CS / EE 320
Computer Organization and Assembly Language
Lecture 20
Spring 2024

Shahid Masud

Topics: Branch Prediction Techniques

Topics

- Static and Dynamic Branch Prediction – some review
- 1-Bit and 2-Bit Branch Prediction
- (m,n), Global and Tournament Predictors
- How to construct a Branch History Table
- Operation and Advantage of Branch History Table
- Performance Evaluation of Branch Prediction in Pipelined CPU
- Loop Unrolling
- Speculation, ILP, etc.
- Some example questions

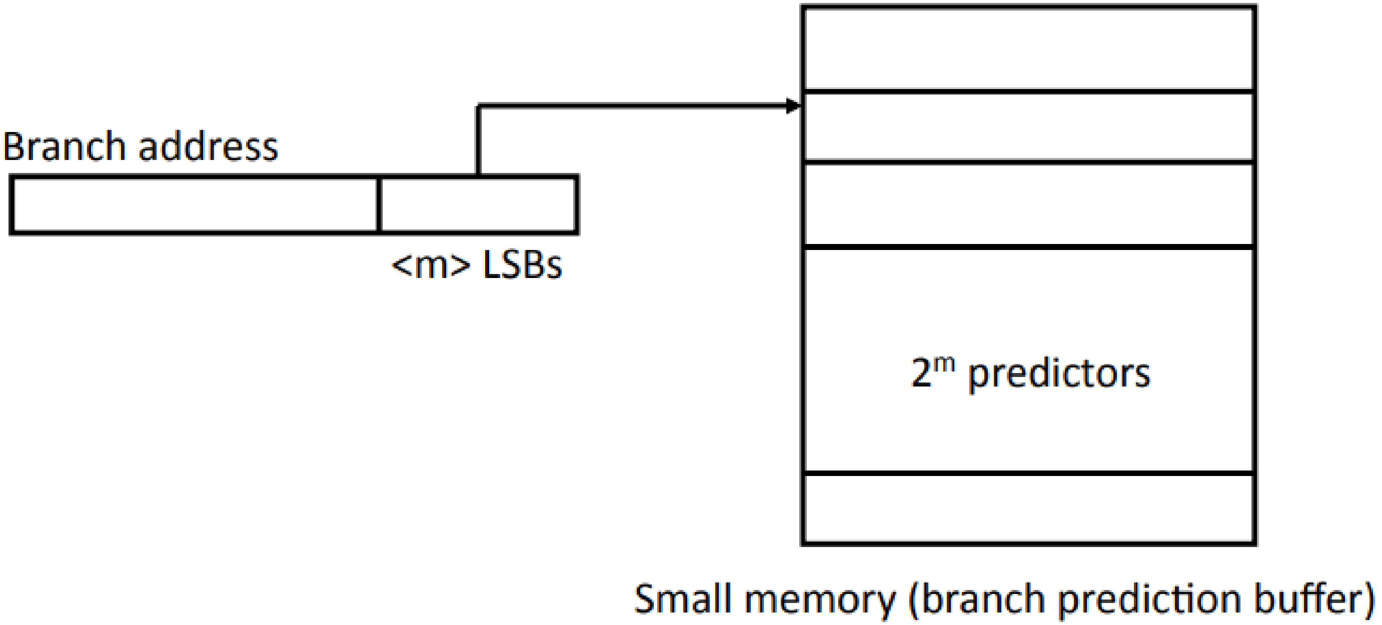
Quiz Next Lecture

Static and Dynamic Branch Prediction

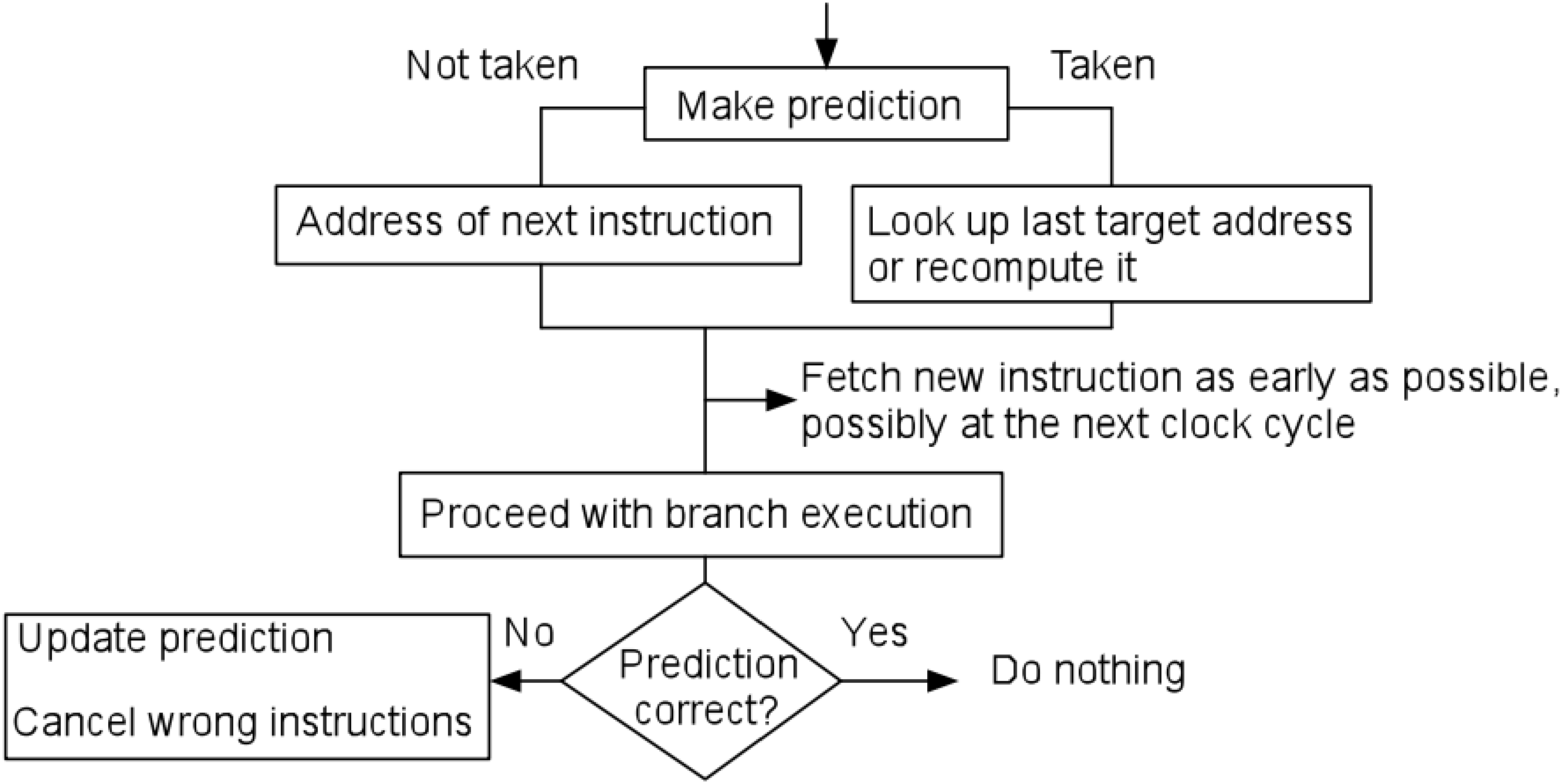
- In deeper and superscalar pipelines, branch penalty is more significant
- Use dynamic prediction
- Branch prediction buffer (aka branch history table)
- Indexed by recent branch instruction addresses
- Stores outcome (taken/not taken)
- To execute a branch
  - Check table, expect the same outcome
  - Start fetching from fall-through or target
  - If wrong, flush pipeline and flip prediction

What is Dynamic Branch Prediction?

- *Idea*: predict the outcome of a branch based on its past behaviour



Flowchart of Dynamic Branch Prediction


Make prediction
Not taken
Taken
Address of next instruction
Look up last target address or recompute it
Fetch new instruction as early as possible, possibly at the next clock cycle
Proceed with branch execution
Prediction correct?
No
Yes
Update prediction
Cancel wrong instructions
Do nothing

Performance = *f*(accuracy, cost of misprediction)

Popular Branch Prediction Schemes

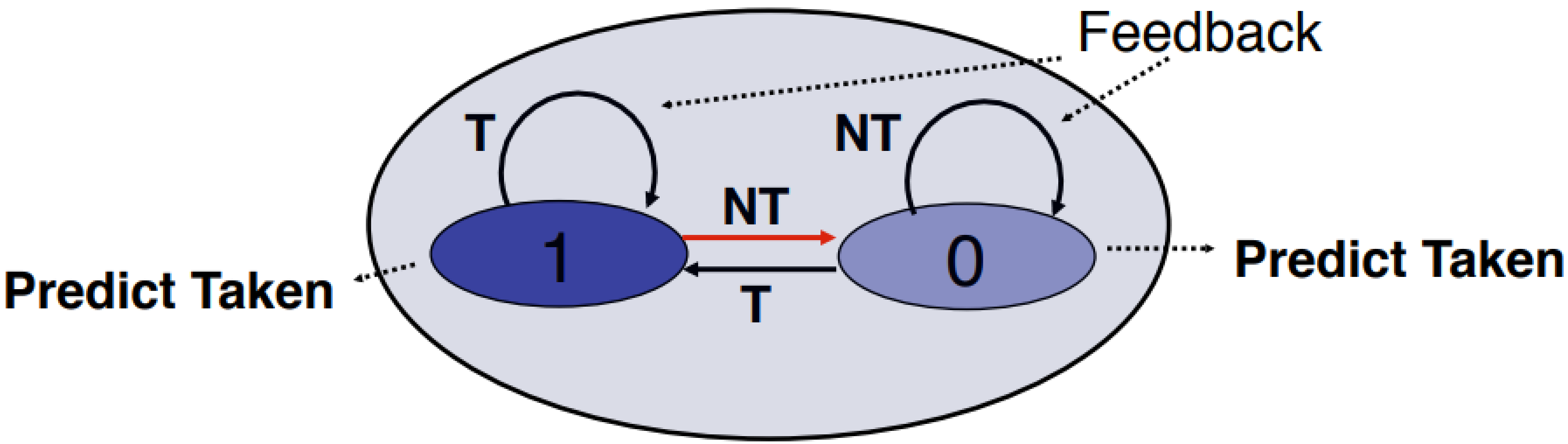
1. 1-bit Branch Predictor
2. 2-bit Branch Predictor
3. Correlating Branch Predictor
4. Tournament Branch Predictor

What is a 1-bit Branch Predictor?

- Branch History Table
  - Indexed by LSBs of PC address
  - Table contains 1-bit values: branch last taken or not?
  - No address check (saves HW; causes aliasing)
  - Works for numerical code with many loops
- In a loop, 1-bit BHT is likely wrong twice
  - Last iteration: predicts the loop continues
  - First iteration: predicts exit instead of looping

1 Bit Branch Predictor

1-bit prediction



Example of 1 bit Predictor

1-bit branch predictor

- *Main idea*: we use the history of a branch's past outcomes to predict its future outcomes.
- *1-bit predictor*: when we execute a branch, first check if it was taken when it was last executed: if yes, predict it will be taken this time; if no, predict it will not be taken this time.

- *Example*: `for (i=0; i<10; i++) { … };`

  This loop is iterated 10 times and involves one branch, e.g.,

  ```
  loop: ...
          SLTI R2,R1,#10 ;is i<10?
          BNEZ R2,loop    ;branch if yes
  ```

  The branch is taken in the first 9 iterations, and not taken on the 10th iteration. What's the result of using 1-bit prediction?

Contd – 1-bit Predictor

- (Assume that when the branch is executed for the first time, we predict that it is not taken.)

Iteration #:	1	2	3	4	5	6	7	8	9	10
Predicted branch outcome:	N	T	T	T	T	T	T	T	T	T
Actual branch outcome:	T	T	T	T	T	T	T	T	T	N

- How is 1-bit branch prediction implemented in hardware?
  - Use a branch history table: there is an entry in the table for every branch encountered in the program. (Actually, table is indexed using lower bits of branch instruction's PC. * -> 1 possible)
  - Each table entry contains *one prediction bit* for that branch, e.g., 0 for predict not taken (N), 1 for predict taken (T).

How the Predictor bit changes?

- The prediction bit is used to predict the branch outcome. It is updated after the branch's *actual* outcome is known.
- The following shows how the prediction bit for the branch in our example changes in each iteration of the loop:

Iter#	Pred. bit	Actual outcome	Update
1	N	T	T
2	T	T	T
3	T	T	T
4	T	T	T
5	T	T	T
6	T	T	T
7	T	T	T
8	T	T	T
9	T	T	T
10	T	N	N

Example of 1-bit Branch Predictor

- Loop with 10 iterations

Iteration #:	1	2	3	4	5	6	7	8	9	10
Actual branch outcome:	T	T	T	T	T	T	T	T	T	N
Predicted branch outcome:	**N**	T	T	T	T	T	T	T	T	**T**

- Mispredict twice for every 10 iterations
  - (Assuming that when the branch first executed we predict that it is not taken)
  - 80% prediction accuracy

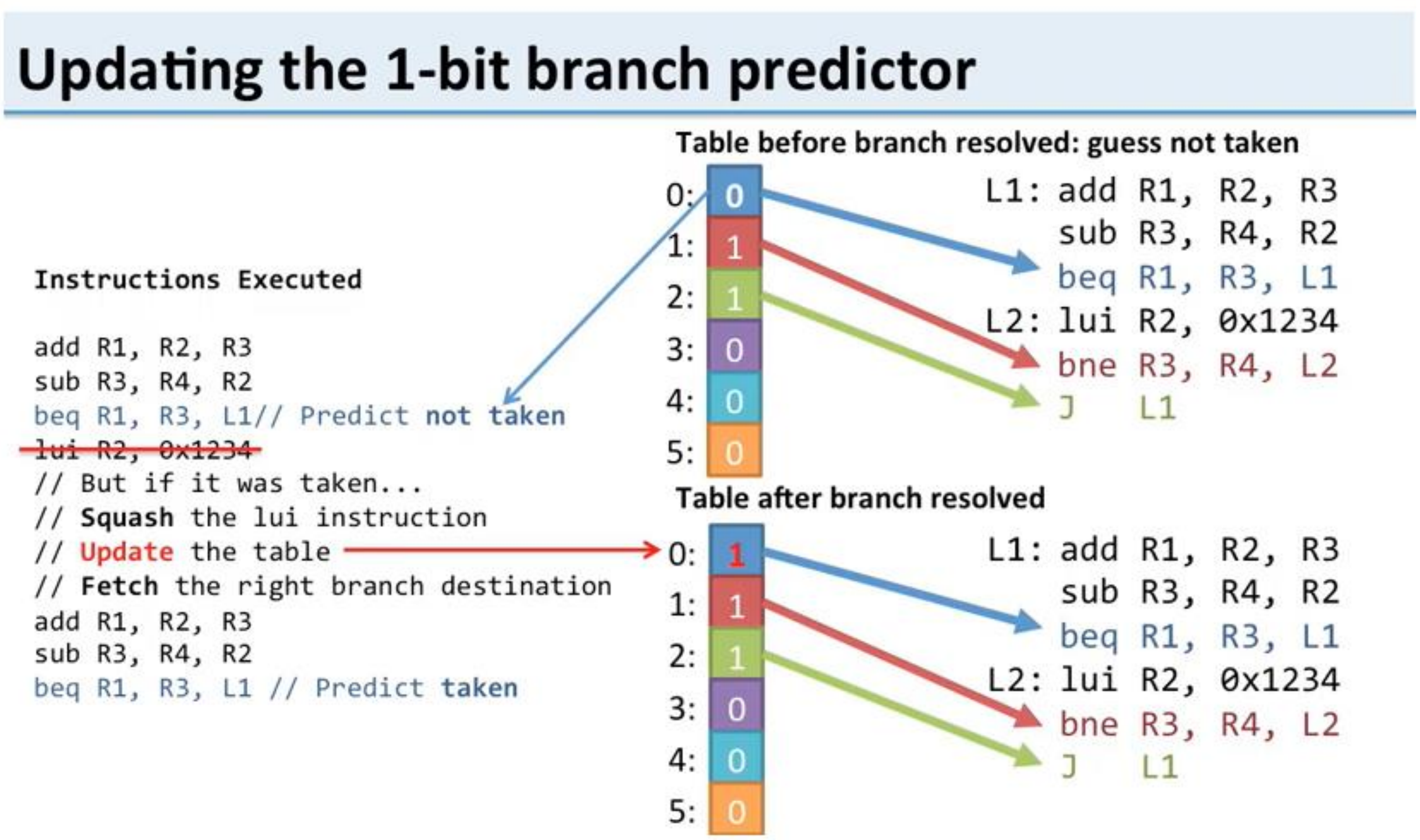
Updating 1-bit Branch Predictor

Iteration	Predictor Bit	Predicted Outcome	Actual Outcome	Update
1	N	N	**T**	**T**
2	T	T	T	T
3	T	T	T	T
4	T	T	T	T
5	T	T	T	T
6	T	T	T	T
7	T	T	T	T
8	T	T	T	T
9	T	T	T	T
10	T	T	**N**	**N**

Problem with 1-bit Branch Predictor

- Problem: 1-bit prediction is wrong whenever there is a transition in the branching pattern
- Example: NTNTNT
  - 1-bit predictor is never correct! (0%)
  - Tossing a coin (no prediction at all) gives 50%!
- However, real code has bias
  - Branches taken several times are likely taken again
- Solution: store more history
  - Try two bits!

1 Bit Branch Prediction and History Table



Updating the 1-bit branch predictor

```
Instructions Executed

add R1, R2, R3
sub R3, R4, R2
beq R1, R3, L1// Predict not taken
~~lui R2, 0x1234~~
// But if it was taken...
// Squash the lui instruction
// Update the table
// Fetch the right branch destination
add R1, R2, R3
sub R3, R4, R2
beq R1, R3, L1 // Predict taken
```

Table before branch resolved: guess not taken

0:	0
1:	1
2:	1
3:	0
4:	0
5:	0

```
L1: add R1, R2, R3
    sub R3, R4, R2
    beq R1, R3, L1
L2: lui R2, 0x1234
    bne R3, R4, L2
    J   L1
```

Table after branch resolved

0:	1
1:	1
2:	1
3:	0
4:	0
5:	0

```
L1: add R1, R2, R3
    sub R3, R4, R2
    beq R1, R3, L1
L2: lui R2, 0x1234
    bne R3, R4, L2
    J   L1
```

2-bit Branch Prediction

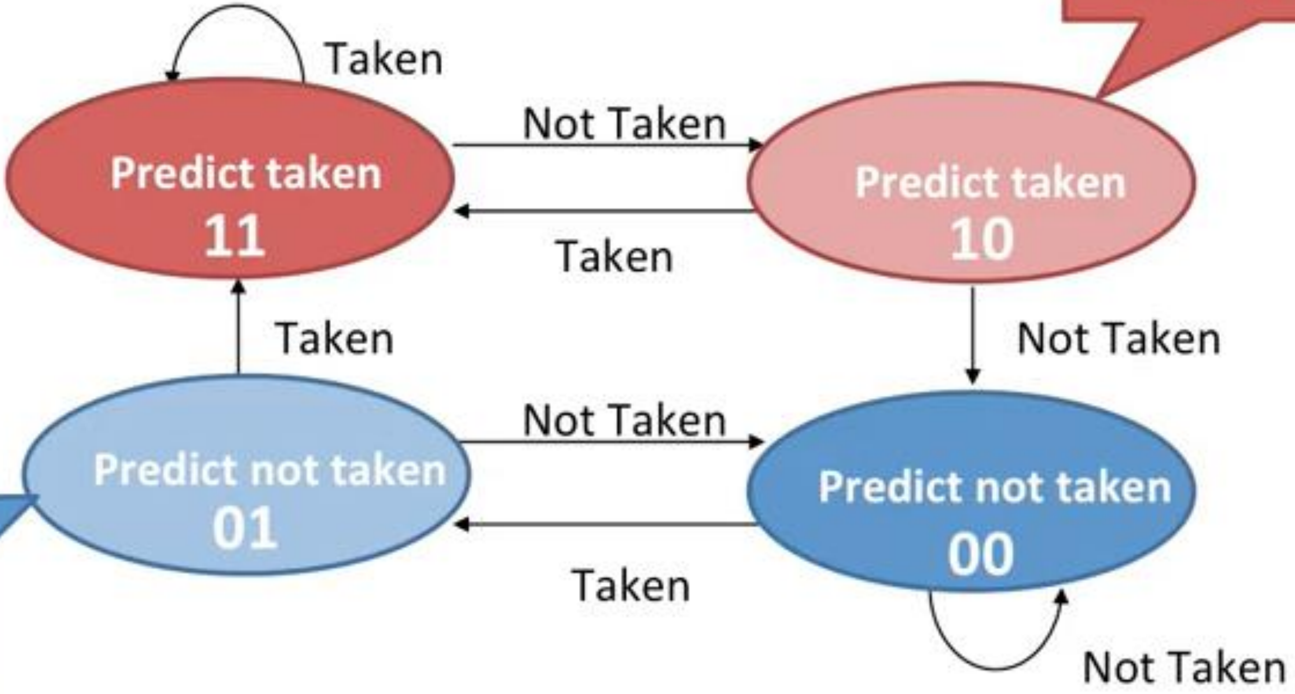
What is 2-bit Branch Prediction?

- Mispredict twice in a row to change prediction
  - Count the number of ‘taken’ (not taken) outcomes
- Branch taken (not taken) twice in a row
  - Predict “taken” (not taken)
- Branch not taken (taken) once
  - Continue to predict “taken” (not taken)
- *n* prediction bits
  - 2*n*-1 mispredictions before prediction changes

2 Bit Branch Prediction

- If 1-bit isn't good enough, how about 2-bits?
  - More "detailed" branch behavior

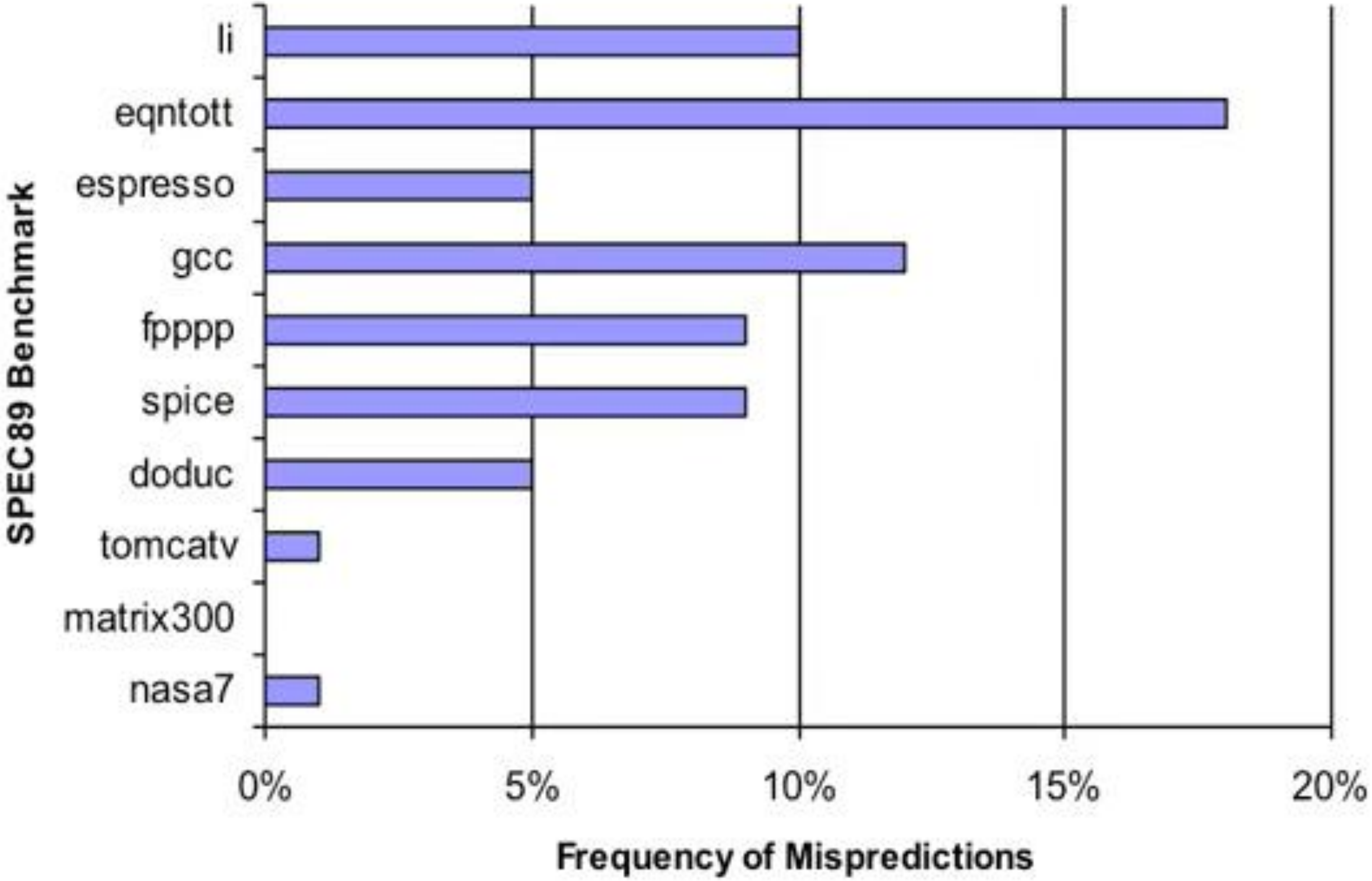
"weak" taken, or a chance to be wrong once but not switch the prediction



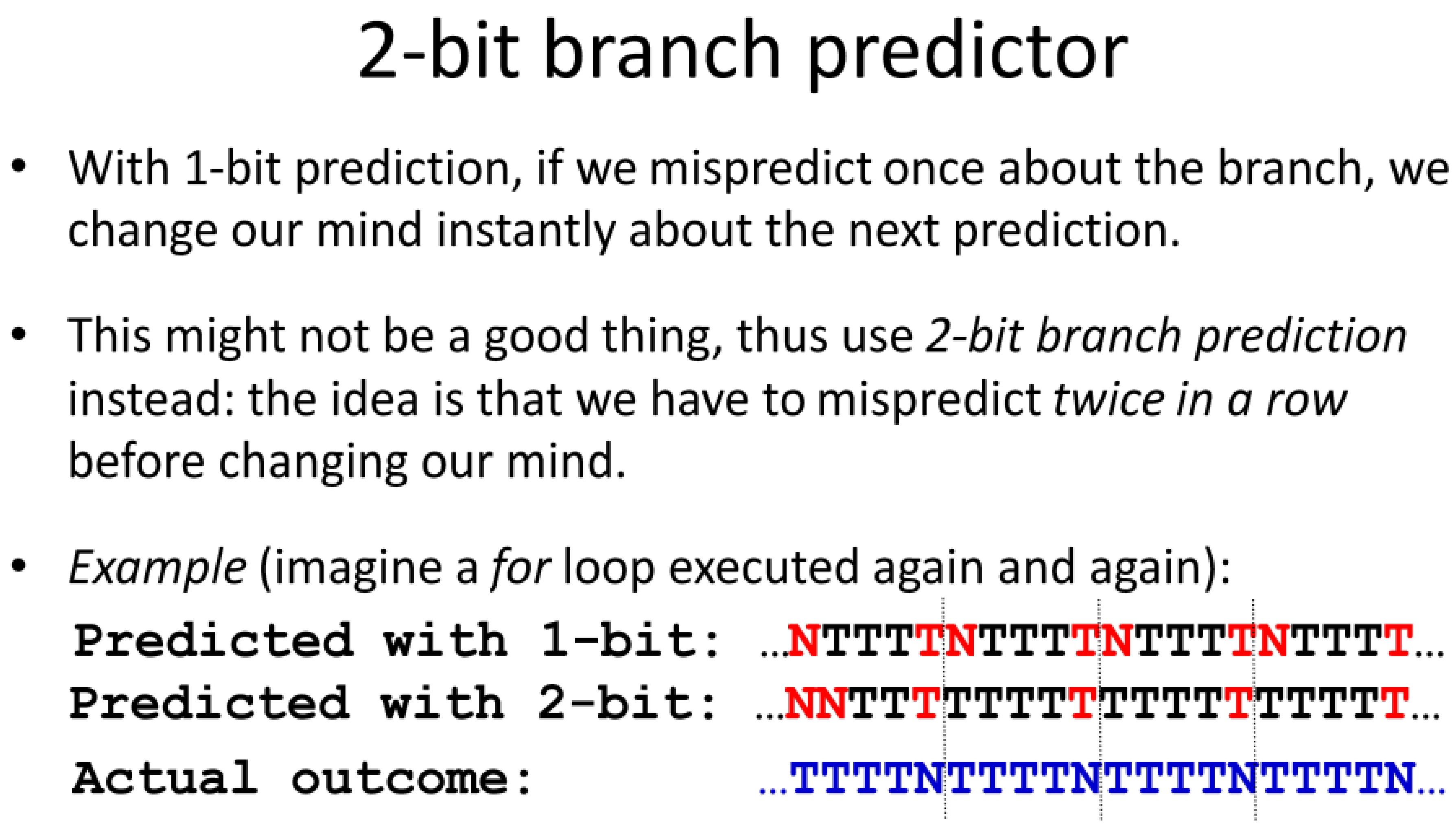
"weak" not taken, or a chance to be wrong once but not switch the prediction

Improvement through 2 Bit Predictor using Benchmarks

- 4096 entry (4096 separate branches)
- 2-bit predictor



2-bit Predictor



2-bit branch predictor

- With 1-bit prediction, if we mispredict once about the branch, we change our mind instantly about the next prediction.
- This might not be a good thing, thus use *2-bit branch prediction* instead: the idea is that we have to mispredict *twice in a row* before changing our mind.
- *Example* (imagine a *for* loop executed again and again):

```
Predicted with 1-bit:  …NTTTTNTTTTNTTTTNTTTT…
Predicted with 2-bit:  …NNTTTTTTTTTTTTTTTTTT…

Actual outcome:        …TTTTNTTTTNTTTTNTTTTN…
```

Example of 2-bit Branch Prediction

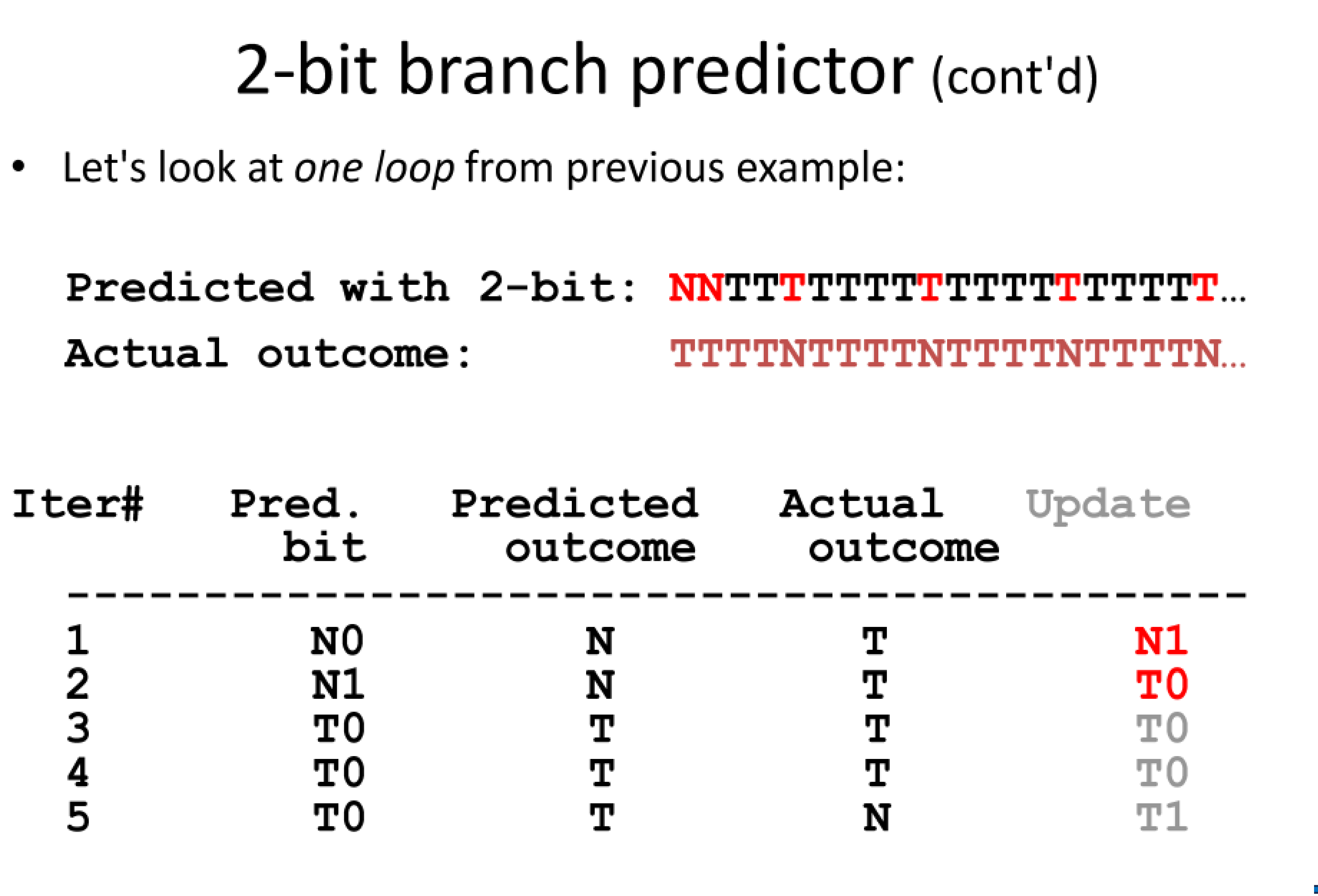
- Consider a for loop executed again and again

  Actual outcome: ...TTTTNTTTTNTTTTNTTTTN...
  Predicted with 1-bit: ...**N**TTTT**TN**TTT**TN**TTT**TN**TTT**T**...
  Predicted with 2-bit: ...TTTT**T**TTTT**T**TTTT**T**TTTTT...

- 1-bit predictor: 60% accuracy
- 2-bit predictor: 80% accuracy

Iteration	Predictor Bits	Predicted Outcome	Actual Outcome	Update
1	10	T	T	11
2	11	T	T	11
3	11	T	T	11
4	11	T	T	11
5	11	T	**N**	**10**

2-bit Prediction Example



2-bit branch predictor (cont'd)

- Let's look at *one loop* from previous example:

```
Predicted with 2-bit: NNTTTTTTTTTTTTTTTTTT…
Actual outcome:       TTTTNTTTTNTTTTNTTTTN…
```

Iter#	Pred. bit	Predicted outcome	Actual outcome	Update
1	N0	N	T	N1
2	N1	N	T	T0
3	T0	T	T	T0
4	T0	T	T	T0
5	T0	T	N	T1

Branch Target Buffer (BTB)

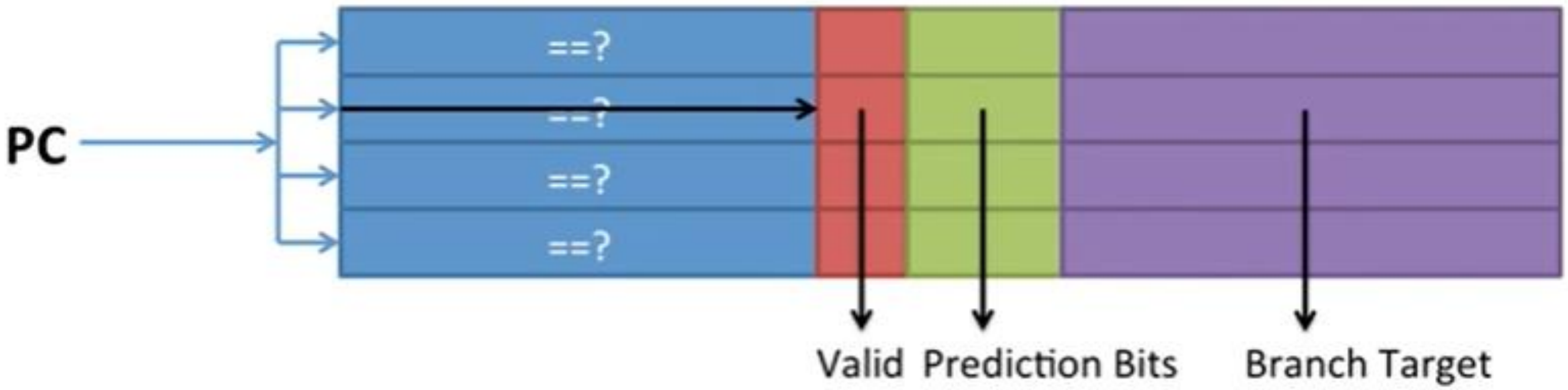
- For each PC, the BTB stores:
  - If the PC is a branch
  - If the branch is predicted taken/not taken
  - The address of the branch if taken
- Update prediction when the branch is resolved

- BTB is not infinitely big, so it can’t hold every branch

(We’ll see a lot more about associative memories in the lecture on caches)

Make a Branch Target Buffer

- We can now predict **if the branch is taken**, but how do we know **where it goes**?
- Add the branch target (address) to the table

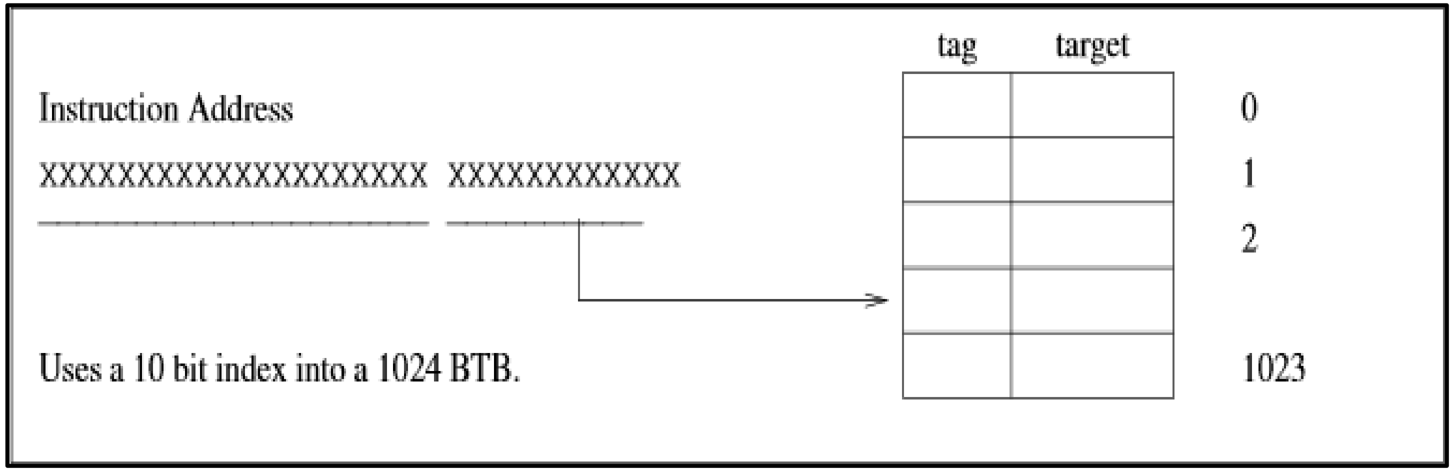


Branch Target Buffer – Advantage in ID Stage

even if we can predict a branch will be taken, we cannot write the branch target to PC until the ID stage. Therefore, we have a 1- cycle stall on predict-taken branches.

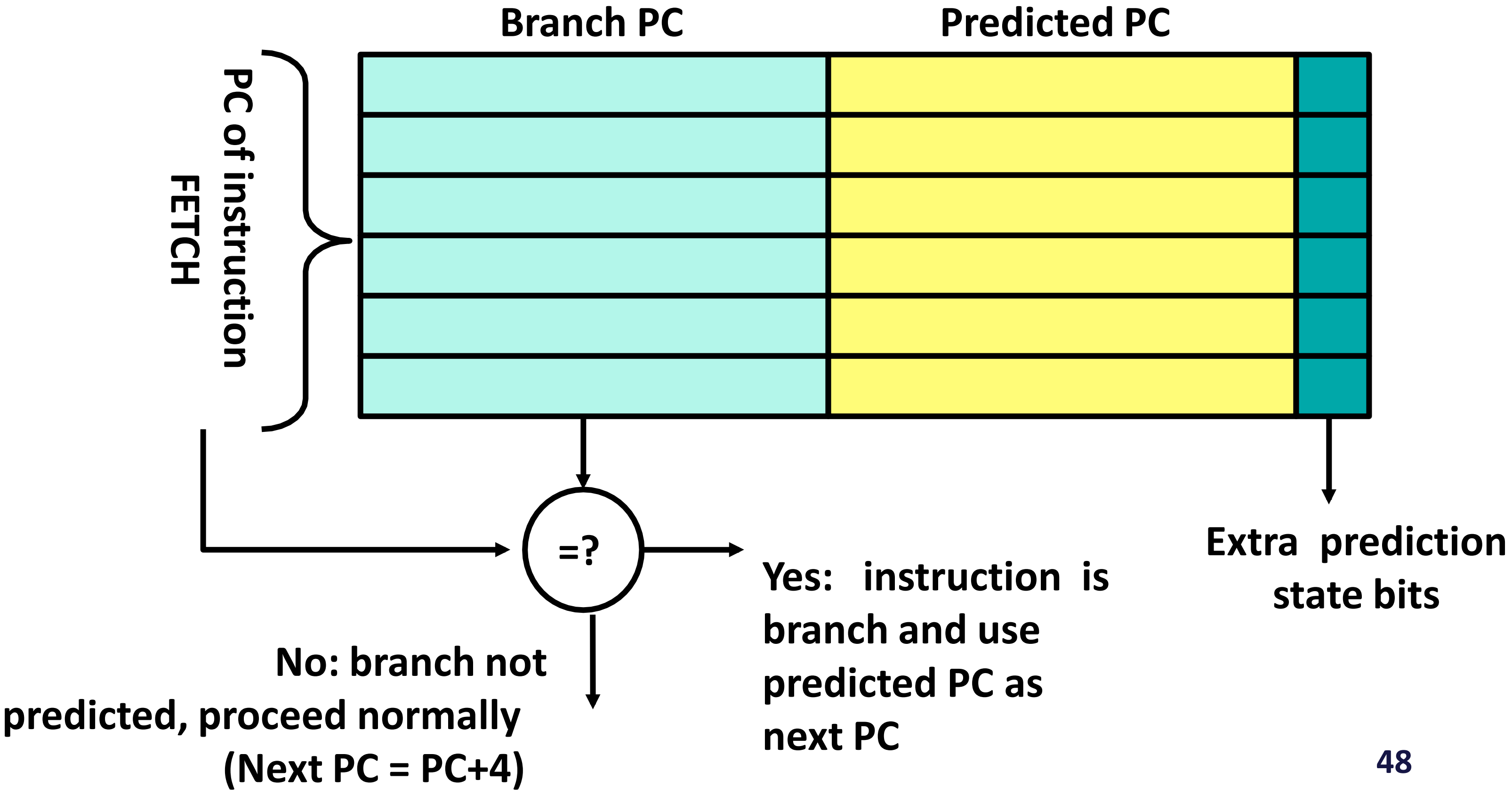
To avoid this, we use a *branch target buffer*. A branch target buffer contains a tag (the higher bits of the instruction) and the target address of the branch.

If the BPB predicts the branch is taken, and the higher order bits of the instruction match the BTB tag, then we write the target to PC.


tag
target
Instruction Address
XXXXXXXXXXXXXXXXXXXX XXXXXXXXXXXX
0
1
2
1023
Uses a 10 bit index into a 1024 BTB.

Branch Target Buffer Operation

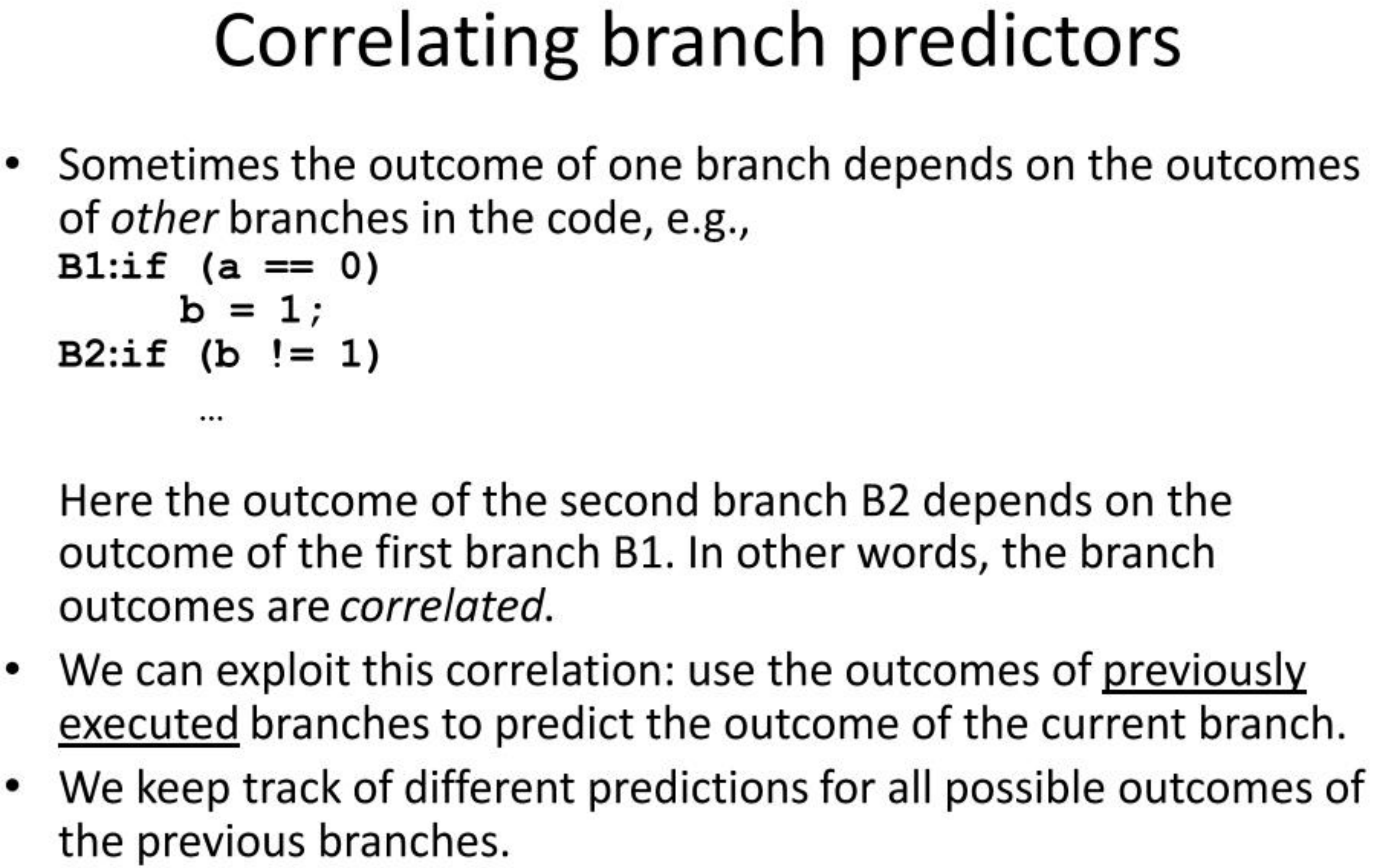
- **Branch Target Buffer (BTB): Address of branch index to get prediction AND branch address (if taken)**
  - **Note: must check for branch match now, since can't use wrong branch address**
- **Example: BTB combined with BHT**



Branch PC
Predicted PC
PC of instruction FETCH
=?
Yes: instruction is branch and use predicted PC as next PC
No: branch not predicted, proceed normally (Next PC = PC+4)
Extra prediction state bits
48

Correlating Branch Prediction

Correlating Branch Predictor



Correlating branch predictors

- Sometimes the outcome of one branch depends on the outcomes of *other* branches in the code, e.g.,

```
B1:if (a == 0)
        b = 1;
B2:if (b != 1)
         …
```

  Here the outcome of the second branch B2 depends on the outcome of the first branch B1. In other words, the branch outcomes are *correlated*.

- We can exploit this correlation: use the outcomes of <u>previously executed</u> branches to predict the outcome of the current branch.
- We keep track of different predictions for all possible outcomes of the previous branches.

Principle of Correlating Predictor

A note on the the (m, n) notation:

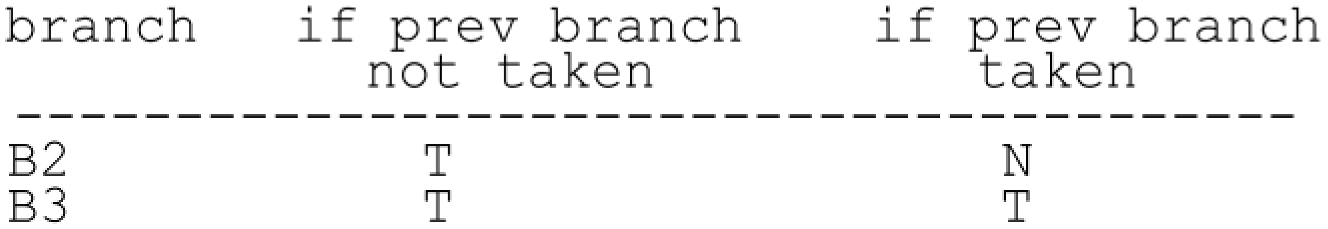
m: number of previous branches correlated

n: number of bit for predictor

- **(1, 1)** correlating branch predictor means
  - Use the outcomes of the previous 1 branch executed
  - And use a 1-bit branch predictor
- **(2, 1)** correlating branch predictor means
  - Use the outcomes of the previous 2 branches executed
  - And use a 1-bit branch predictor
- **(1, 2)** correlating branch predictor means
  - Use the outcomes of the previous 1 branch executed
  - And use a 2-bit branch predictor

Example of Correlating Branch Predictor

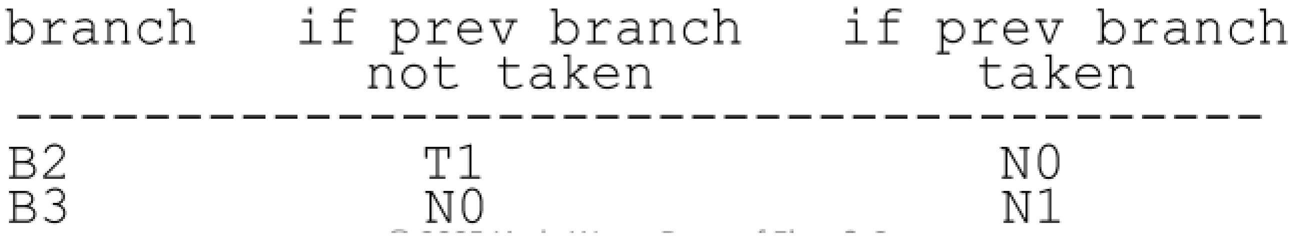
- (1, 1) correlating branch predictor: each branch history table entry has 2 fields of 1 bit each, e.g.,



branch	if prev branch not taken	if prev branch taken
B2	T	N
B3	T	T

B2 will be predicted taken if the previous branch executed was not taken and not taken if the previous branch executed was taken. B3 will be predicted taken in both cases.

- Similarly, for a (1, 2) predictor, we could have:



branch	if prev branch not taken	if prev branch taken
B2	T1	N0
B3	N0	N1

Tournament Branch Prediction

Introducing Tournament Branch Prediction

- 1-bit predictor failed to capture simple loop behavior
  - Increase local state to account for frequently taken branches
- 2-bit predictor failed to capture all branch behavior
  - Add global state to improve performance
- Correlating predictors
  - Prefer global history to local history … why not leverage both?
- Tournament predictors use two predictors
  - One based on global information
  - One based on local information
  - A selector dynamically chooses between the two
- Pentium4 and Power5 – 30Kb tournament predictors

Tournament Branch Predictor

Chooses between a local predictor and a global predictor on a per-branch basis

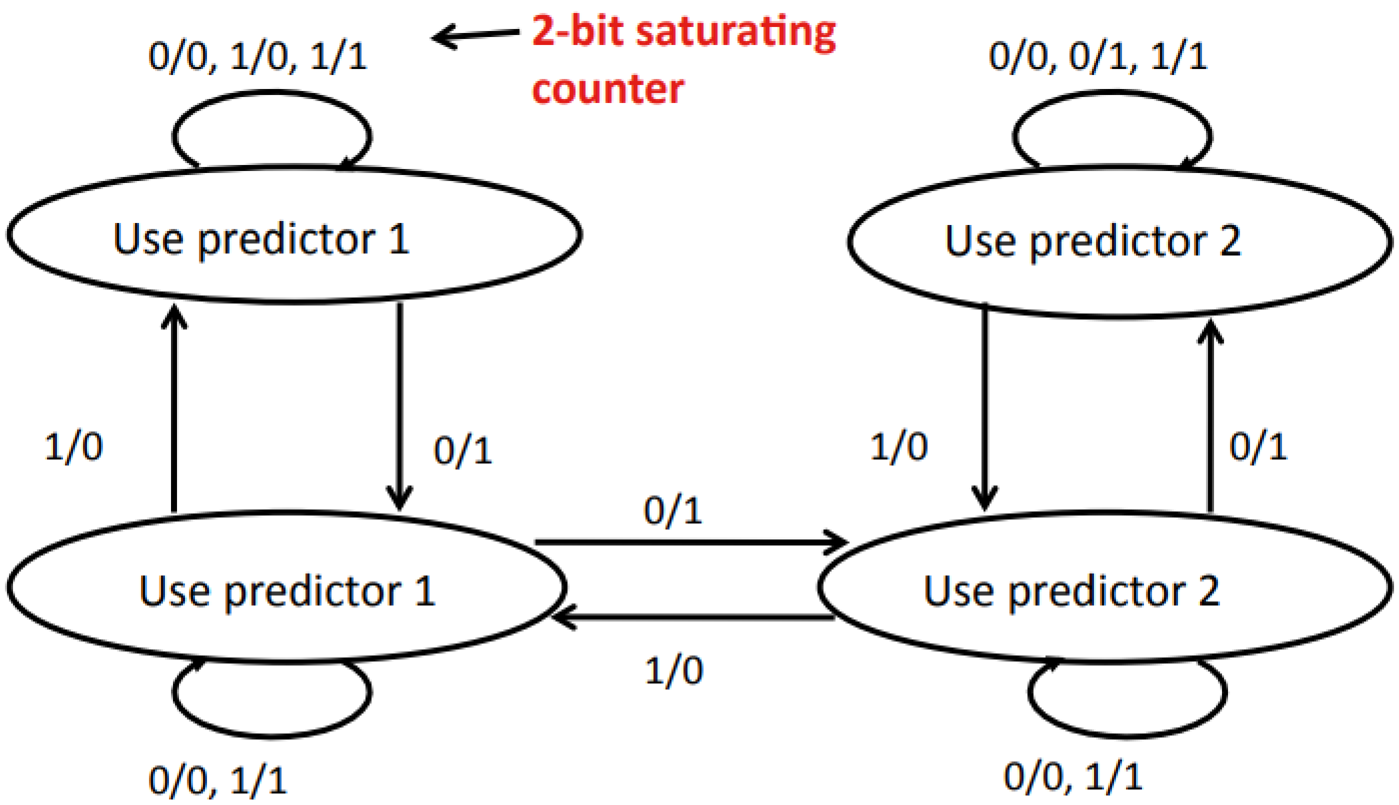
Keeps a table of n-bit branch predictors indexed by the last m global branch results

Uses a table of n-bit branch predictors indexed by the last k local branch results

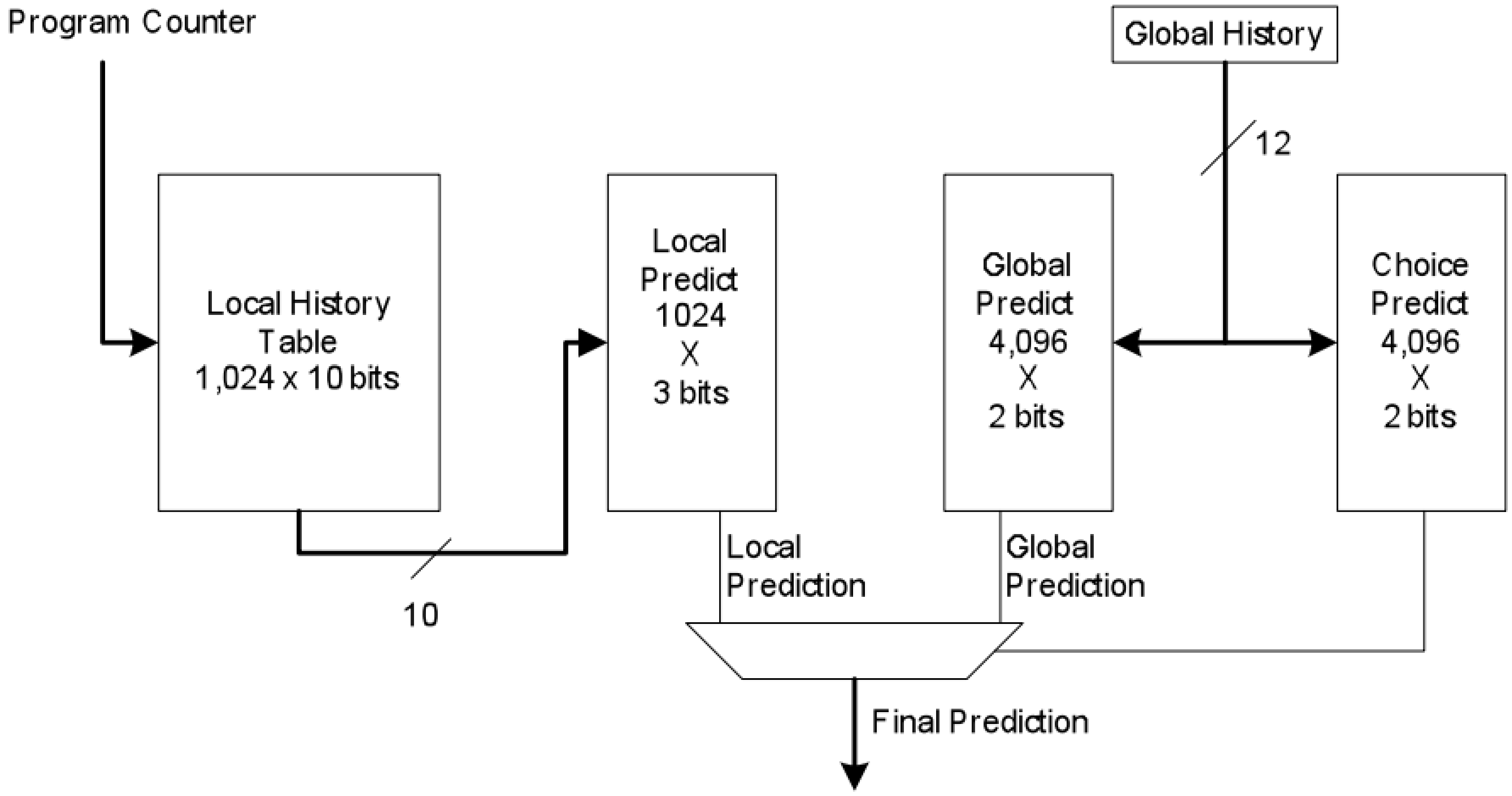
Uses another n-bit predictor to choose between the local and global predictors for each branch instruction

Tournament Predictor State Diagram

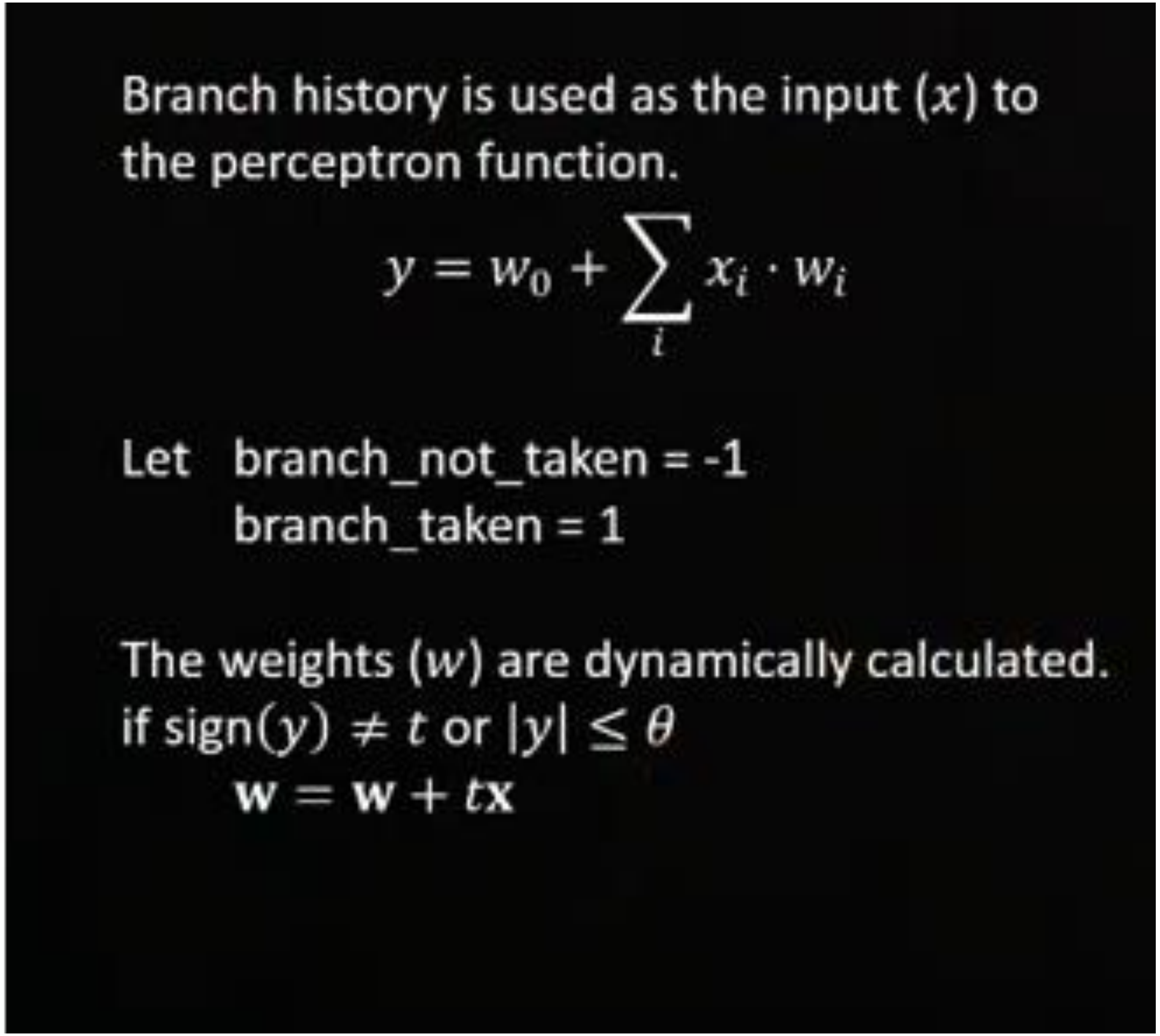
- Dynamically combine local and global predictors
  - Use 2-bit saturating counter for selector
  - Must miss twice before changing the predictor
- Different branches may prefer global, local, or a mix



Alpha processor Tournament Predictor


Program Counter
Global History
12
Local History Table 1,024 x 10 bits
Local Predict 1024 X 3 bits
Global Predict 4,096 X 2 bits
Choice Predict 4,096 X 2 bits
10
Local Prediction
Global Prediction
Final Prediction

Perceptron Based Branch Prediction



Branch history is used as the input (x) to the perceptron function.

$$y = w_0 + \sum_i x_i \cdot w_i$$

Let branch_not_taken = -1

branch_taken = 1

The weights (w) are dynamically calculated.

if sign(y) $\neq t$ or $|y| \leq \theta$

$$\mathbf{w} = \mathbf{w} + t\mathbf{x}$$

Perceptron Predictor Diagram

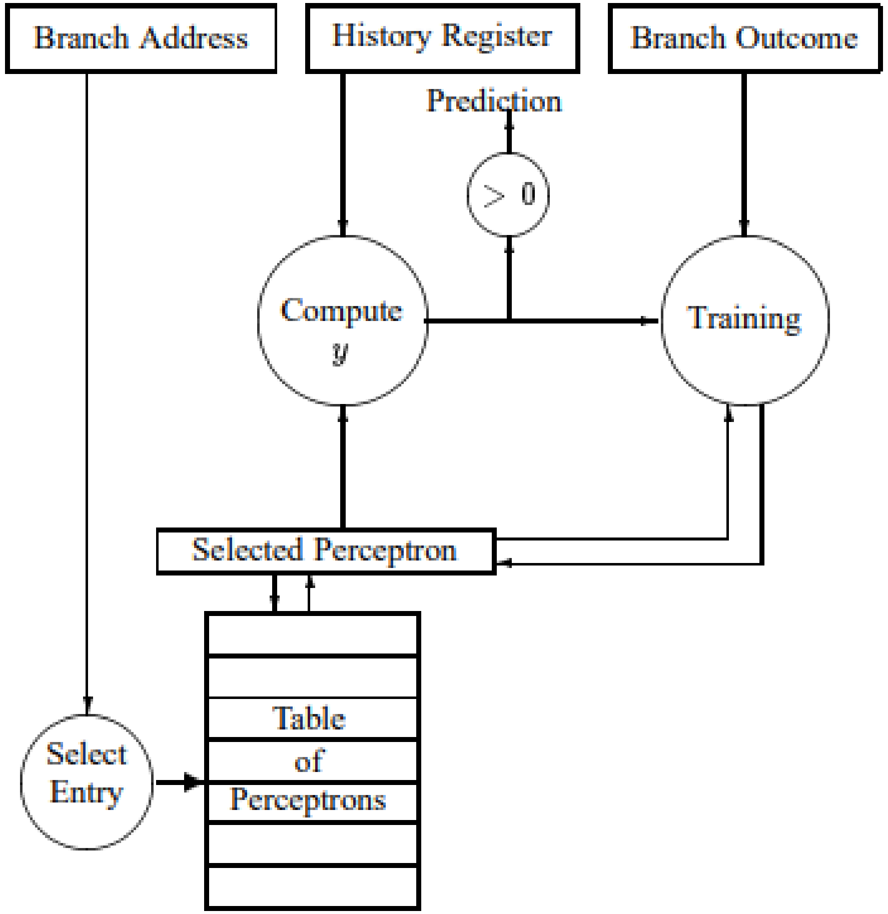


Figure 2: Perceptron Predictor Block Diagram. The branch address is hashed to select a perceptron that is read from the table. Together with the global history register, the output of the perceptron is computed, giving the prediction. The perceptron is updated with the training algorithm, then written back to the table.

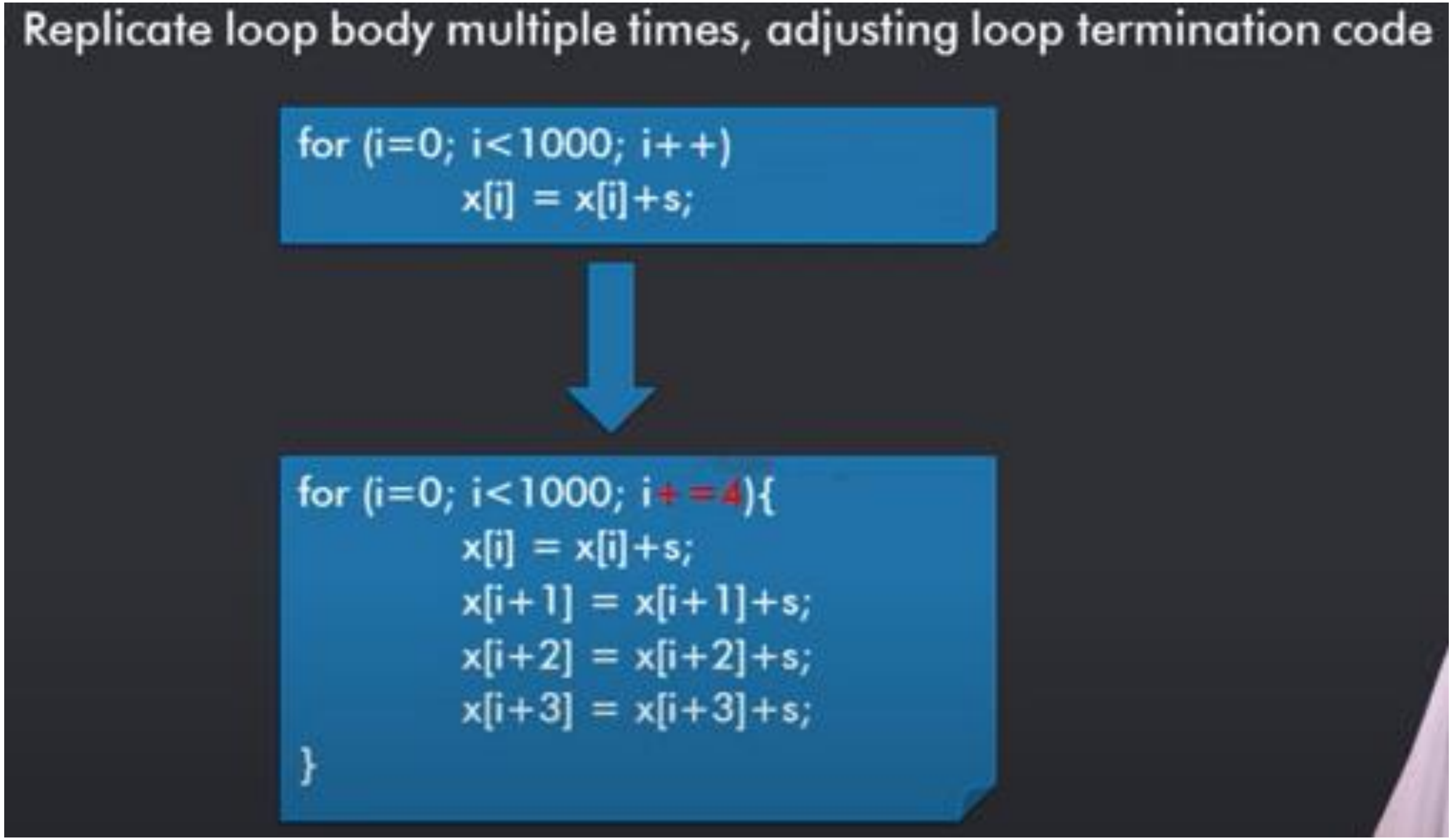
Ref:
Dynamic Branch Prediction with Perceptrons
Daniel A. Jim´enez,
Calvin Lin
The University of Texas at Austin

Pipelining in Loop Unrolling

Loop Unrolling

- Replicate loop body to expose more parallelism
  - Reduces loop-control overhead
- Use different registers per replication
  - Called “register renaming”
  - Avoid loop-carried “anti-dependencies”
    - Store followed by a load of the same register
    - Aka “name dependence”
      - Reuse of a register name

Loop Unrolling in C Code

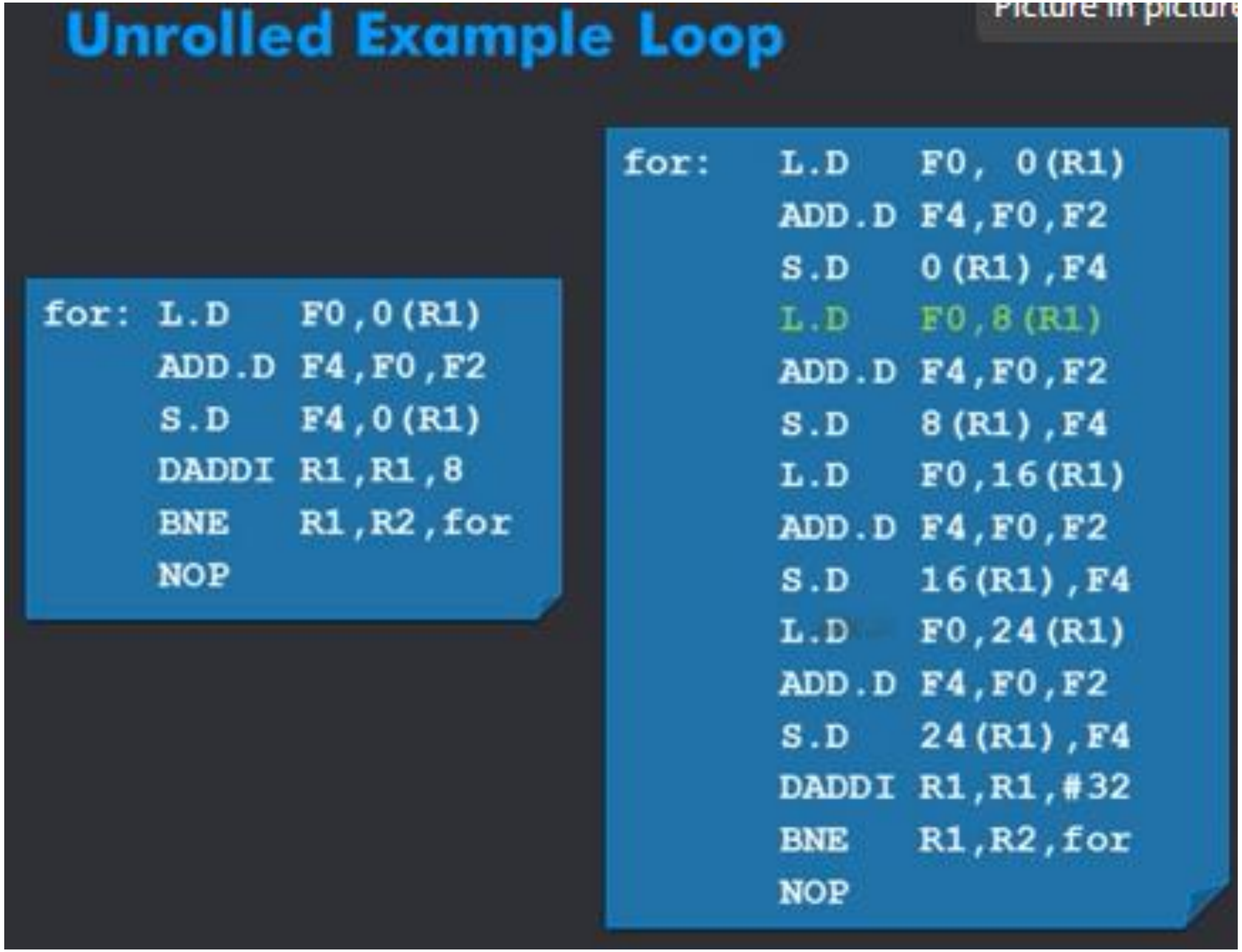


Replicate loop body multiple times, adjusting loop termination code

```
for (i=0; i<1000; i++)
          x[i] = x[i]+s;
```

```
for (i=0; i<1000; i+=4){
          x[i] = x[i]+s;
          x[i+1] = x[i+1]+s;
          x[i+2] = x[i+2]+s;
          x[i+3] = x[i+3]+s;
}
```

Loop Unrolled in Assembly Language



Unrolled Example Loop

```
for: L.D    F0,0(R1)
     ADD.D  F4,F0,F2
     S.D    F4,0(R1)
     DADDI  R1,R1,8
     BNE    R1,R2,for
     NOP
```

```
for:    L.D    F0, 0(R1)
        ADD.D  F4,F0,F2
        S.D    0(R1),F4
        L.D    F0,8(R1)
        ADD.D  F4,F0,F2
        S.D    8(R1),F4
        L.D    F0,16(R1)
        ADD.D  F4,F0,F2
        S.D    16(R1),F4
        L.D    F0,24(R1)
        ADD.D  F4,F0,F2
        S.D    24(R1),F4
        DADDI  R1,R1,#32
        BNE    R1,R2,for
        NOP
```

Another Assembly Lange Eg. Of Loop Unrolling

An Array Sum Loop

The following C code will compute the sum of the entries in a 100-entry vector A.

```
double arraySum = 0;
for (int i = 0; i < 100; i++) {
  arraySum += A[i];
}
```

Unrolled Assembly code on next slide

MIPS assembly language code initializations

The code below omits the loop initializations:

Initialize loop count ($7) to 100.
Initialize arraySum ($f10) to 0.
Initialize A[i] pointer ($5) to the base address of A.

MIPS assembly language code for Array Sum Loop

```
loop3:
    l.d     $f10, 0($5)       ; $f10 ← A[i]
    add.d   $f8, $f8, $f10    ; $f8 ← $f8 + A[i]
    addi    $5, $5, 8         ; increment pointer for A[i]
    addi    $7, $7, -1        ; decrement loop count
test:
    bgtz    $7, loop3         ; Continue if loop count > 0
```

The addi instructions in this code are loop maintenance: advancing addresses and counting iterations.

Unrolled Assembly Code from previous slide

The Array Sum Loop Unrolled

This is the same code with loop unrolling with a loop unrolling factor of 4.

Notice that the displacements in the loads are incremented by the compiler in the second, third, and fourth copies of the original loop body instructions. The immediate values in the addi instructions have been multiplied by 4 so that the effect of one iteration in the unrolled loop is the same as 4 iterations of the original loop.

Notice that branches and loop maintenance instructions have been reduced by a factor of 4.

```
loop3:
    l.d     $f10, 0($5)        ; iteration with displacement 0
    add.d   $f8, $f8, $f10

    l.d     $f10, 8($5)        ; iteration with displacement 8
    add.d   $f8, $f8, $f10

    l.d     $f10, 16($5)       ; iteration with displacement 16
    add.d   $f8, $f8, $f10

    l.d     $f10, 24($5)       ; iteration with displacement 24
    add.d   $f8, $f8, $f10

    addi    $5, $5, 32
    addi    $7, $7, -4
test:
    bgtz    $7, loop3          ; Continue loop if $7 > 0
```

Pipelining in Loop Unrolling

- Unroll few sequential instructions by a small loop unrolling factor k
- Use instruction re-ordering and register-renaming to reduce dependencies.
- Thus NOP wait will not be required within the unrolled portion of loop.

Loop Unrolling Example

	ALU/branch	Load/store	cycle
Loop:	addi $s1, $s1,–16	lw $t0, 0($s1)	1
	nop	lw $t1, 12($s1)	2
	addu $t0, $t0, $s2	lw $t2, 8($s1)	3
	addu $t1, $t1, $s2	lw $t3, 4($s1)	4
	addu $t2, $t2, $s2	sw $t0, 16($s1)	5
	addu $t3, $t4, $s2	sw $t1, 12($s1)	6
	nop	sw $t2, 8($s1)	7
	bne $s1, $zero, Loop	sw $t3, 4($s1)	8

- IPC = 14/8 = 1.75
  - Closer to 2, but at cost of registers and code size

Instruction-Level Parallelism (ILP)

- Pipelining: executing multiple instructions in parallel
- To increase ILP
  - Deeper pipeline
    - Less work per stage $\Rightarrow$ shorter clock cycle
  - Multiple issue
    - Replicate pipeline stages $\Rightarrow$ multiple pipelines
    - Start multiple instructions per clock cycle
    - CPI < 1, so use Instructions Per Cycle (IPC)
    - E.g., 4GHz 4-way multiple-issue
      - 16 BIPS, peak CPI = 0.25, peak IPC = 4
    - But dependencies reduce this in practice

What is Multiple Issue?

- Static multiple issue
  - Compiler groups instructions to be issued together
  - Packages them into “issue slots”
  - Compiler detects and avoids hazards
- Dynamic multiple issue
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime

Speculation

- “Guess” what to do with an instruction
  - Start operation as soon as possible
  - Check whether guess was right
    - If so, complete the operation
    - If not, roll-back and do the right thing
- Common to static and dynamic multiple issue
- Examples
  - Speculate on branch outcome
    - Roll back if path taken is different
  - Speculate on load
    - Roll back if location is updated

Compiler/Hardware Speculation

- Compiler can reorder instructions
  - e.g., move load before branch
  - Can include “fix-up” instructions to recover from incorrect guess
- Hardware can look ahead for instructions to execute
  - Buffer results until it determines they are actually needed
  - Flush buffers on incorrect speculation

Speculation and Exceptions

- What if exception occurs on a speculatively executed instruction?
  - e.g., speculative load before null-pointer check
- Static speculation
  - Can add ISA support for deferring exceptions
- Dynamic speculation
  - Can buffer exceptions until instruction completion (which may not occur)

Exceptions in a Pipeline

- Another form of control hazard
- Consider overflow on add in EX stage

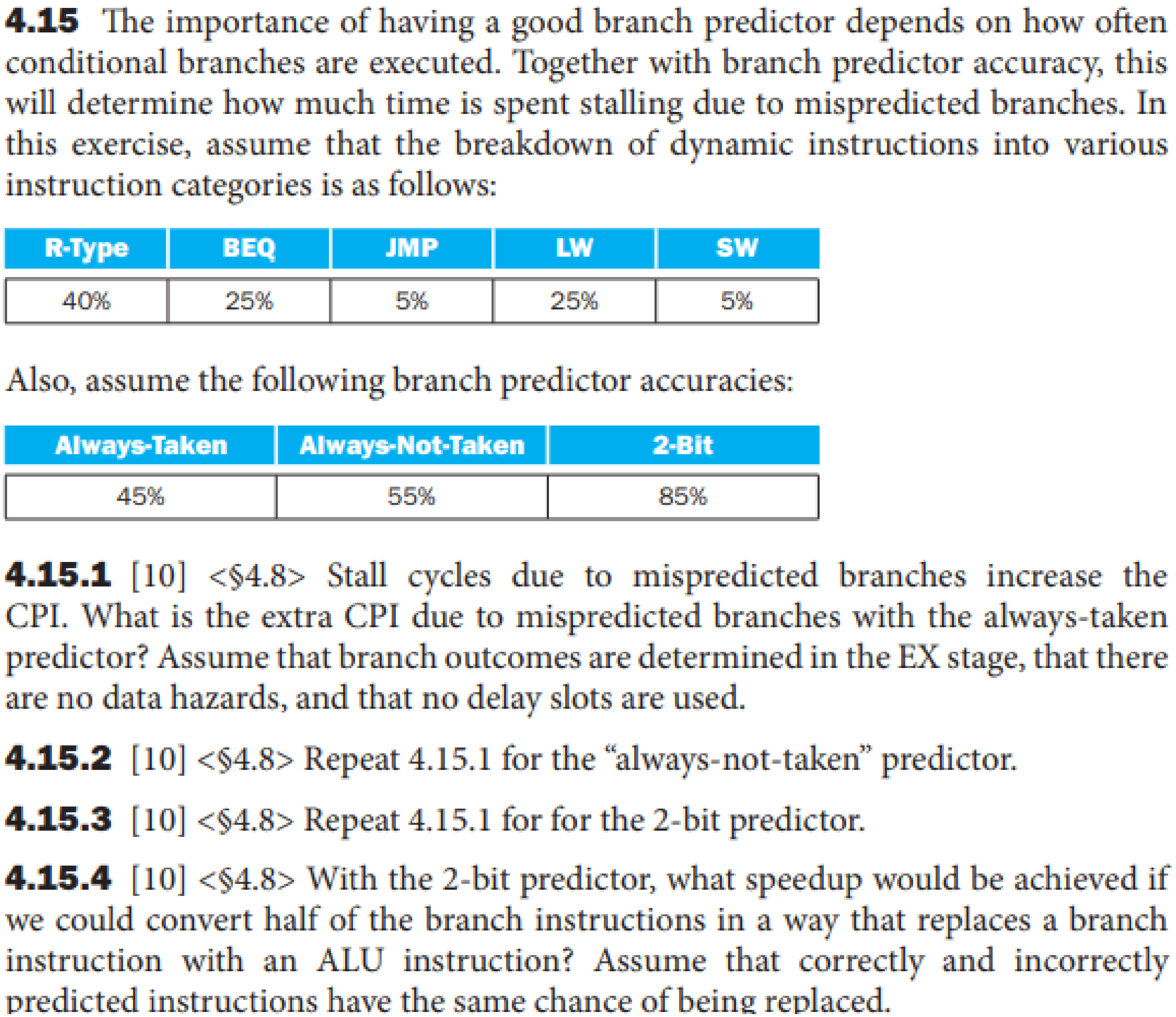
  `add $1, $2, $1`

  - Prevent $1 from being clobbered
  - Complete previous instructions
  - Flush add and subsequent instructions
  - Set Cause and EPC register values
  - Transfer control to handler
- Similar to mispredicted branch
  - Use much of the same hardware

Exception Properties

- Restartable exceptions
  - Pipeline can flush the instruction
  - Handler executes, then returns to the instruction
    - Refetched and executed from scratch
- PC saved in EPC register
  - Identifies causing instruction
  - Actually PC + 4 is saved
    - Handler must adjust

Example Questions - 1



4.15 The importance of having a good branch predictor depends on how often conditional branches are executed. Together with branch predictor accuracy, this will determine how much time is spent stalling due to mispredicted branches. In this exercise, assume that the breakdown of dynamic instructions into various instruction categories is as follows:

R-Type	BEQ	JMP	LW	SW
40%	25%	5%	25%	5%

Also, assume the following branch predictor accuracies:

Always-Taken	Always-Not-Taken	2-Bit
45%	55%	85%

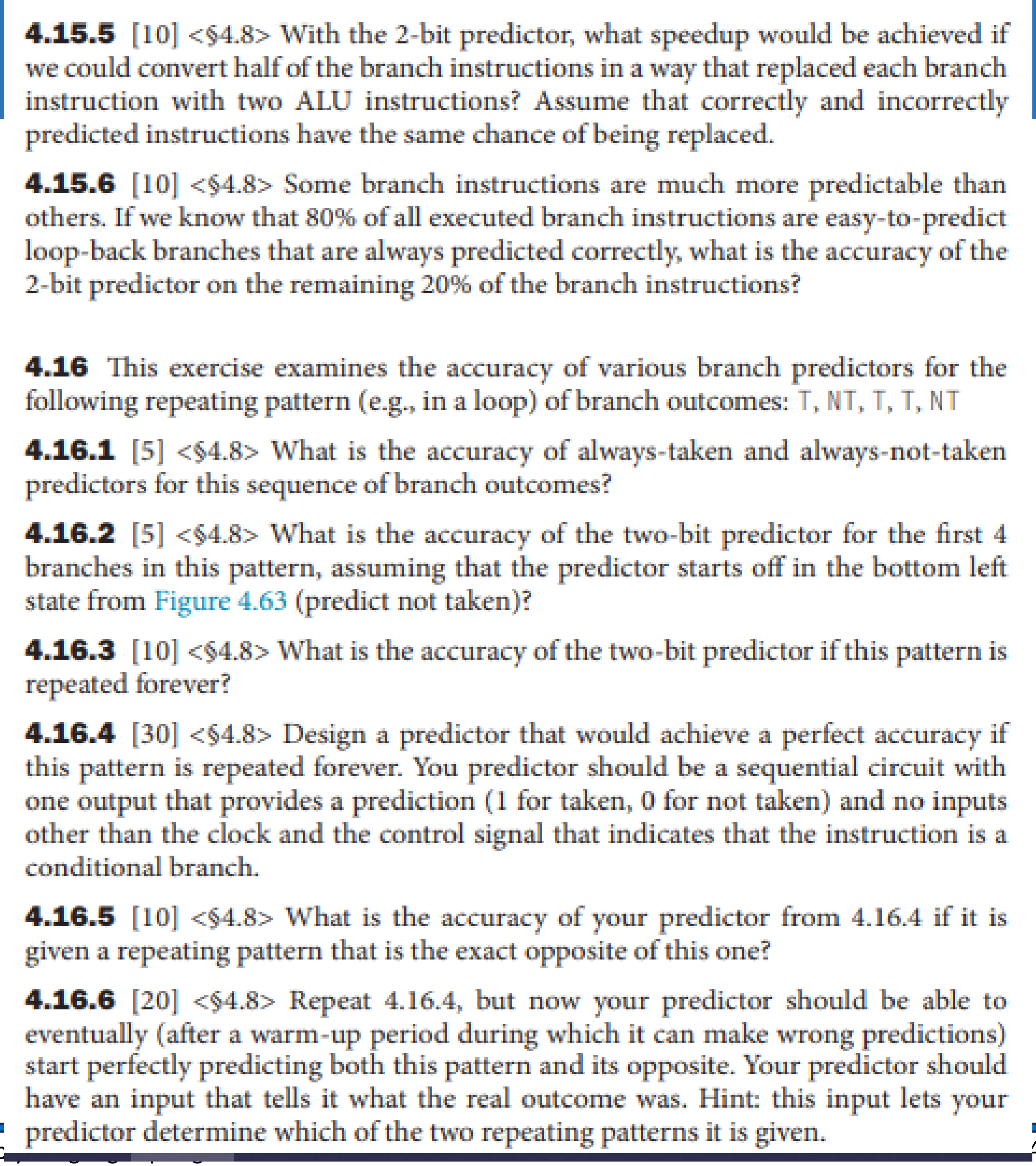
4.15.1 [10] <§4.8> Stall cycles due to mispredicted branches increase the CPI. What is the extra CPI due to mispredicted branches with the always-taken predictor? Assume that branch outcomes are determined in the EX stage, that there are no data hazards, and that no delay slots are used.

4.15.2 [10] <§4.8> Repeat 4.15.1 for the "always-not-taken" predictor.

4.15.3 [10] <§4.8> Repeat 4.15.1 for for the 2-bit predictor.

4.15.4 [10] <§4.8> With the 2-bit predictor, what speedup would be achieved if we could convert half of the branch instructions in a way that replaces a branch instruction with an ALU instruction? Assume that correctly and incorrectly predicted instructions have the same chance of being replaced.

Example Questions - 2



4.15.5 [10] <§4.8> With the 2-bit predictor, what speedup would be achieved if we could convert half of the branch instructions in a way that replaced each branch instruction with two ALU instructions? Assume that correctly and incorrectly predicted instructions have the same chance of being replaced.

4.15.6 [10] <§4.8> Some branch instructions are much more predictable than others. If we know that 80% of all executed branch instructions are easy-to-predict loop-back branches that are always predicted correctly, what is the accuracy of the 2-bit predictor on the remaining 20% of the branch instructions?

4.16 This exercise examines the accuracy of various branch predictors for the following repeating pattern (e.g., in a loop) of branch outcomes: T, NT, T, T, NT

4.16.1 [5] <§4.8> What is the accuracy of always-taken and always-not-taken predictors for this sequence of branch outcomes?

4.16.2 [5] <§4.8> What is the accuracy of the two-bit predictor for the first 4 branches in this pattern, assuming that the predictor starts off in the bottom left state from Figure 4.63 (predict not taken)?

4.16.3 [10] <§4.8> What is the accuracy of the two-bit predictor if this pattern is repeated forever?

4.16.4 [30] <§4.8> Design a predictor that would achieve a perfect accuracy if this pattern is repeated forever. You predictor should be a sequential circuit with one output that provides a prediction (1 for taken, 0 for not taken) and no inputs other than the clock and the control signal that indicates that the instruction is a conditional branch.

4.16.5 [10] <§4.8> What is the accuracy of your predictor from 4.16.4 if it is given a repeating pattern that is the exact opposite of this one?

4.16.6 [20] <§4.8> Repeat 4.16.4, but now your predictor should be able to eventually (after a warm-up period during which it can make wrong predictions) start perfectly predicting both this pattern and its opposite. Your predictor should have an input that tells it what the real outcome was. Hint: this input lets your predictor determine which of the two repeating patterns it is given.

Readings

- Chap 4 of P&H Textbook

- Examples from Tutorial 5 Exceptions Branch Prediction, McGill University