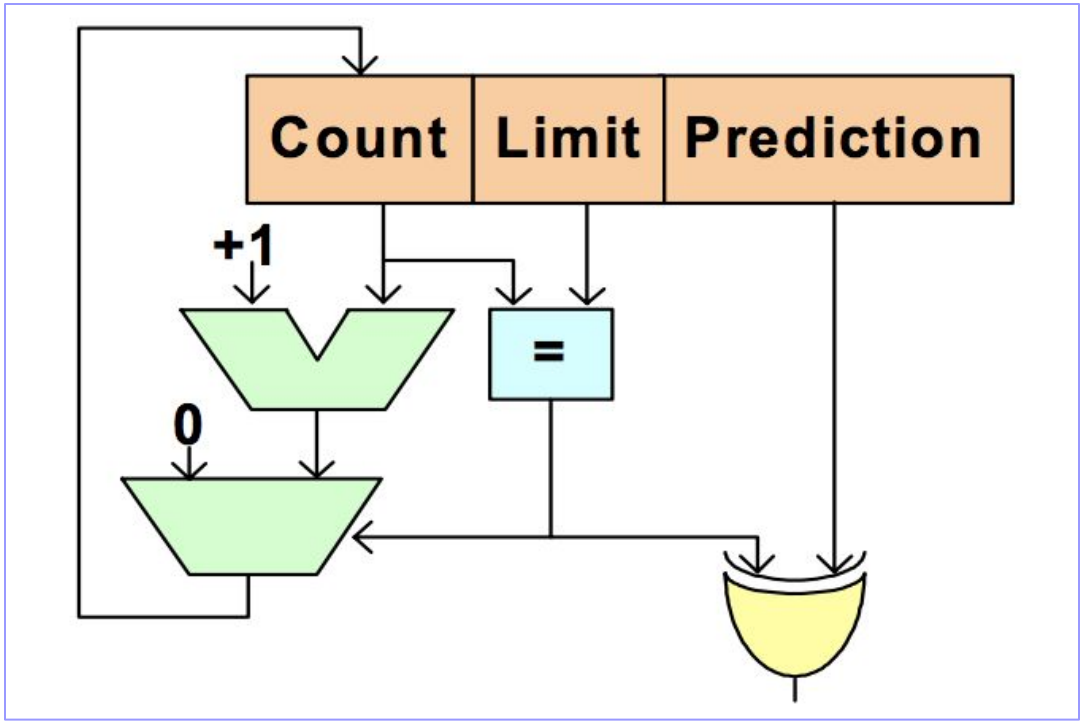


Loop Detector

- When a given branch shows a repeated pattern of many NT followed by a single T (or opposite) a special loop detector is used.
- Introduced in Pentium M.
- This is needed as often the number of loops can be much larger than the length of a history register.

Gochman et al.,“The Intel Pentium M Processor: Microarchitecture and Performance,”Intel Technology Journal, May 2003

Loop Detector Architecture



Quiz 16

- 2 BP works fine for 95% of instructions.
- PShare works fine for 95% plus 2% more.
- GShare works fine for 95% plus other 5%.
- The overall predictor is _____ predictor that chooses between A and _____ predictor and _____ predictor, which itself chooses between _____ and _____.

Handling Return Instructions

- We have seen several types of branches to predict.

  BNE R0, R1, label

  JUMP CALL

- Direction and Target Address can be predicted by the techniques seen.
- But Function Return is tricky.
  - While the direction can be predicted by known techniques.
  - But the target will be mis-predicted.

Why the Target Address will be Missed?

0x1230: CALL FUN

0x1250: CALL FUN

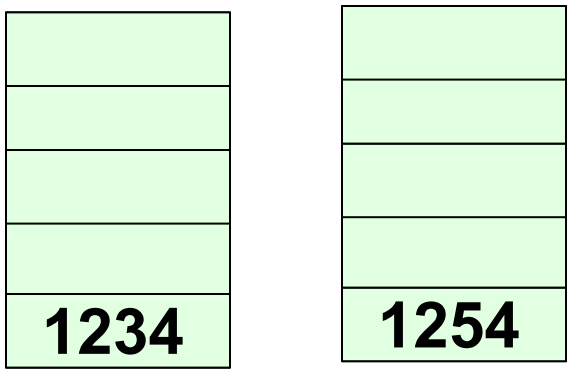
FUN: ..

RET

The BTB will remember 1234 as the target address for the first RET instruction. For the 2nd RET instruction, it is a miss!

A Branch Target Buffer (BTB) will never predict accurately.

Return Address Stack (RAS)




```
0x1230: CALL FUN

0x1250: CALL FUN

FUN: ..
     RET
```

- We maintain a small hardware stack, say of size 16.
  - It is fast!
- During function can push the return address.
- When we see the return address we pop the stack.
- When the stack is full we push in the recent address and an old address is removed from the stack.

Repair Mechanisms

- For GShare there are two options for updates:
  - Non-speculatively, whereby a bit in the global register will be inserted at (branch) instruction execute/commit.
  - Speculatively, whereby a bit is inserted when the branch is predicted at decode time.
- First, strategy is simple to implement.
- However, it has drawbacks.

Consider, if(aa==2) //b_1 branch

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

Consider, a branch b that is predicted at time t and committed at time $t + \Delta t$. This could be branch b1. Any branch that needs to be predicted during the interval Δt, say b2 or b3, will not benefit, even if it is predictable.

Repair Mechanisms

- For GShare there are two options for updates:
  - Non-speculatively, whereby a bit in the global register will be inserted at (branch) instruction execute/commit.
  - Speculatively, whereby a bit is inserted when the branch is predicted at decode time.
- First, strategy is simple to implement.
- However, it has drawbacks.

Consider, if(aa==2) //b_1 branch

aa=0;

if(bb==2) //b_2 branch

bb=0;

if(aa!=bb){//b_3 branch

…

}

This leads to possible miss of correlated branches.

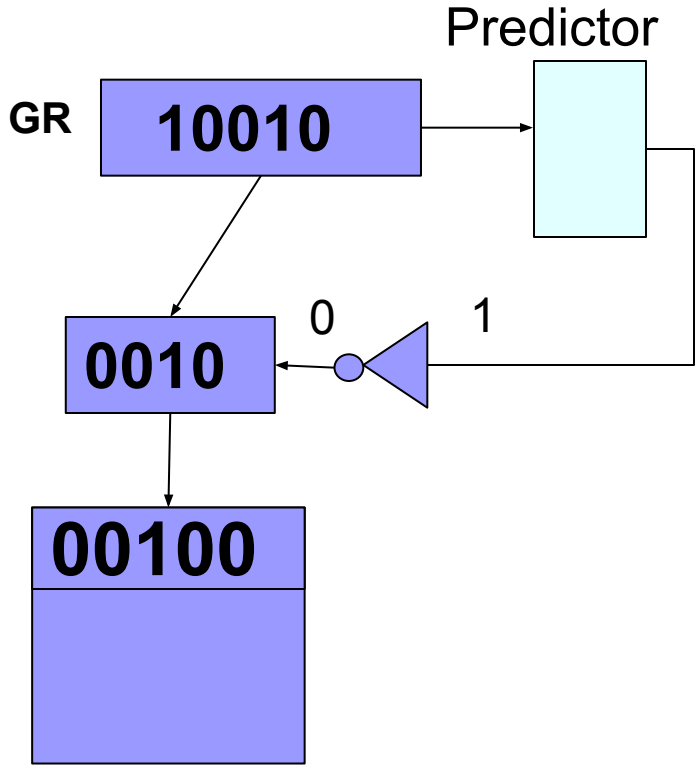
Moreover, the global registers for two consecutive predictions for any branch, say b, might include different ancestors of branch b, even if the same path leading to b was repeated (so same history).

This is because of mispredictions or cache misses that alter the timing of branches before b.

So, as the timing to commit would differ for the prior branches, there effects are not yet reflected in the history of the current branch b.

Update using Speculation

- Speculatively update the global register when the prediction is performed.
- If there is a misprediction, now we need a repair mechanism.



- Check-point the global register during prediction.
  - $GR_{old} \leftarrow GR$
- As instruction fetching is done in program order, the check-pointed GR can be put in a FIFO queue.
- The flip of the predicted bit is shifted in GR_{old} and queued.
- If the repair is done at commit stage, the head of the queue appended wit the flipped bit becomes the new global register.
- All other entries are discarded.

Bimodal and Private History Predictors

- Updating speculatively does not make much sense.
  - It would reinforce the decision.
  - Not affecting other entries in the BTB.
  - Other BTB entries does not affect the branch entry to be updated.
  - So, it is usually done at commit stage.
    - Benchmarks show it has no practical effect on prediction accuracy.

BTB with Direction Predictors

- Both target and direction needs to be simultaneously predicted.
- When the direction is mispredicted, it is very costly.
  - All instructions fetched after the branch must be nullified.
  - Recovery can happen as late as when the mispredicted branch is committed.
- The second is when a branch is correctly predicted to be taken but there is a BTB miss.
  - Penalty is that no new instruction can be fetched until the target address is computed during the decode stage.
  - Affects the filling of the pipeline.
- The third is when the branch is correctly predicted to be taken and there is a BTB hit, but the target address does not match.
  - This is called misfetch.
  - Can occur in case of indirect jumps.
  - From Intel Pentium M, there is a dedicated predictor for indirect branches that use global history registers associated with the target addresses to decrease misfetches.

Computing the branch execution penalty

- Penalty for a mispredict: m_{pred}
- Penalty for a misfetch: m_{fetch}
- % of branches that the misfetched:$Miss_f$
- % of branches that the mispredicted:$Miss_p$
- Contribution to CPI due to the branches/branch execution penalty:

$$bep = Miss_f \times m_{fetch} + Miss_p \times m_{pred}$$

Consider the following code:

```
int c;
int main ()
 {
int i, j;
for (i=0; i<1000; i++)
  { for (j=0; j<4; j++) {
      c++;
    }
 }
return c;
 }
```

```
main:
        leal    4(%esp), %ecx           ; function overhead
        andl    $-16, %esp
        pushl   -4(%ecx)
        pushl   %ecx
        xorl    %ecx, %ecx              ; i = 0 (%ecx)
.L2:
        xorl    %edx, %edx              ; j = 0 (%edx)
.L3:
        movl    c, %eax                 ; %eax = c
        addl    $1, %edx                ; j++
        addl    $1, %eax                ; %eax++
        movl    %eax, c                 ; c = %eax
        cmpl    $4, %edx                ; if j < 4 then goto L3
        jne     .L3                     ; INNER LOOP BRANCH
        addl    $1, %ecx                ; i++
        cmpl    $1000, %ecx             ; if i < 1000 then goto L2
        jne     .L2                     ; OUTER LOOP BRANCH
        movl    c, %eax                 ; return c
        popl    %ecx                    ; function overhead
        leal    -4(%ecx), %esp
        ret
```

Assuming no conflicts between branch addresses, and assuming all entries are initialized to 0, find #mispredictions for:

1. BHT with 1-bit history.
2. BHT with 2-bit counters.

Solution-with 1 bit entries

- There are entries:
  - Inner loop branch.
  - Outer loop branch.
  - Pattern for inner loop: T T T N
  - Prediction: N T T T
  - So, there will be 1000*2=2000 mispredictions due to inner loop.
  - The outer loop also will be mispredicted 2 times (one at the beginning and one at the end).
  - Total, 2002 mispredictions

With 2 bit entries

	T	T	T	N	T	T	T	N	T	T	T	N
SN	WN	WT	ST	WT	ST	ST	ST	WT	ST	ST	ST	WT
	N	N	T	T	T	T	T	T	T	T	T	T

There are 3 mispredictions in the first loop and only 1 misprediction from the next loops.

Thank You!