361.1.4201
Computer Architecture
Branch Prediction II
Dr. Guy Tel-Zur

Based on slides by Prof. Onur Mutlu

Carnegie Mellon University

Spring 2015

Dr. Guy Tel-Zur

Agenda for Today & Next Few Lectures

- Single-cycle Microarchitectures
- Multi-cycle and Microprogrammed Microarchitectures
- Pipelining
- Issues in Pipelining: Control & Data Dependence Handling
- **Branch prediction**
- State Maintenance and Recovery, …
- Out-of-Order Execution

אנימציה

Reminder: Readings for Next Few Lectures (I)

- **Guy: P&H CO&D, Chapter 4 – "The Processor"**
- **Guy: H&P CAQA, Appendix C - "Pipelining: Basic and**
- **Intermediate Concepts"**

- Smith and Sohi, "The Microarchitecture of Superscalar Processors," Proceedings of the IEEE, 1995
  - More advanced pipelining
  - Interrupt and exception handling
  - Out-of-order and superscalar execution concepts

- McFarling, "Combining Branch Predictors," DEC WRL Technical Report, 1993.

Reminder: Readings for Next Few Lectures (II)

- Smith and Plezskun, "Implementing Precise Interrupts in Pipelined Processors," IEEE Trans on Computers 1988 (earlier version in ISCA 1985).

Recap of Last Lecture

- Predicated Execution Primer

  ביצוע קבוע מראש (תזכורת פירוק תנאי "אם" לרגיסטר)

- Delayed Branching
  - With and without squashing
- Branch Prediction
  - Reducing misprediction penalty (branch resolution latency)
  - Branch target buffer (BTB)
- Static Branch Prediction
- Dynamic Branch Prediction
- How Big Is the Branch Problem?

Review: More Sophisticated Direction Prediction

- Compile time (static)
  - Always not taken
  - Always taken
  - BTFN (Backward taken, forward not taken)
  - Profile based (likely direction)
  - Program analysis based (likely direction)
- **Today: Run time (dynamic)**
  - Last time prediction (single-bit)
  - Two-bit counter based prediction
  - Two-level prediction (global vs. local)
  - Hybrid
  - Advanced algorithms (e.g., using perceptrons)

Review: Importance of The Branch Problem

- Assume N = 20 (20 pipe stages), W = 5 (5 wide fetch)
- Assume: 1 out of 5 instructions is a branch
- Assume: Each 5 instruction-block ends with a branch

How long does it take to fetch 500 instructions?

- 100% accuracy
  - 100 cycles (all instructions fetched on the correct path)
  - No wasted work
- 99% accuracy
  - 100 (correct path) + 20 (wrong path) = 120 cycles
  - 20% extra instructions fetched
- 98% accuracy
  - 100 (correct path) + 20 * 2 (wrong path) = 140 cycles
  - 40% extra instructions fetched
- 95% accuracy
  - 100 (correct path) + 20 * 5 (wrong path) = 200 cycles
  - 100% extra instructions fetched

Importance of The Branch Problem - continued

- Assume N = 20 (20 pipe stages), W = 5 (5 wide fetch)
- Assume: 1 out of 5 instructions is a branch
- Assume: Each 5 instruction-block ends with a branch

- How long does it take to fetch 500 instructions?
  - 100% accuracy
    - 100 cycles (all instructions fetched on the correct path)
    - No wasted work; IPC = 500/100
  - 90% accuracy
    - 100 (correct path) + 20 * 10 (wrong path) = 300 cycles
    - 200% extra instructions fetched; IPC = 500/300
  - 85% accuracy
    - 100 (correct path) + 20 * 15 (wrong path) = 400 cycles
    - 300% extra instructions fetched; IPC = 500/400
  - 80% accuracy
    - 100 (correct path) + 20 * 20 (wrong path) = 500 cycles
    - 400% extra instructions fetched; IPC = 500/500

הספירה כאן היא
במחזורי שעון

שיטה דינמית

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 תוך כדי ריצת התכנית

  + No need for static profiling: input set representativeness problem goes away

- Disadvantages

  -- More complex (requires additional hardware)

Last Time Predictor

- Last time predictor
  - Single bit per branch (stored in BTB)
  - Indicates which direction branch went last time it executed

    TTTTTTTTTTNNNNNNNNNN → 90% accuracy

BTB=Branch Target Buffer

מהיכן ה-10%. תהיה החמצה בתחזית הראשונה ובתחזית של היפוך המגמה. 2 מתוך 20

- Always mispredicts the last iteration and the first iteration of a loop branch
  - Accuracy for a loop with N iterations = (N-2)/N

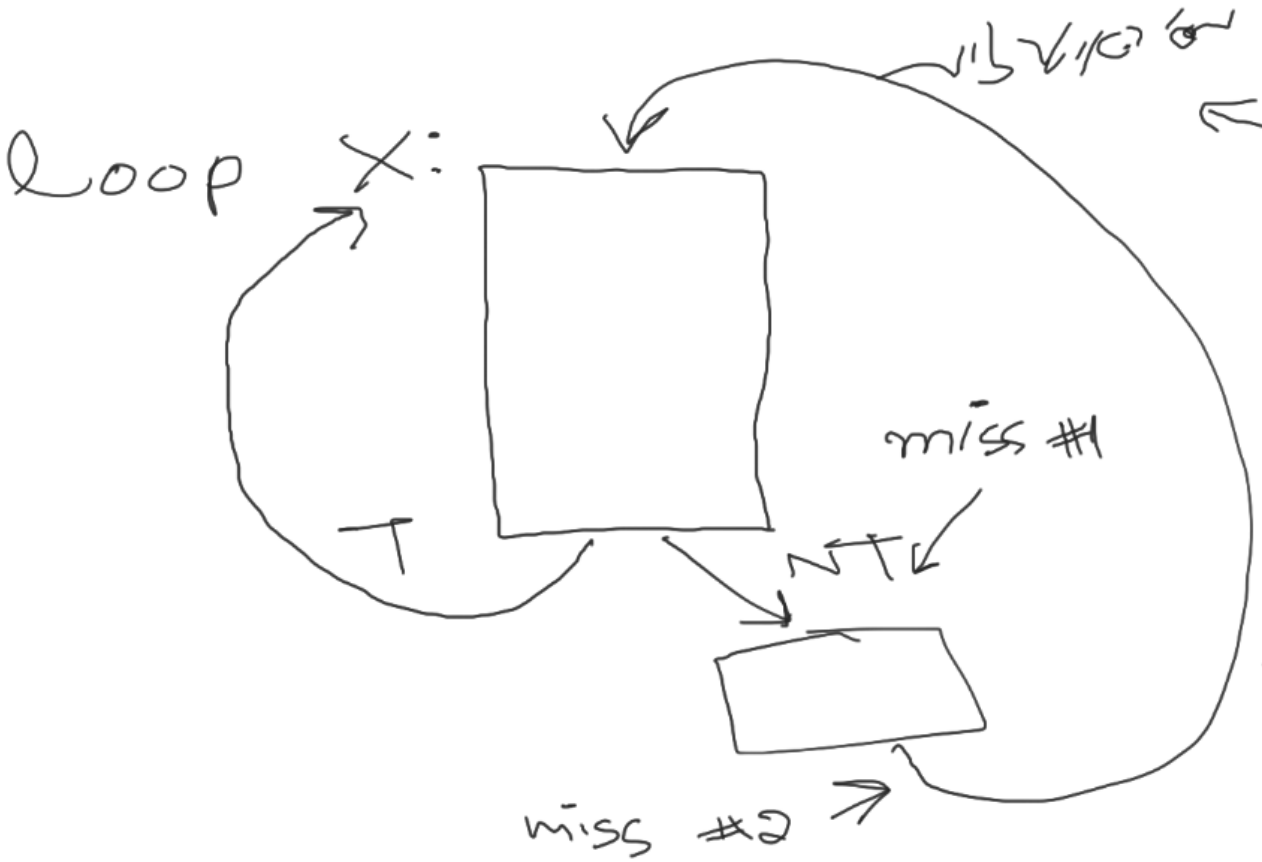
+ Loop branches for loops with large N (number of iterations)

-- Loop branches for loops with small N (number of iterations)

TNTNTNTNTNTNTNTNTNTN → 0% accuracy

Last-time predictor CPI = [1 + (0.20*0.15) * 2] = 1.06 (Assuming 85% accuracy)

Last Time Predictor

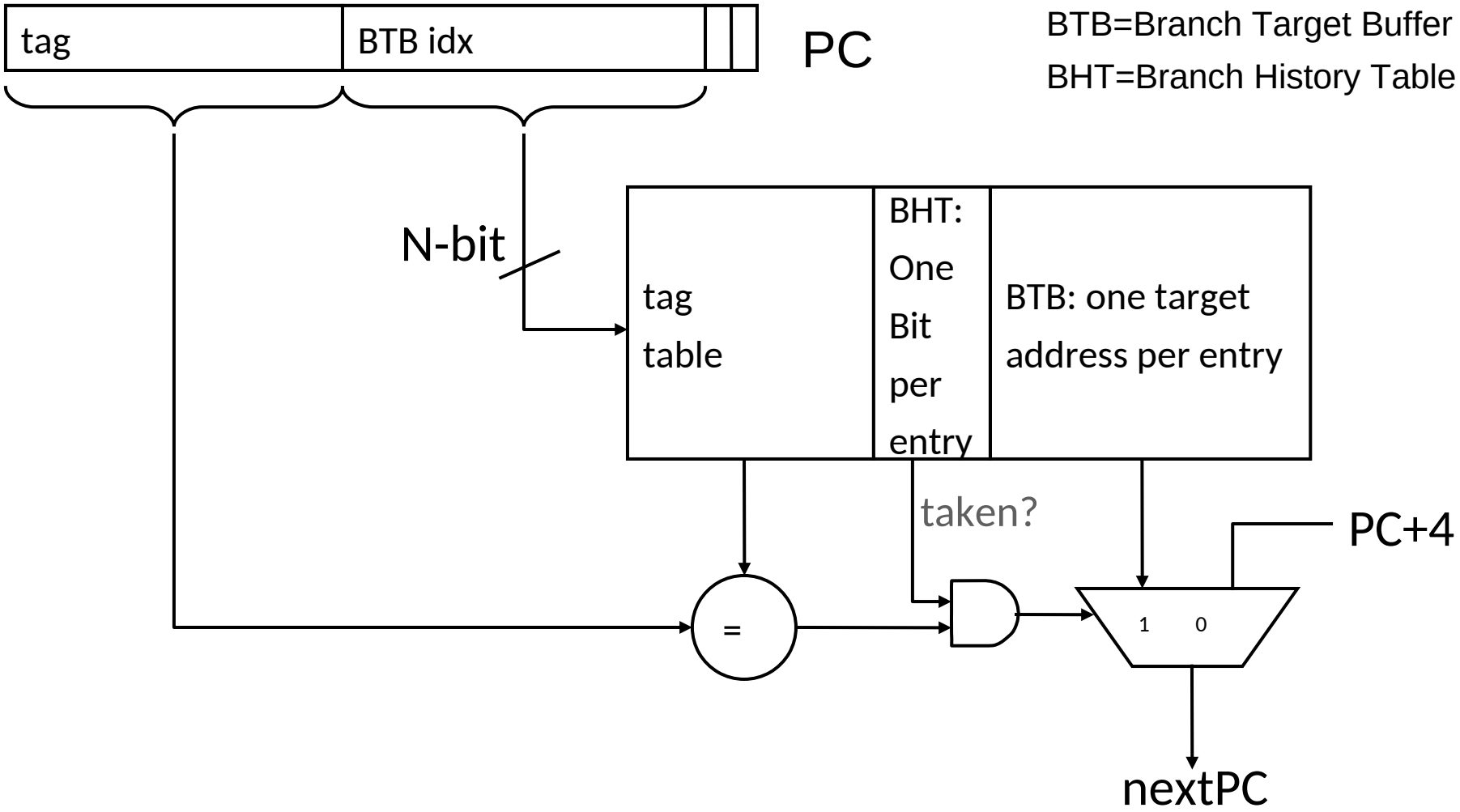


$\frac{N-2}{N}$ =

ok if N >>>

but what if N is small?

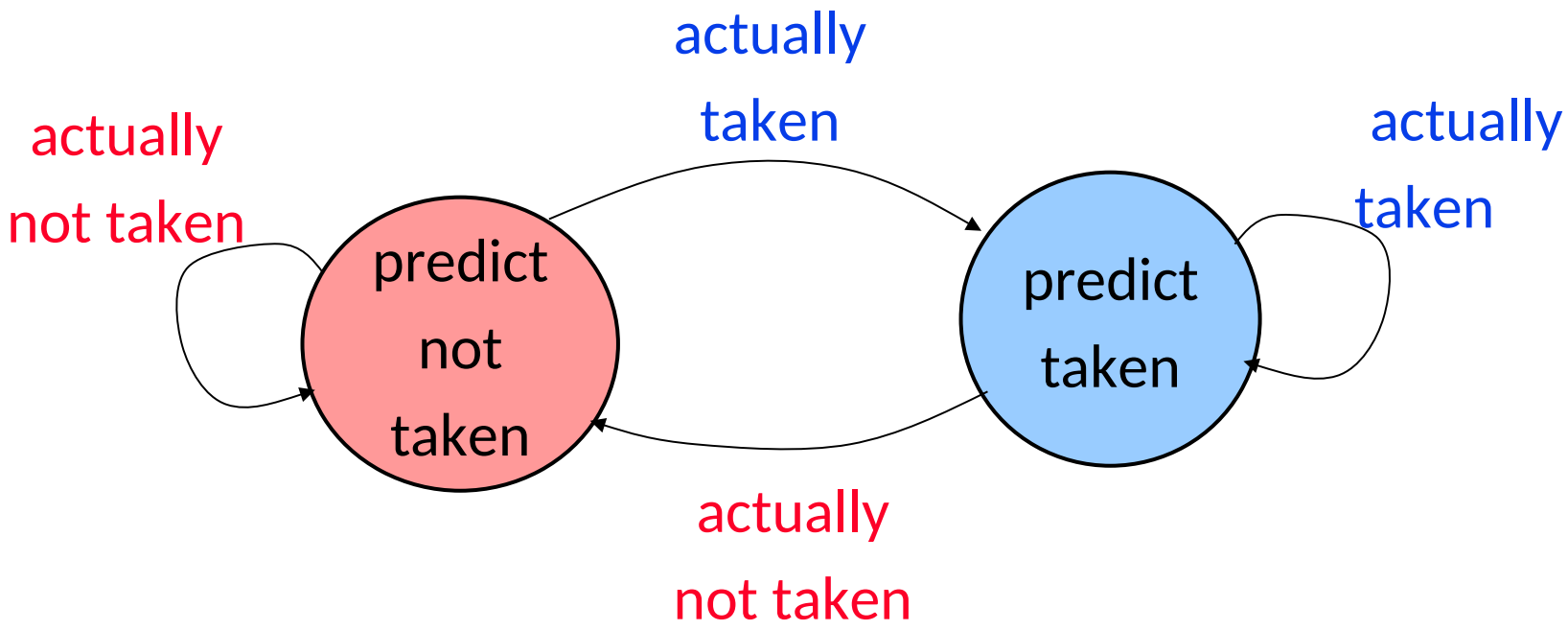
Implementing the Last-Time Predictor



The 1-bit BHT (Branch History Table) entry is updated with the correct outcome after each execution of a branch

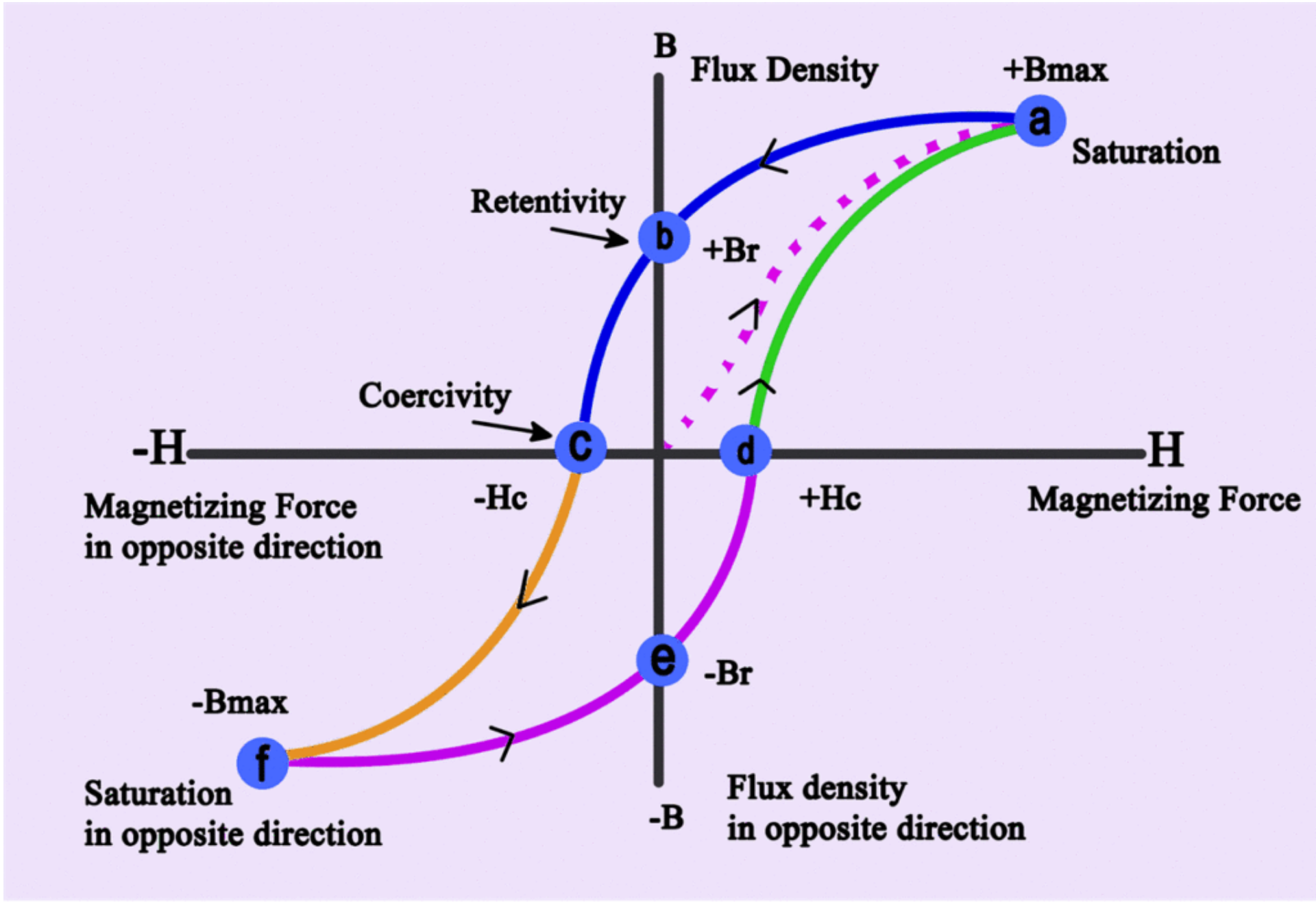
State Machine for Last-Time Prediction

1 bit FSM



Improving the Last Time Predictor

- Problem: A last-time predictor changes its prediction from T→NT or NT→T too quickly
  - even though the branch may be mostly taken or mostly not taken
- Solution Idea: Add hysteresis to the predictor so that prediction does not change on a single different outcome
  - Use two bits to track the history of predictions for a branch instead of a single bit
  - Can have 2 states for T or NT instead of 1 state for each
- Smith, "A Study of Branch Prediction Strategies," ISCA 1981.


B
Flux Density
+Bmax
a
Saturation
Retentivity
b
+Br
Coercivity
c
d
-H
H
-Hc
+Hc
Magnetizing Force
in opposite direction
Magnetizing Force
e
-Br
-Bmax
f
Saturation
in opposite direction
-B
Flux density
in opposite direction

Two-Bit Counter Based Prediction

- Each branch associated with a two-bit counter (2BC)
- One more bit provides hysteresis
- A strong prediction does not change with one single different outcome
- Also called binomial prediction
- Accuracy for a loop with N iterations = (N-1)/N

TNTNTNTNTNTNTNTNTNTNTN → 50% accuracy

צריך פעמיים N כדי לשנות את הדעה

(assuming counter initialized to weakly taken)

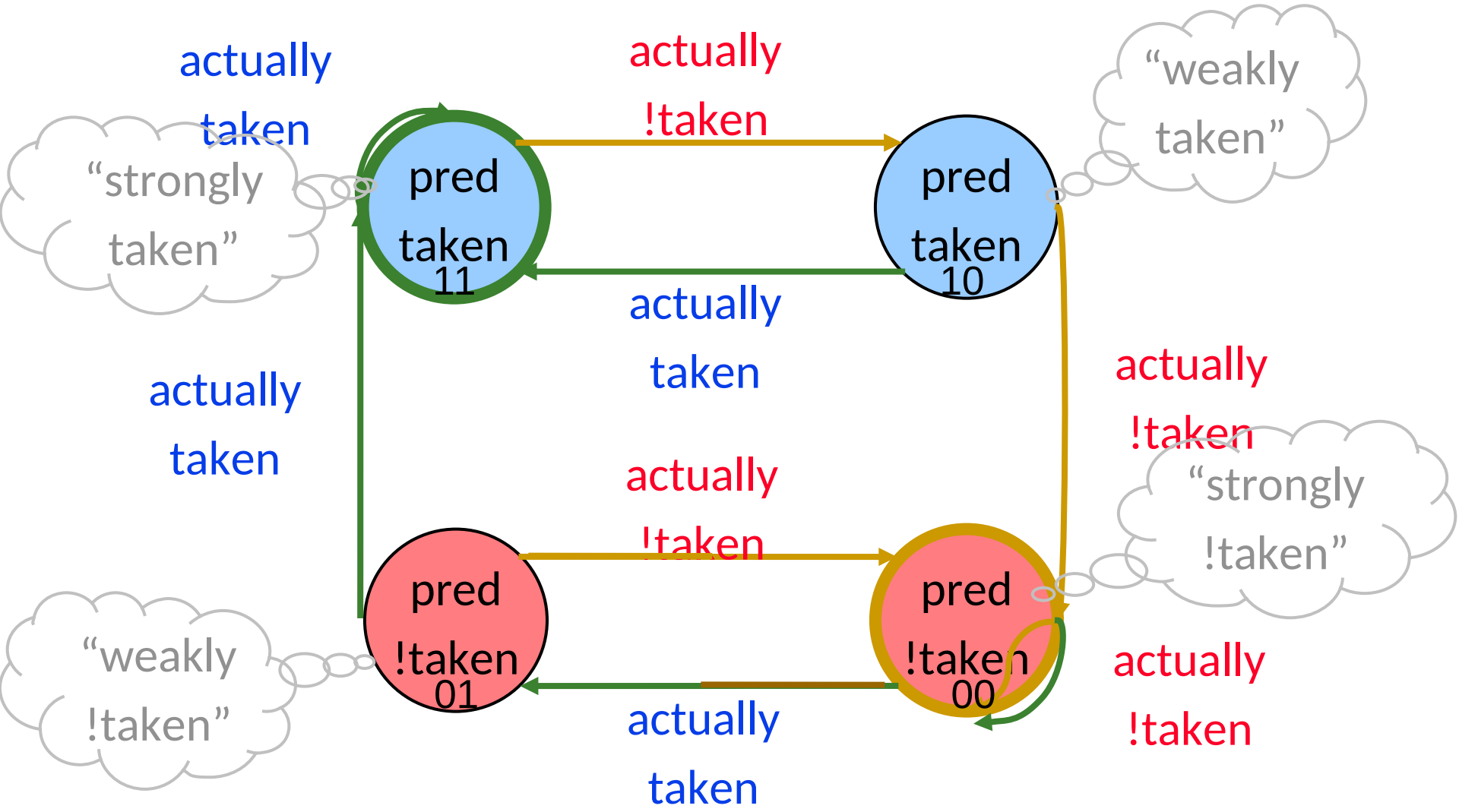
בדוגמה: 2 מתוך 20 --> 10%

+ Better prediction accuracy

2BC predictor CPI = [1 + (0.20*0.10) * 2] = 1.04 (90% accuracy)

-- More hardware cost (but counter can be part of a BTB entry)

Hysteresis Using a 2-bit Counter



Change prediction after 2 consecutive mistakes

Credit: H&P CAQA 3.9

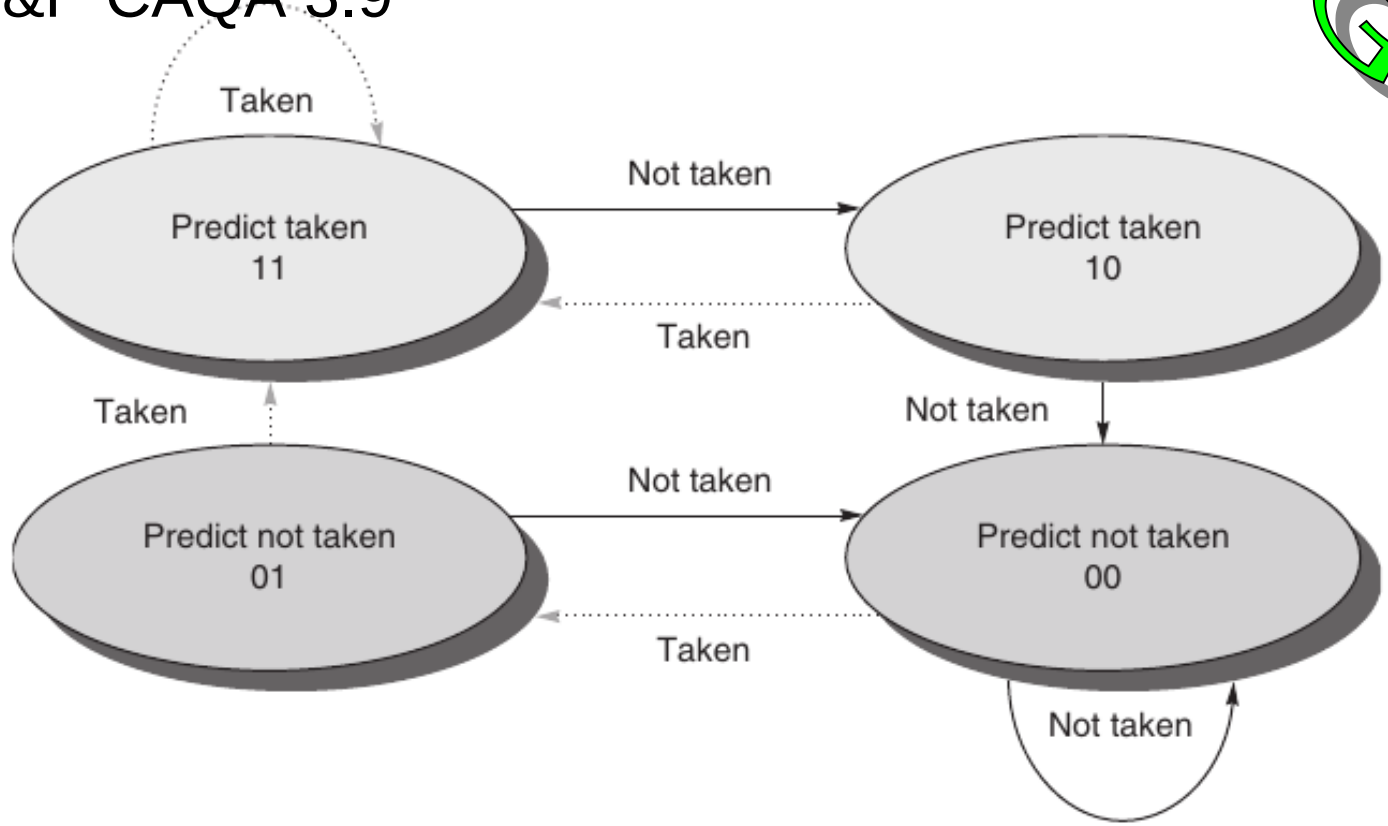
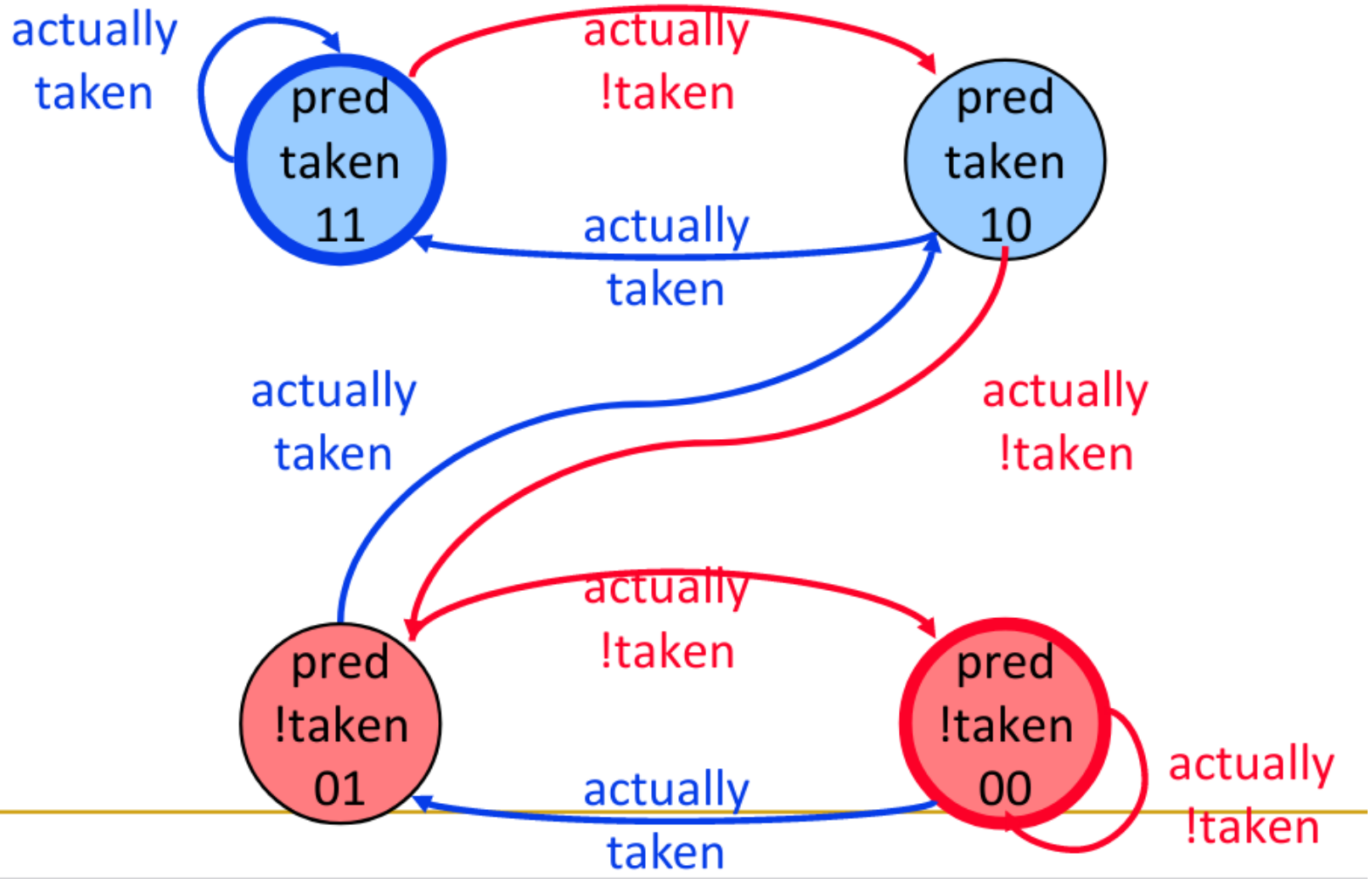


Figure C.18 The states in a 2-bit prediction scheme. By using 2 bits rather than 1, a branch that strongly favors taken or not taken—as many branches do—will be mispredicted less often than with a 1-bit predictor. The 2 bits are used to encode the four states in the system. The 2-bit scheme is actually a specialization of a more general scheme that has an *n*-bit saturating counter for each entry in the prediction buffer. With an *n*-bit counter, the counter can take on values between 0 and $2^n - 1$: When the counter is greater than or equal to one-half of its maximum value ($2^n - 1$), the branch is predicted as taken; otherwise, it is predicted as untaken. Studies of *n*-bit predictors have shown that the 2-bit predictors do almost as well, thus most systems rely on 2-bit branch predictors rather than the more general *n*-bit predictors.

State Machine for 2-bit Saturating Counter

- Counter using *saturating arithmetic*
  - Arithmetic with maximum and minimum values



Is This Good Enough?

- ~85-90% accuracy for **many** programs with 2-bit counter based prediction (also called bimodal prediction)
- Is this good enough?
- How big is the branch problem?

Rethinking the 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

מאמר רשות

Alternative implementations of two-level adaptive branch prediction

Authors: Tse-YuYeh, Yale N.Patt

https://dl.acm.org/doi/10.1145/146628.139709

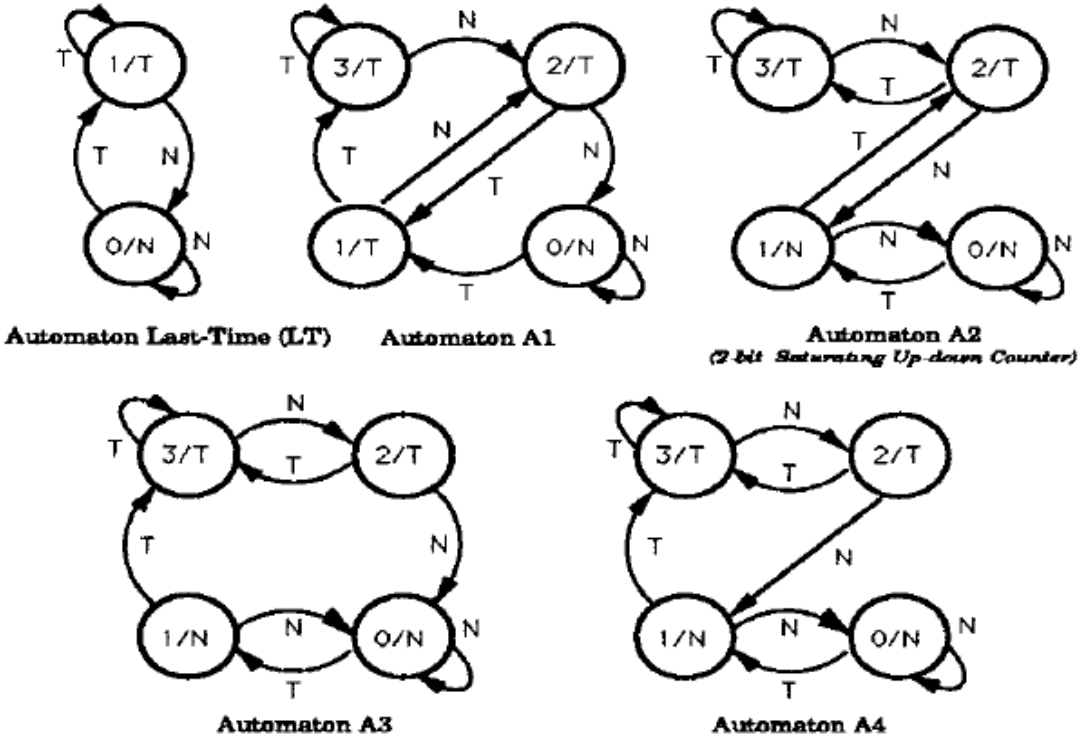


Figure 2: State diagrams of the finite-state Moore machines used for making prediction and updating the pattern history table entry.

מאמר רשות

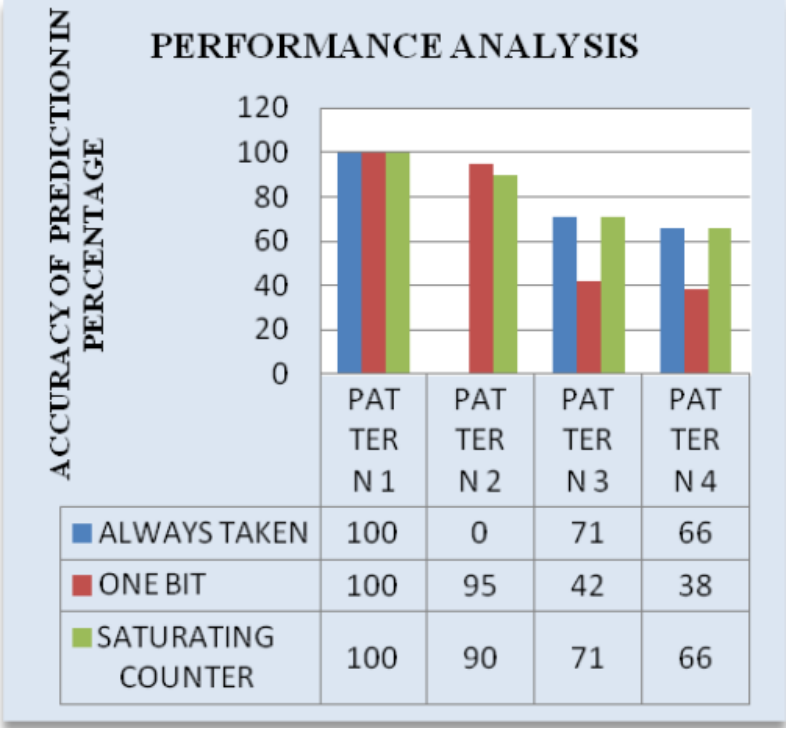
Dynamic Branch Prediction Modeller for RISC Architecture

Harsh Arora; Sagar Kotecha; Romil Samyal

2013 International Conference on Machine Intelligence and Research Advancement

https://ieeexplore.ieee.org/document/6918861

DOI: 10.1109/ICMIRA.2013.84



Dynamic Branch Prediction Modeller for RISC Architecture

Harsh Arora
School of Computing Science and Engineering,
Vellore Institute of Technology University, Vellore, India.
harsh_arora @ieee.org

Sagar Kotecha
MulticoreWare Inc.
Chennai,
Tamil Nadu, India.
sagarkotecha@gmail.com

Romil Samyal
IBM India Pvt Ltd.
Bangalore,
Karnataka, India.
rsamyal1@in.ibm.com

***Abstract*---In the past decade, by taking advantage of the RISC architecture, computer designers were able to benefits from the ILP and started using deeper pipelines, wider issue rates and superscalar techniques. However, due to the presence of branch instruction there is change in flow of instruction execution. In case of branch either we have to stall the pipeline until the branch instruction executes or we have to predict the branch output i.e. either branch is taken or not taken. Adding stalls in pipeline leads to the performance loss. Branch prediction is a widely referenced technique to overcome the Performance loss. It minimizes Performance loss by predicting the branch behaviour and issue subsequent instructions before the actual branch outcome is known. Whenever a prediction of branch is correct, penalty in pipeline is either reduced or completely avoided. Ironically, with shorter clock periods, the branch predictor has less time to make a prediction and might have to be scaled back to make it faster, which decreases accuracy and reduces the advantage of higher clock rates.**

We are going to analyze several algorithms for branch predictions in our first phase i.e. PHASE Φ. In order to support our analysis we are going to construct our own simulator for RISC architecture and putting-up some concrete results for these algorithms that can be referred for future researches.

the target address encoded in the branch instruction, indirect branches get their target address from a register, and returns get their target address from a link register or a stack. Technically, indirect branches and returns could also be conditional. The majority of the branches, 72% on average, are conditional, 17% are unconditional immediate, 10% are returns, and 1% are indirect[6]. This highlights the importance of conditional branch prediction.

The outcomes of branches that are close together in the source code are often dependent or related. This is because they are often based on the same or related variables. Knowing the outcome of the earlier branch therefore often helps us predict the outcome of the later branch which is also referred to as a branch correlation. One of the most common reasons for a branch to follow a repeating execution pattern is that it is the loop ending branch, or a branch inside the loop body. The pattern that is repeating can be any string of outcomes repeated over and over. However, the most common repeating pattern is a sequence of '*n*' taken outcomes followed by a single not-taken outcome, followed by taken outcomes and so on. A representative example of this type of pattern is **110 110 110**, where "1" represents a

Review: Can We Do Better? (than always taken or always not taken)

- Last-time and 2BC predictors exploit "last-time" predictability

תובנות

- Realization 1: A branch's outcome can be correlated with **other** branches' outcomes
  - **Global branch correlation – ניתן לקשור בין מופעים שונים של הסתעפויות**
- Realization 2: A branch's outcome can be correlated with **past** outcomes of the same branch (other than the outcome of the branch "last-time" it was executed)
  - **Local branch correlation – ניתן לקשור מצב נוכחי עם היסטורית ההסתעפות**

Yeh and Patt, "Two-Level Adaptive Training Branch Prediction," MICRO 1991.

Global Branch Correlation (I)

- Recently executed branch outcomes in the execution path is correlated with the outcome of the next branch

דוגמה 1

```
if (cond1)
...
if (cond1 AND cond2)
```

- If first branch not taken, second also not taken

דוגמה 2

```
if (x<1) . . .
if (x>1) . . .
```

או

```
branch Y: if (cond1) a = 2;
...
branch X: if (a == 0)
```

- If first branch taken, second definitely not taken

Global Branch Correlation (II)

```
branch Y: if (cond1)
...
branch Z: if (cond2)
...
branch X: if (cond1 AND cond2)
```

- If Y and Z both taken, then X also taken
- If Y or Z not taken, then X also not taken

Global Branch Correlation (III)

- Eqntott, SPEC 1992

```
if (aa==2)              ;; B1
        aa=0;
if (bb==2)              ;; B2
        bb=0;
if (aa!=bb) {   }       ;; B3
if (aa==bb) {   }       ;; B3
        ....
```

גיא: דוגמה זו לקוחה מ-
H&P Computer Architecture 3.3

הדוגמה שייכת לקוד EQNTOTT
מ-SPEC89

אם מסתכלים רק על אחד מהתנאים
B1 או B2 לא ניתן להסיק את B3.
לכן כדי לקבוע את המסקנה שרשומה
למטה צריך לדעת על ההסתעפויות
הן של B1 והן של B2.

EQNTOTT

Demo :הערה לעצמי

~/.../CPU/lectures/10/code/eqntott-tests

הדגם את test2

NULLSTONE

COMPANY
Home
Contacts
Customers
Testimonials

PRODUCTS
Overview
NULLSTONE for C
NULLSTONE for Java
Request Information

SUPPORT
Release Notes
Download
PGP Information
Service Report
Write Us

INFORMATION
Performance Results
Glossary of Terms

RELATED LINKS
Compiler Connection
Compiler Jobs

EQNTOTT-Specific Optimizations

Abstract

The EQNTOTT benchmark is a compute-intensive program that spends the majority of execution time in the function cmppt(). Some compilers use EQNTOTT-specific optimizations to achieve the best possible SPEC92 performance. Although EQNTOTT has not been included in the SPEC95 benchmark suite (in part due to problems discussed below), the issues relating to "benchmark-specific optimizations" are still the subject of debate.

Source Code

The examples discussed below have been codified into three test programs that exhibit the EQNTOTT-specific optimizations. The source code is available in eqntott-tests.tar.Z.

Overview

The EQNTOTT benchmark is a compute-intensive program that spends the majority of execution time in the function cmppt(). The function cmppt() has a loop that compares two

https://nullstone.com/eqntott/eqntott.htm

Global Branch Correlation (III) - המשך

Here is the MIPS code that we would typically generate for this code fragment assuming that aa and bb are assigned to registers R1 and R2:

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

If b1 is not taken (i.e., aa==0@b3) and b2 is not taken (i.e. bb=0@b3) then b3 is certainly taken

ניסיון לעשות תחזית רק ע"ס אחד מהפיצולים יהיה שגוי.

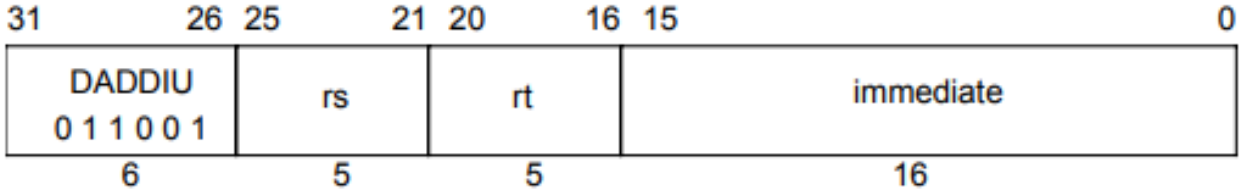
רק התחשבות בשני הפיצולים תיתן תחזית נכונה.

תחזית על סמך יותר מהסתעפות אחת נקראת Correlating.

Credit: H&P CAQA 5th Ed., section 3.3

DADDIU

Doubleword Add Immediate Unsigned



Format: DADDIU rt, rs, immediate **MIPS III**

Purpose: To add a constant to a 64-bit integer.

Description: rt ← rs + immediate

The 16-bit signed *immediate* is added to the 64-bit value in GPR *rs* and the 64-bit arithmetic result is placed into GPR *rt*.

No Integer Overflow exception occurs under any circumstances.

Restrictions:

None

Operation: **64-bit processors**

```
GPR[rt] ← GPR[rs] + sign_extend(immediate)
```

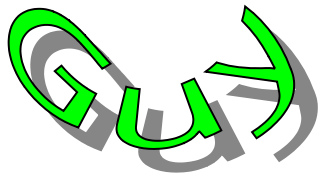
Exceptions:

Reserved Instruction

Programming Notes:

The term "unsigned" in the instruction name is a misnomer; this operation is 64-bit modulo arithmetic that does not trap on overflow. It is appropriate for arithmetic which is not signed, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as "C" language arithmetic.

SPEC



SPEC = Standard Performance Evaluation Corporation

https://www.spec.org/

SPECint and SPECfp

SPECrate®2017 Floating Point	SPECspeed®2017 Floating Point	Language [1]	KLOC [2]	Application Area
503.bwaves_r	603.bwaves_s	Fortran	1	Explosion modeling
507.cactuBSSN_r	607.cactuBSSN_s	C++, C, Fortran	257	Physics: relativity
508.namd_r		C++	8	Molecular dynamics
510.parest_r		C++	427	Biomedical imaging: optical tomography with finite elements
511.povray_r		C++, C	170	Ray tracing
519.lbm_r	619.lbm_s	C	1	Fluid dynamics
521.wrf_r	621.wrf_s	Fortran, C	991	Weather forecasting
526.blender_r		C++, C	1,577	3D rendering and animation
527.cam4_r	627.cam4_s	Fortran, C	407	Atmosphere modeling
	628.pop2_s	Fortran, C	338	Wide-scale ocean modeling (climate level)
538.imagick_r	638.imagick_s	C	259	Image manipulation
544.nab_r	644.nab_s	C	24	Molecular dynamics
549.fotonik3d_r	649.fotonik3d_s	Fortran	14	Computational Electromagnetics
554.roms_r	654.roms_s	Fortran	210	Regional ocean modeling

Capturing Global Branch Correlation

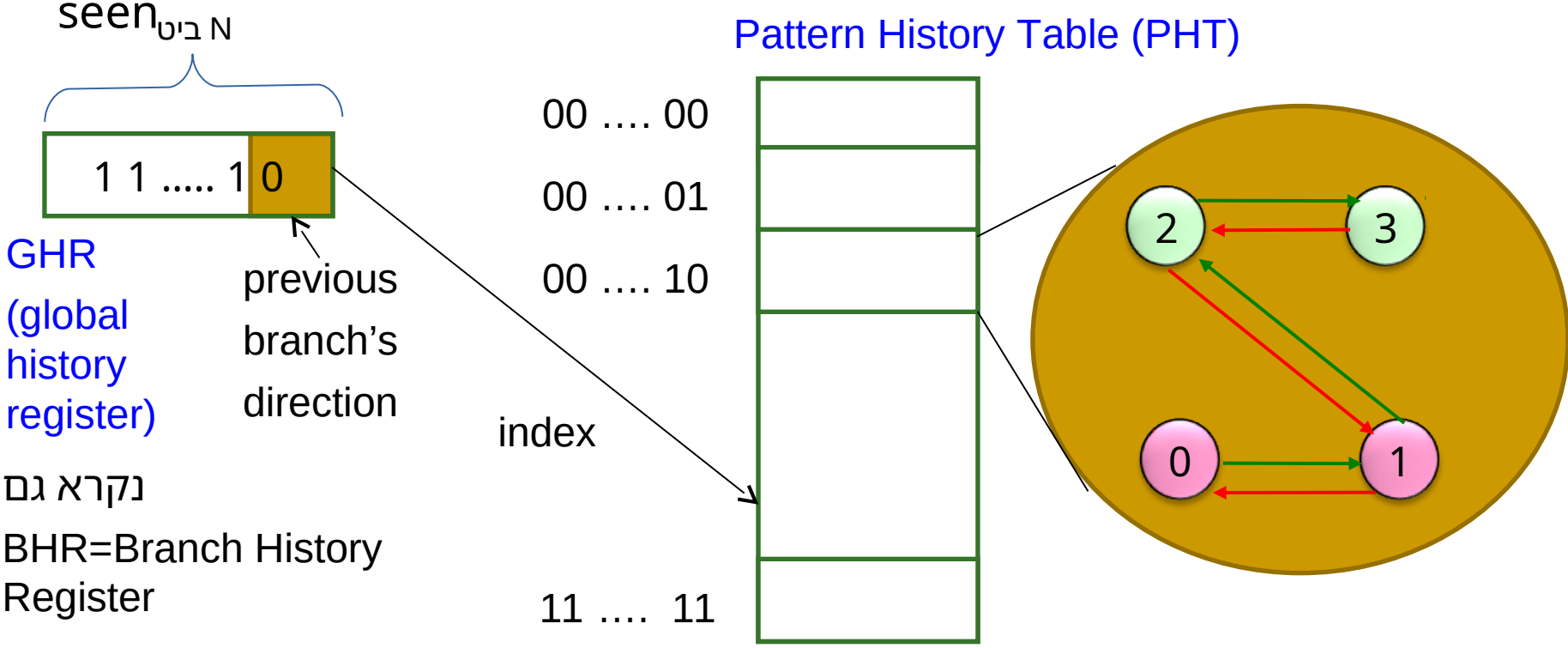
- Idea: Associate branch outcomes with "global T/NT history" of all branches
- Make a prediction based on the outcome of the branch the last time the same global branch history was encountered
- Implementation:
  - Keep track of the "global T/NT history" of all branches in a register → **Global History Register (GHR)**
  - Use GHR to index into a table that recorded the outcome that was seen for each GHR value in the recent past → **Pattern History Table** (table of 2-bit counters)
- Global history/branch predictor
- Uses two levels of history (GHR + history at that GHR)

Yeh and Patt, "Two-Level Adaptive Training Branch Prediction," MICRO 1991.

Two Level Global Branch Prediction

תזכורת: אנחנו עוסקים עתה רק בהמלצה לגבי לקיחת ההסתעפות. לא בכתובת. לשם כך ישנו ה- BTB

- First level: Global branch history register (N bits)
  - The direction of last N branches
- Second level: Table of saturating counters for each history entry
  - The direction the branch took the last time the same history was seen



Yeh and Patt, "Two-Level Adaptive Training Branch Prediction," MICRO 1991.

מתוך המאמר המקורי Yeh & Patt 1991

Shift left when update

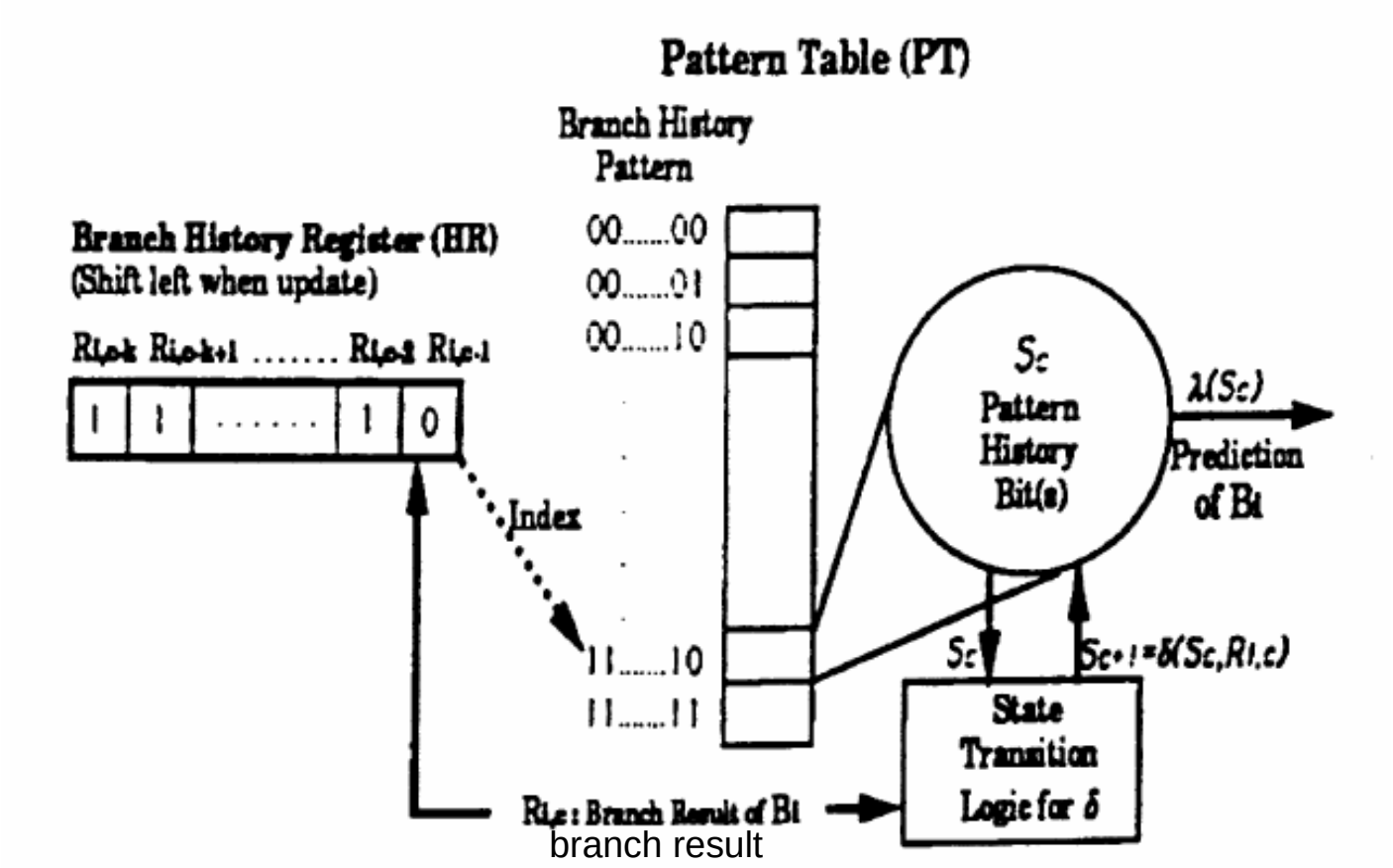


Figure 1: The structure of the Two-Level Adaptive Training scheme.

How Does the **Global** Predictor Work?

```
for (i=0; i<100; i++)
    for (j=0; j<3; j++)
```

After the initial startup time, the conditional branches have the following behavior, assuming GR is shifted to the left:

נכנסנו ללולאת J כאשר GR הראה 1110

test	value	GR	result
j<3	j=1	1101	taken
j<3	j=2	1011	taken
j<3	j=3	0111	not taken
i<100		1110	usually taken

GR = Global Register

This branch tests I.

Last 3 branches test j

History: TTTN

Predict taken for i

Next history: TTNT

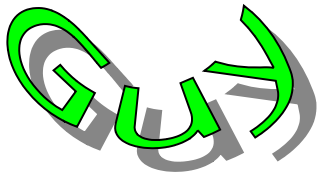
(shift in last outcome)

אחרי 1110 נקבל שוב 1101 והתבנית תתייצב ותחזור על עצמה

פה יש קצת עיוות, ה- GR שבמאמר הוא פר PC ולא ממש GR אמיתי. MCFARLING לא מתחשב בשינוי של ה-GR כתוצאה מהבדיקה של i!!! אחרת היינו צפויים לראות מצב 1111 שלא נראה בשקף או במאמר

- McFarling, "Combining Branch Predictors," DEC WRL TR 1993.

McFarling, "Combining Branch Predictors," DEC WRL TR 1993



test	value	GR	result
j<3	j=1	1101	taken
j<3	j=2	1011	taken
j<3	j=3	0111	not taken
i<100		1110	usually taken

McFarling, "Combining Branch Predictors," DEC WRL TR 1993

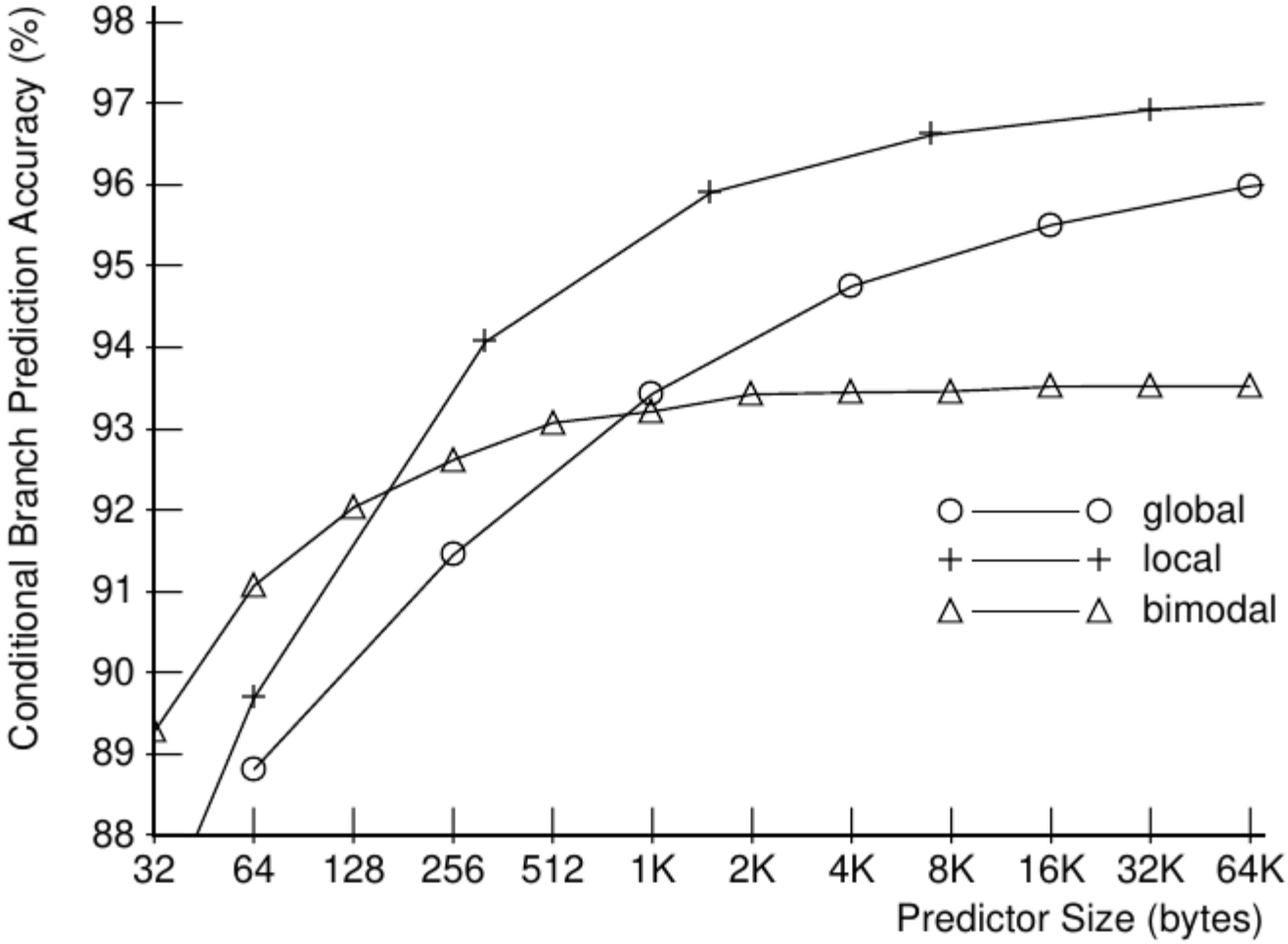


Figure 7: Global History Predictor Performance

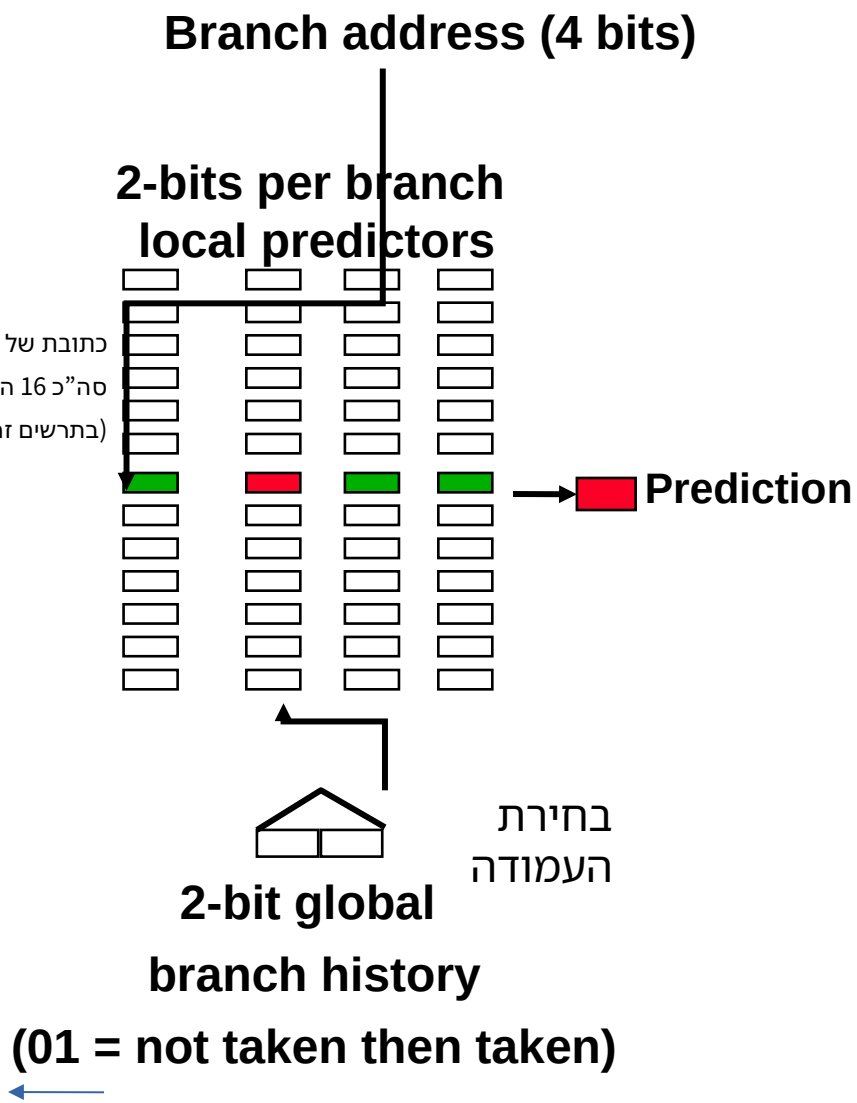
Correlating Branches – אינדוקס דו-ממדי

Idea: taken/not taken of recently executed branches is related to behavior of next branch (as well as the history of that branch behavior)

- Then behavior of recent branches selects between, say, 4 predictions of next branch, updating just that prediction

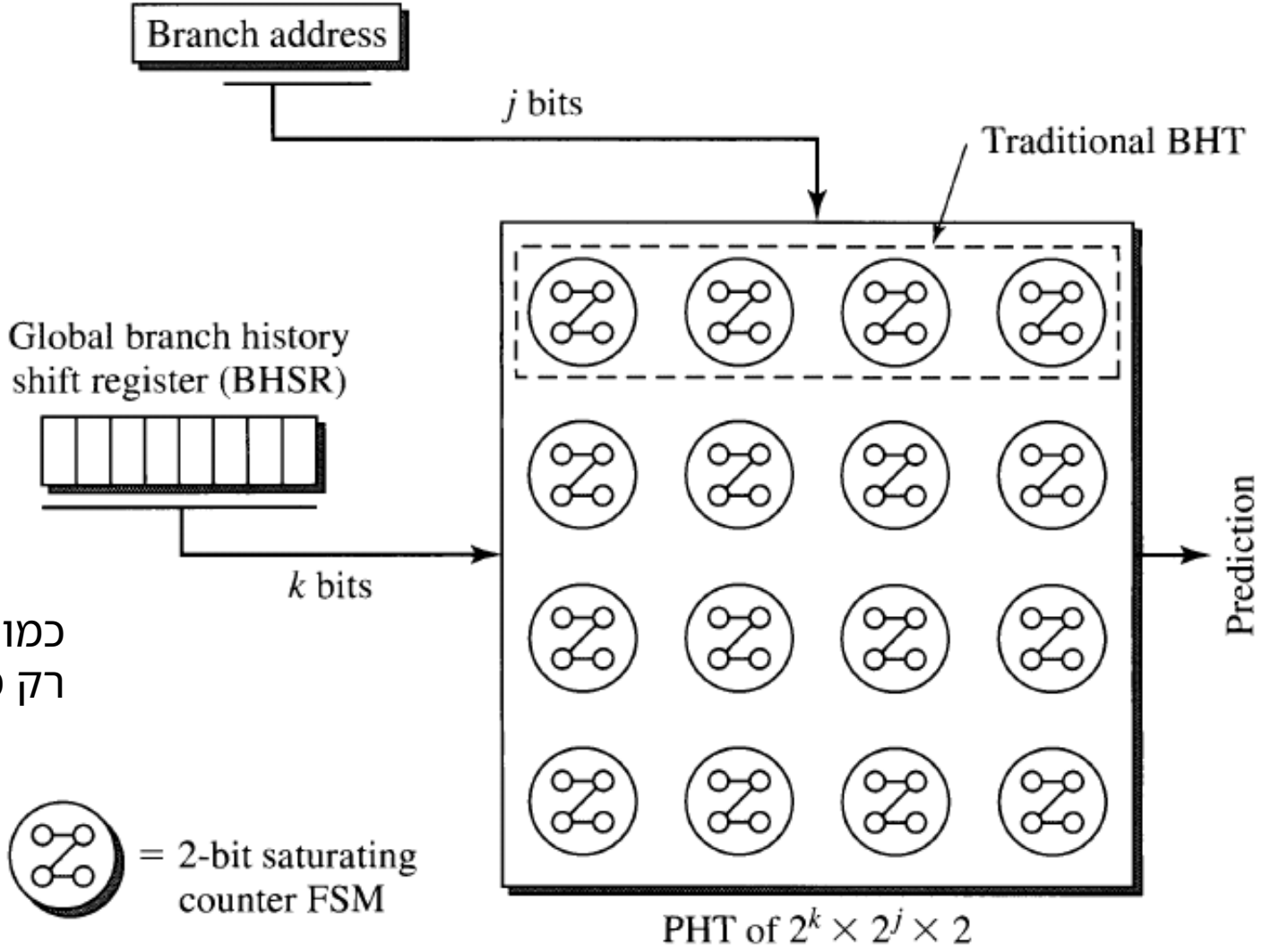
- (2,2) predictor: 2-bit global, 2-bit local

כתובת של הסתעפות מסוימת ע"ס 4 ביטים בלבד.
סה"כ 16 הסתעפויות.
(בתרשים זה לא נראות 16 שורות)



4 ביט של כתובת ההסתעפות -> 16 שורות
2 ביט של היסטוריה גלובלית -> 4 עמודות
בתוך החיתוך בין השורה לעמודה יש את התחזית

Following Yeh & Patt



כמו התרשים הקודם
רק מסובב

credit:

“Modern Processor Design” by

Shen & Lipasti

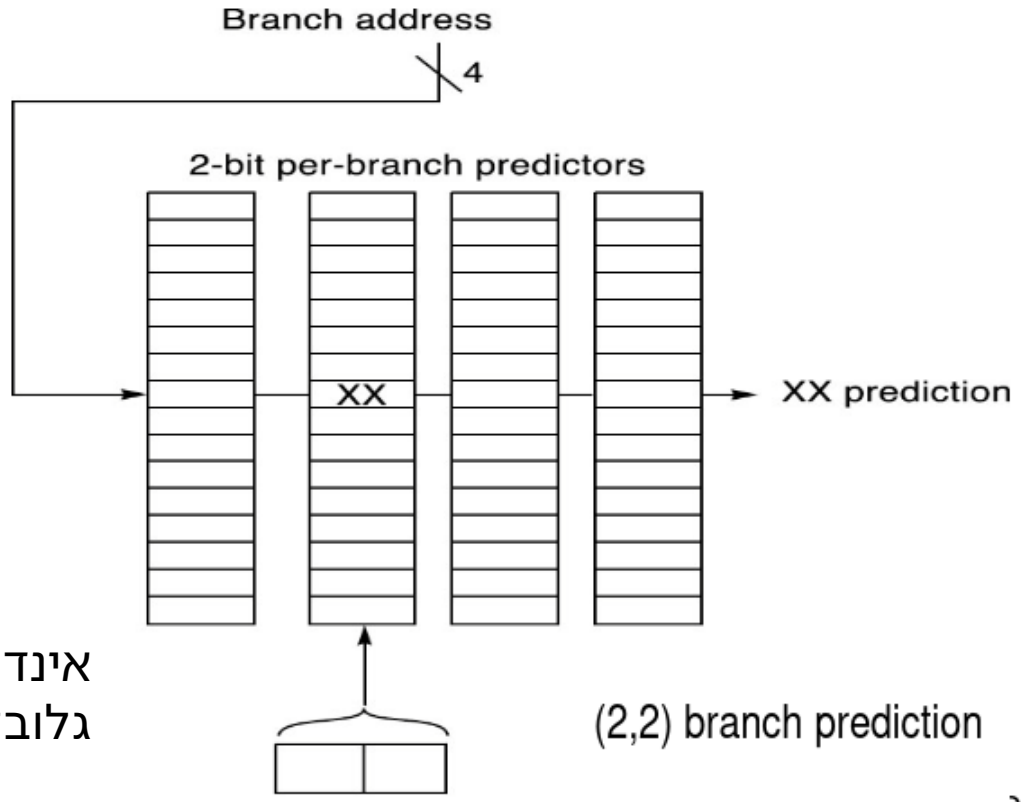
Figure 5.12

Correlated Branch Predictor with Global BHSR and Shared PHTs (GAs).

Correlating Branch Predictor

- If we use 2 branches as histories, then there are 4 possibilities (T-T, NT-T, NT-NT, NT-T).
- For each possibility, we need to use a predictor (1-bit, 2-bit).
- And this repeats for every branch.

```
if (aa==2)          T
    aa = 0
if (bb==2)          T
    bb = 0
if(aa!=bb) { ...    NT
```



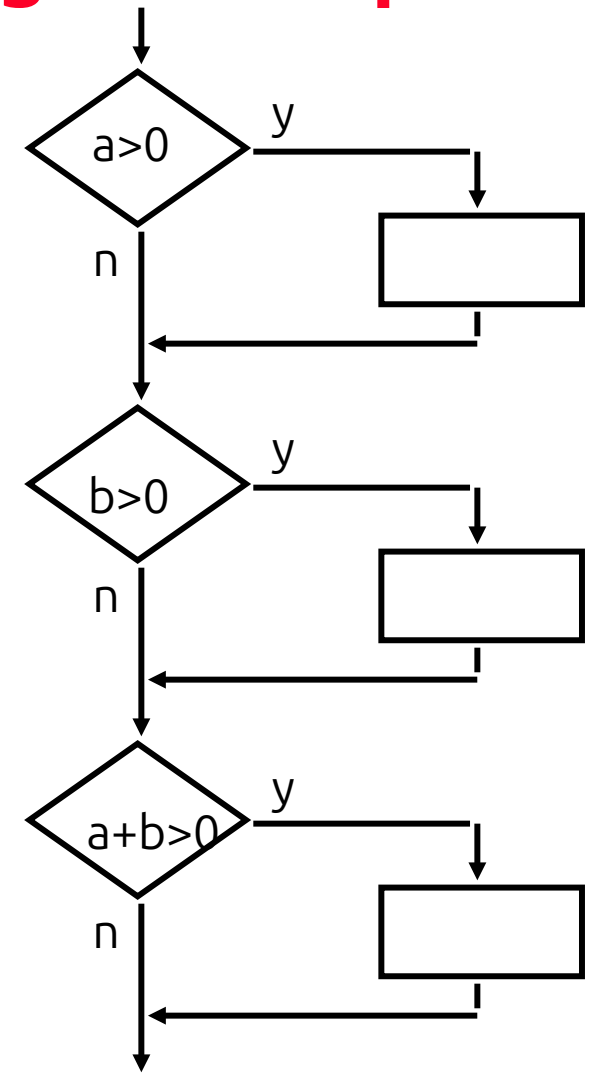
אינדוקס X-X: כתובת – היסטוריה גלובלית

Correlating branch prediction

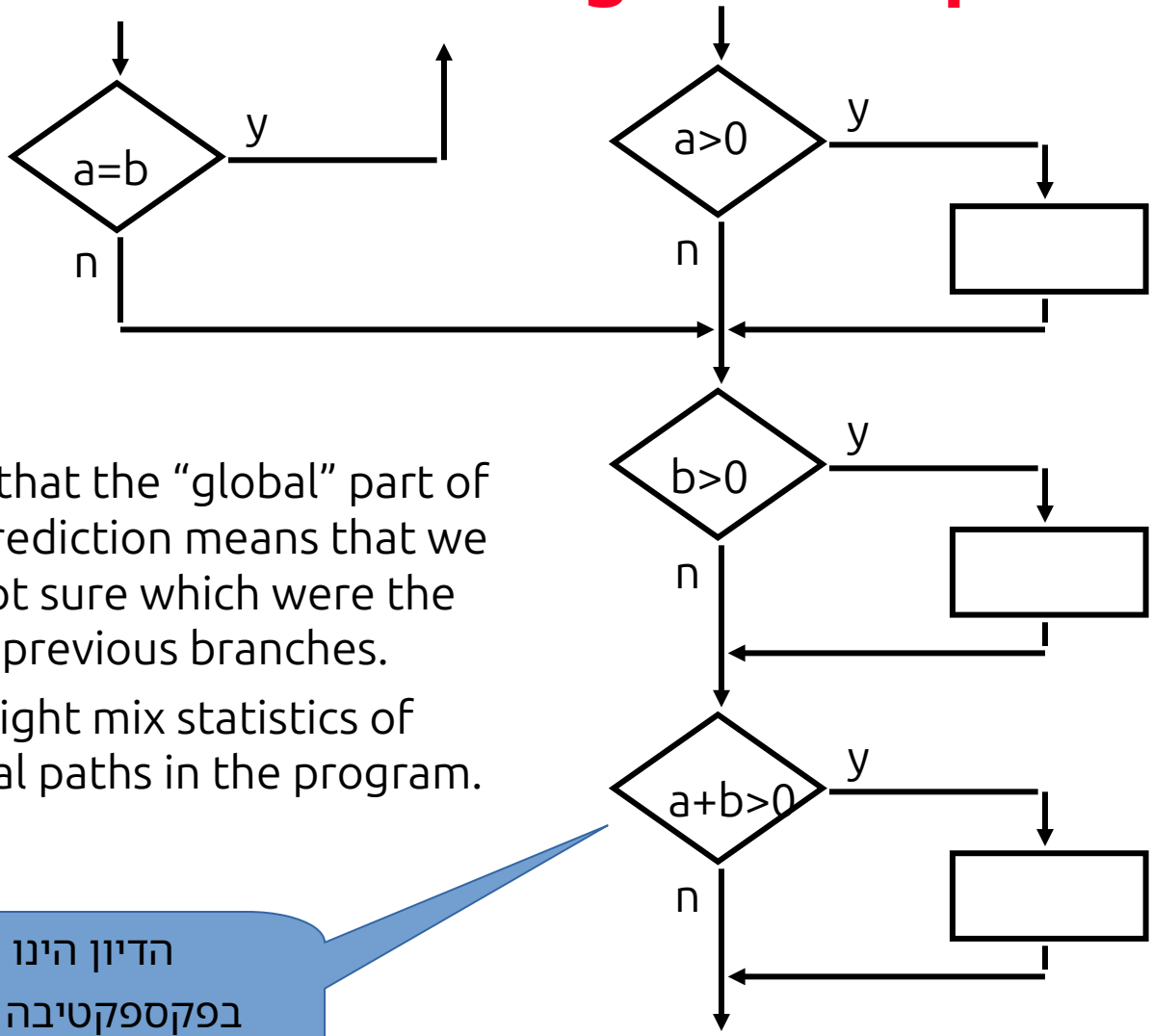
Say that a and b are random variables uniformly distributed from -1 to +1.

It is clear that

The statistics of the 3rd branch depends on the 1st and 2nd branches



Correlating branch prediction



Note that the “global” part of the prediction means that we are not sure which were the last 2 previous branches.

We might mix statistics of several paths in the program.

הדיון הינו
בפקספקטיבה של
ההסתעפות השלישית

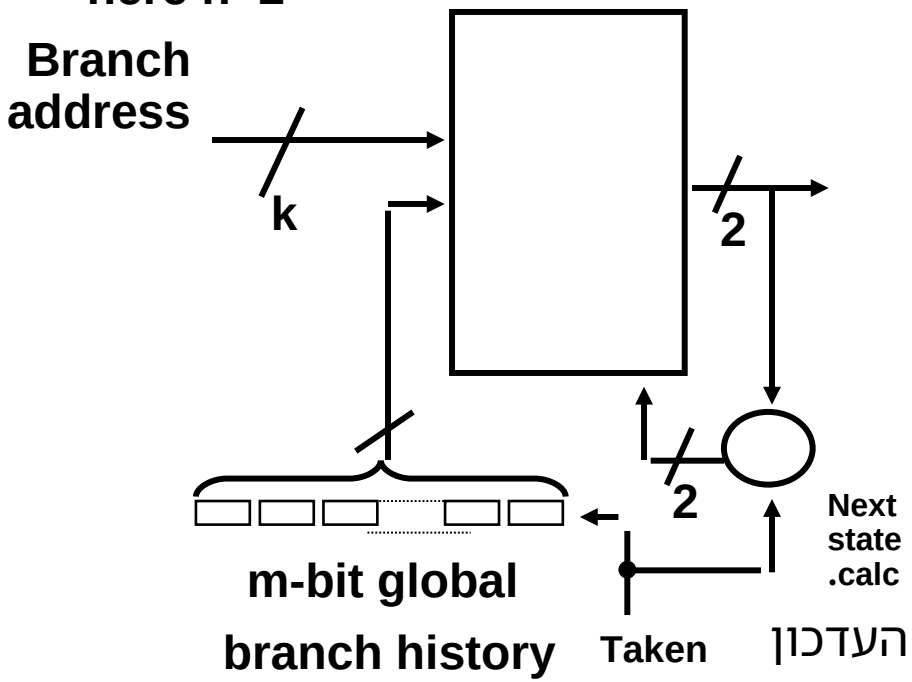
Correlating Branches (cont.)

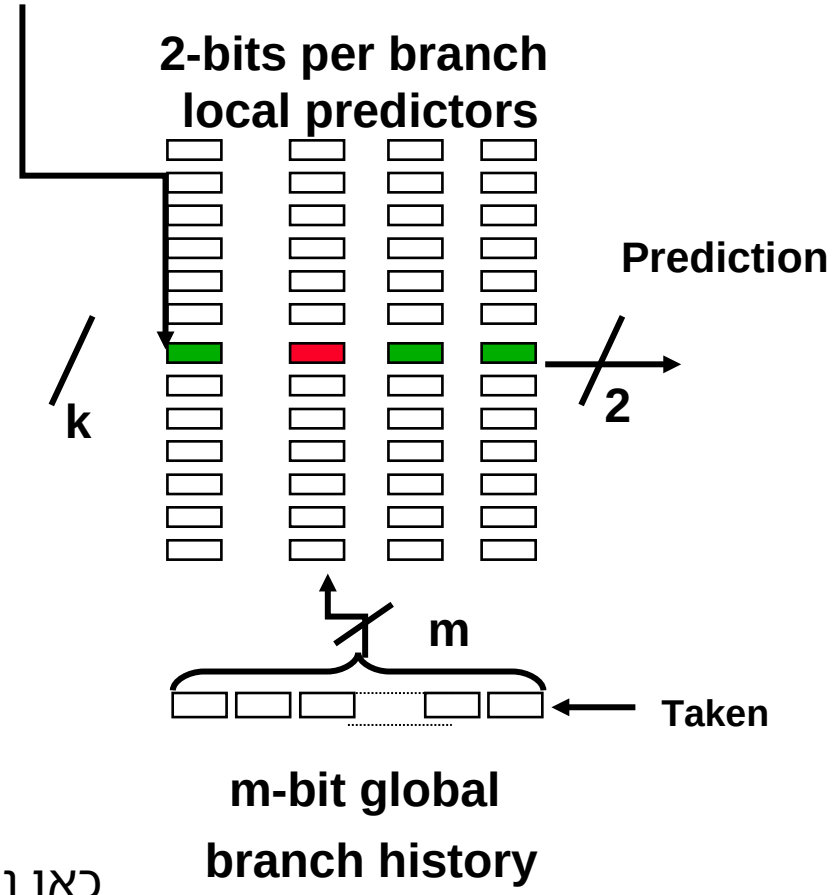
אנימציה

(m,n) predictor:
m bit global,
n bit local

Size=2^k x 2^m x n

here n=2





כאן נראה המקרה הכללי. לפני כן ראינו דוגמאות עבור m=2

לקריאה נוספת

https://www.geeksforgeeks.org/correlating-branch-prediction/

Intel Pentium Pro Branch Predictor

- Two level global branch predictor
- 4-bit global history register
- בדיוק באותו גודל שראינו בדוגמה קודם במאמר של McFarling
- Multiple pattern history tables (of 2 bit counters)
- Which pattern history table to use is determined by lower order bits of the branch address
- First widely commercially successful out-of-order execution machine

PENTIUM® PRO PROCESSOR AT 150, 166, 180, and 200 MHz

intel®

1.0. INTRODUCTION

The Pentium Pro processor is the next in the Intel386™, Intel486™, and Pentium family of processors. The Pentium Pro processor implements a **Dynamic Execution** microarchitecture—a unique combination of multiple branch prediction, data flow analysis, and speculative execution.

symbol implies that the signal is inverted. For example, D[3:0] = 'HLHL' refers to a hex 'A', and D#[3:0] = 'LHLH' also refers to a hex 'A'. (H= High logic level, L= Low logic level)

The word *Preliminary* appears o
with your local Field Applicat
recent information.

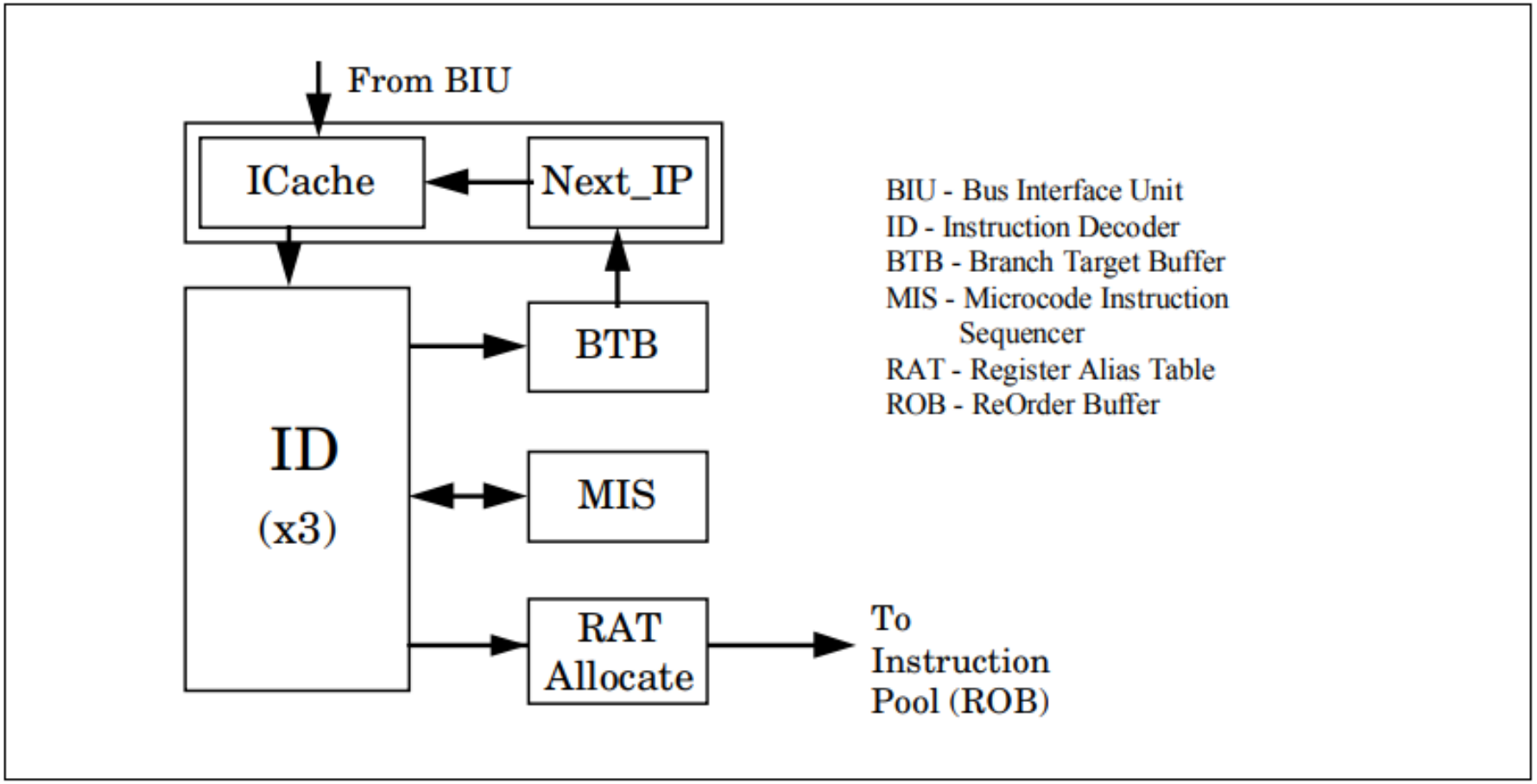
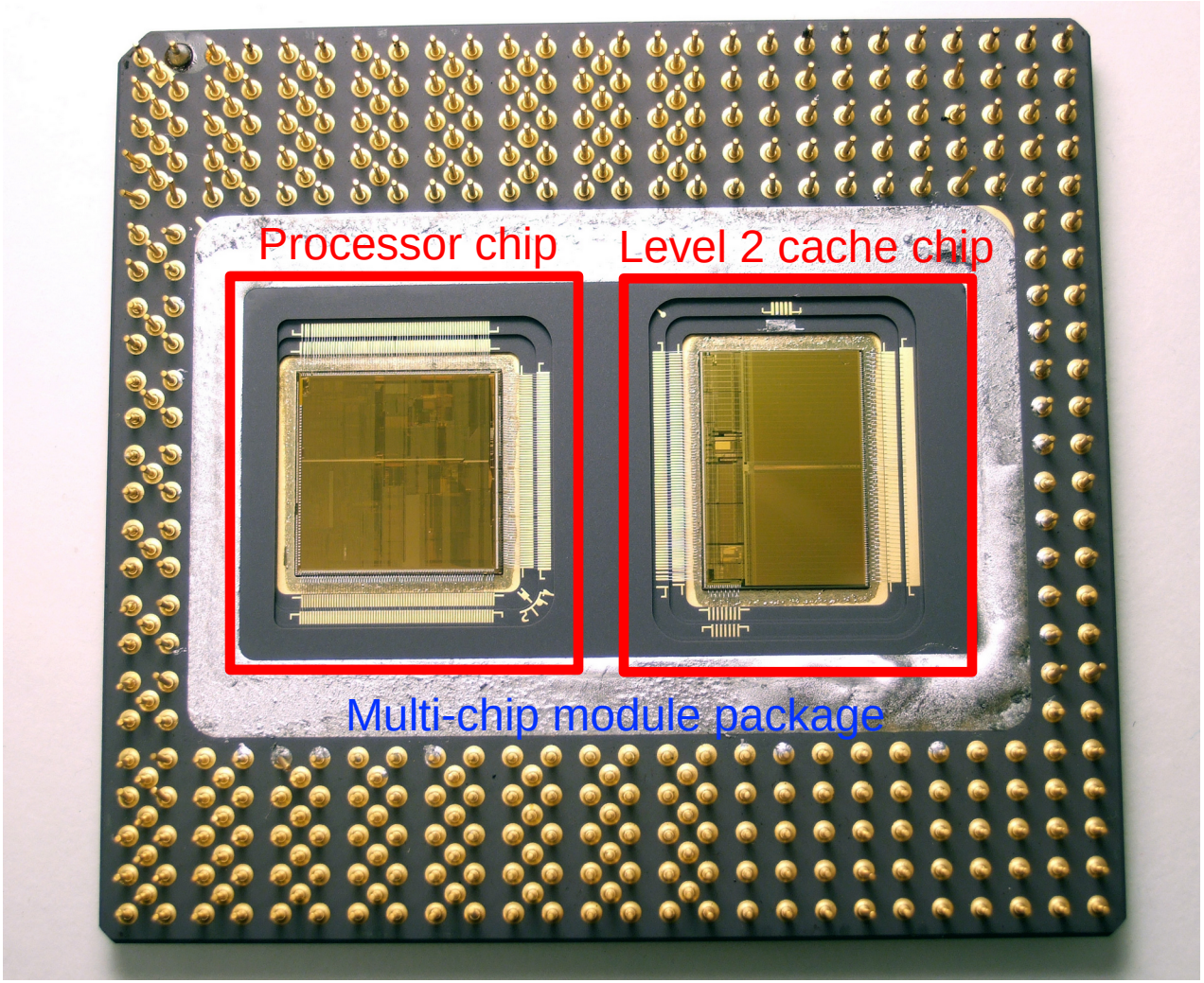


Figure 3. Inside the Fetch/Decode Unit

Intel Pentium Pro (1995)



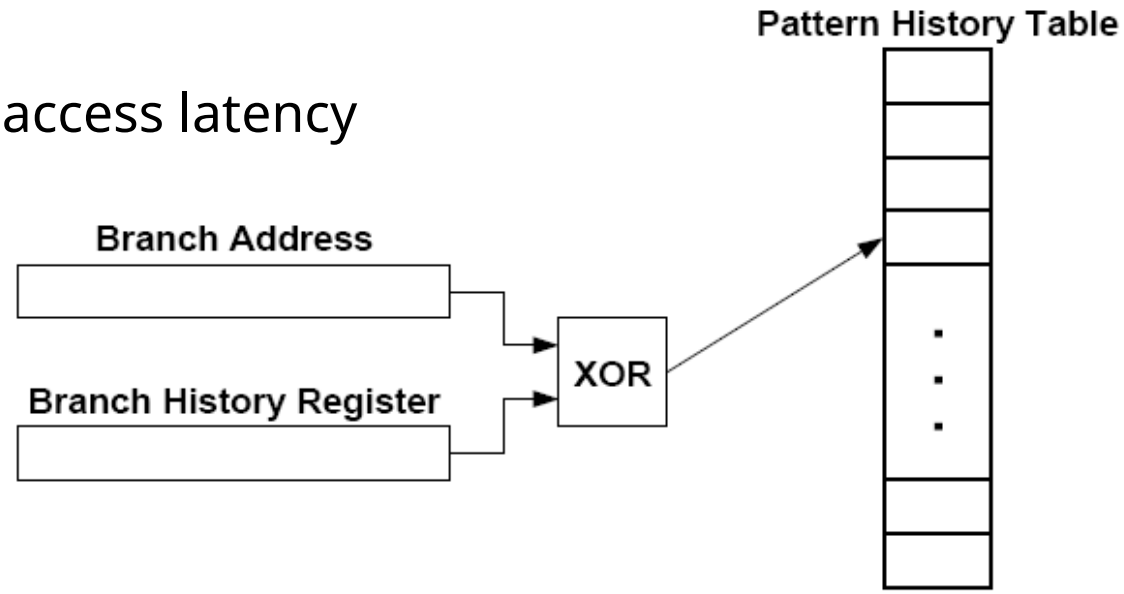
Improving Global Predictor Accuracy

- Idea: Add more context information to the global predictor to take into account which branch is being predicted
  - Gshare predictor: GHR hashed with the Branch PC

+ More context information used for prediction

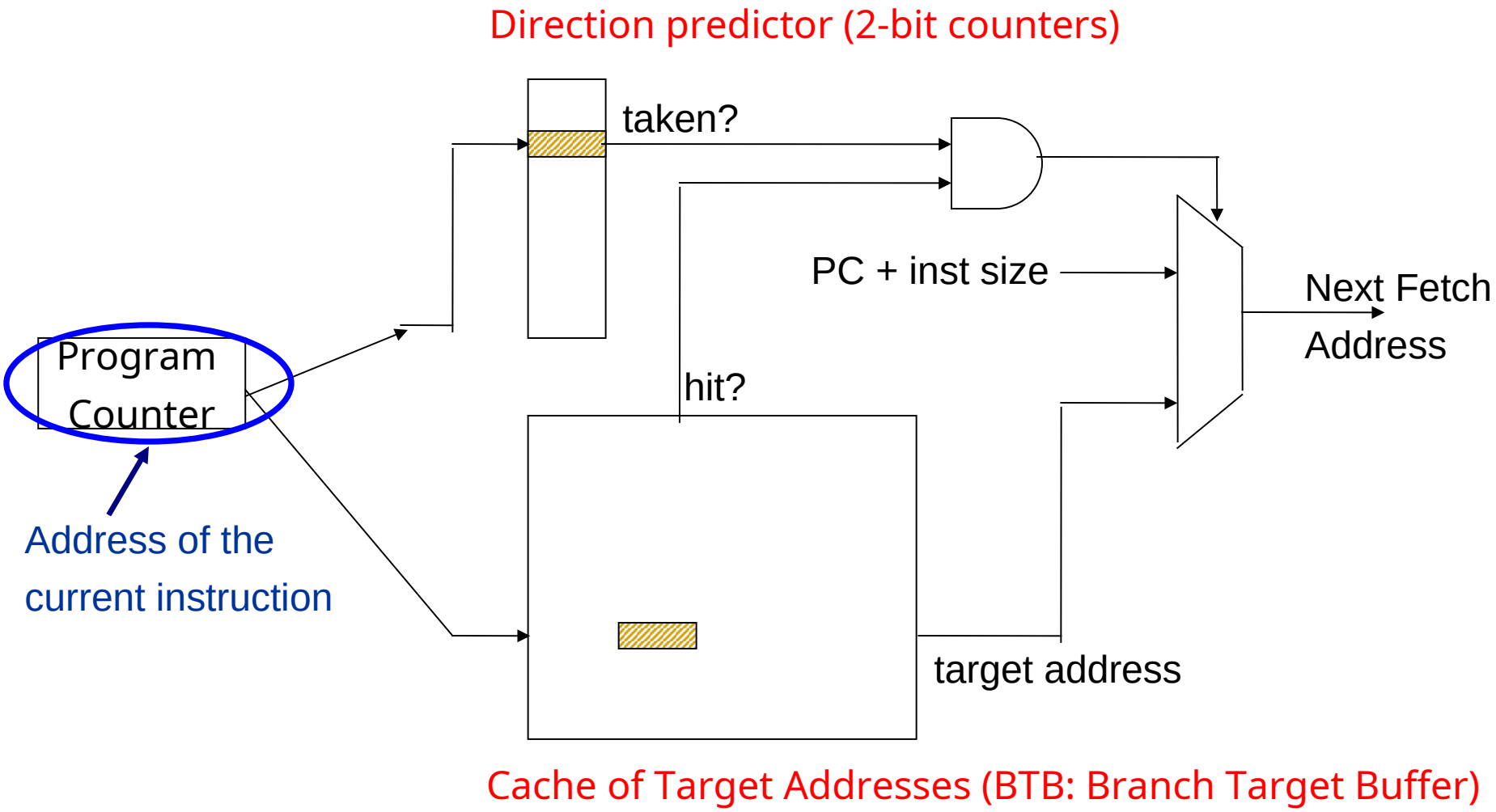
+ Better utilization of the two-bit counter array (PHT = Pattern History Table).

-- Increases access latency

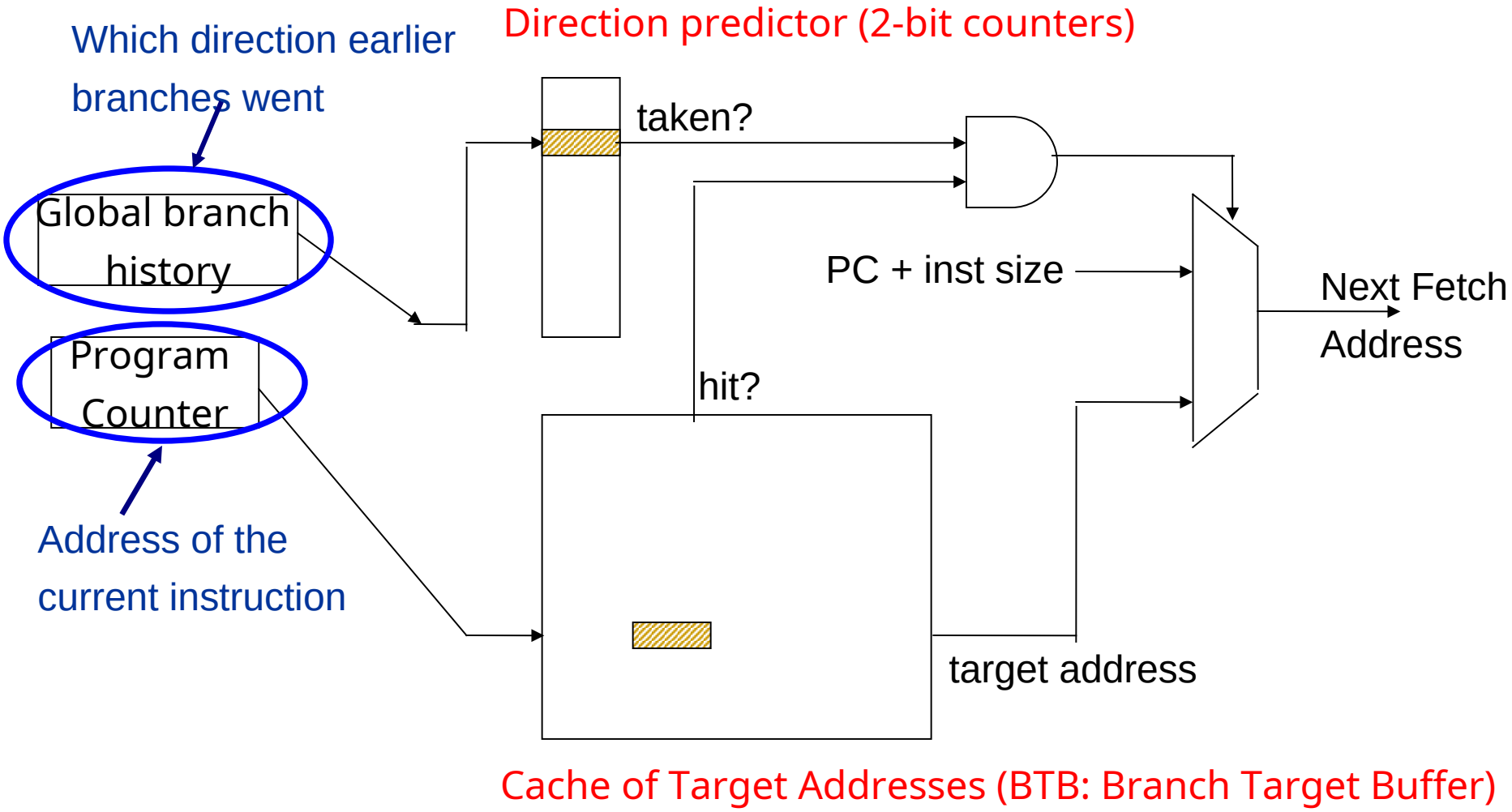


- McFarling, "Combining Branch Predictors," DEC WRL Tech Report, 1993.

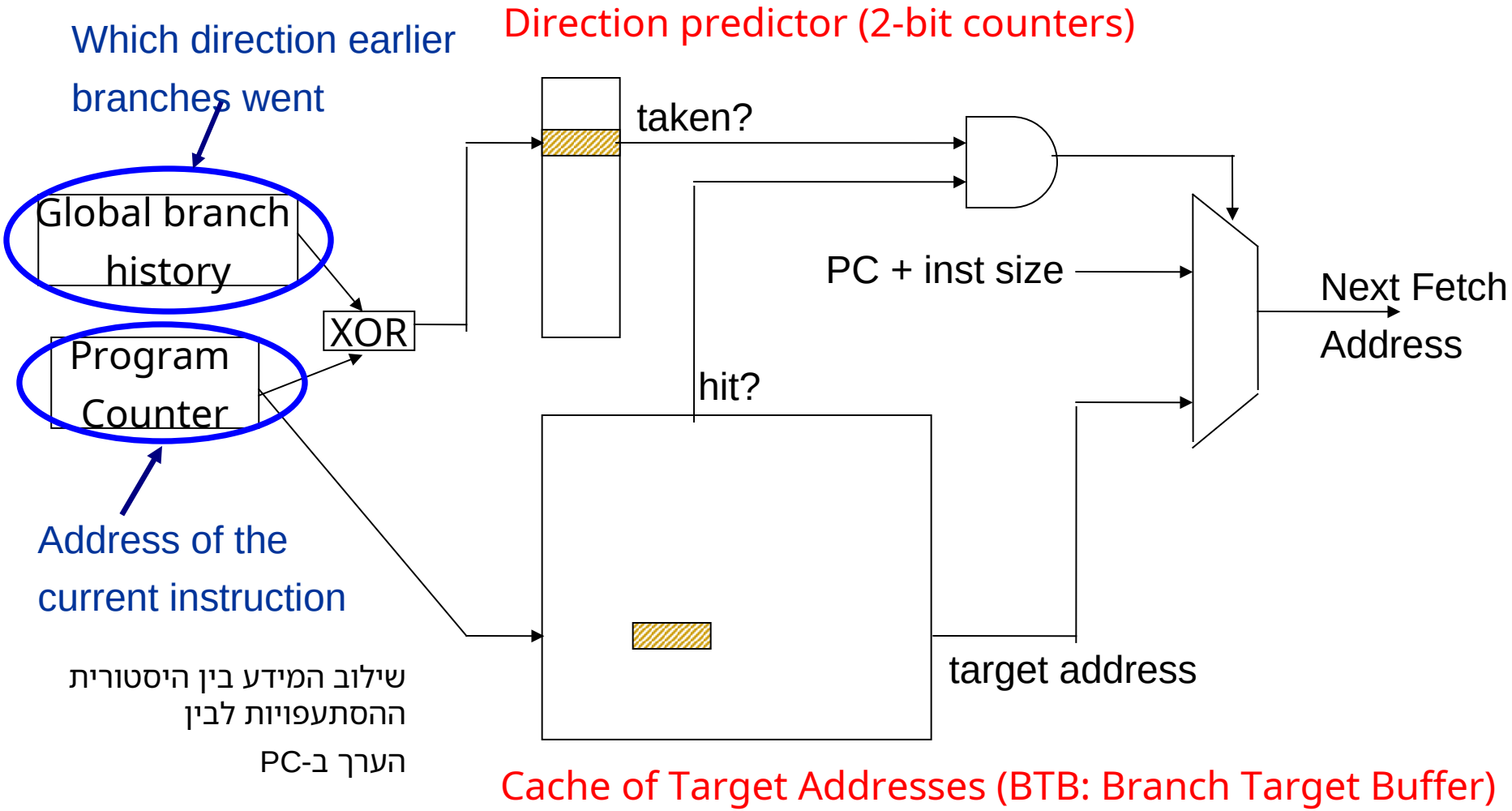
Review: One-Level Branch Predictor



Two-Level Global History Branch Predictor



Two-Level **Gshare** Branch Predictor



Combining Branch Predictors
Scott McFarling

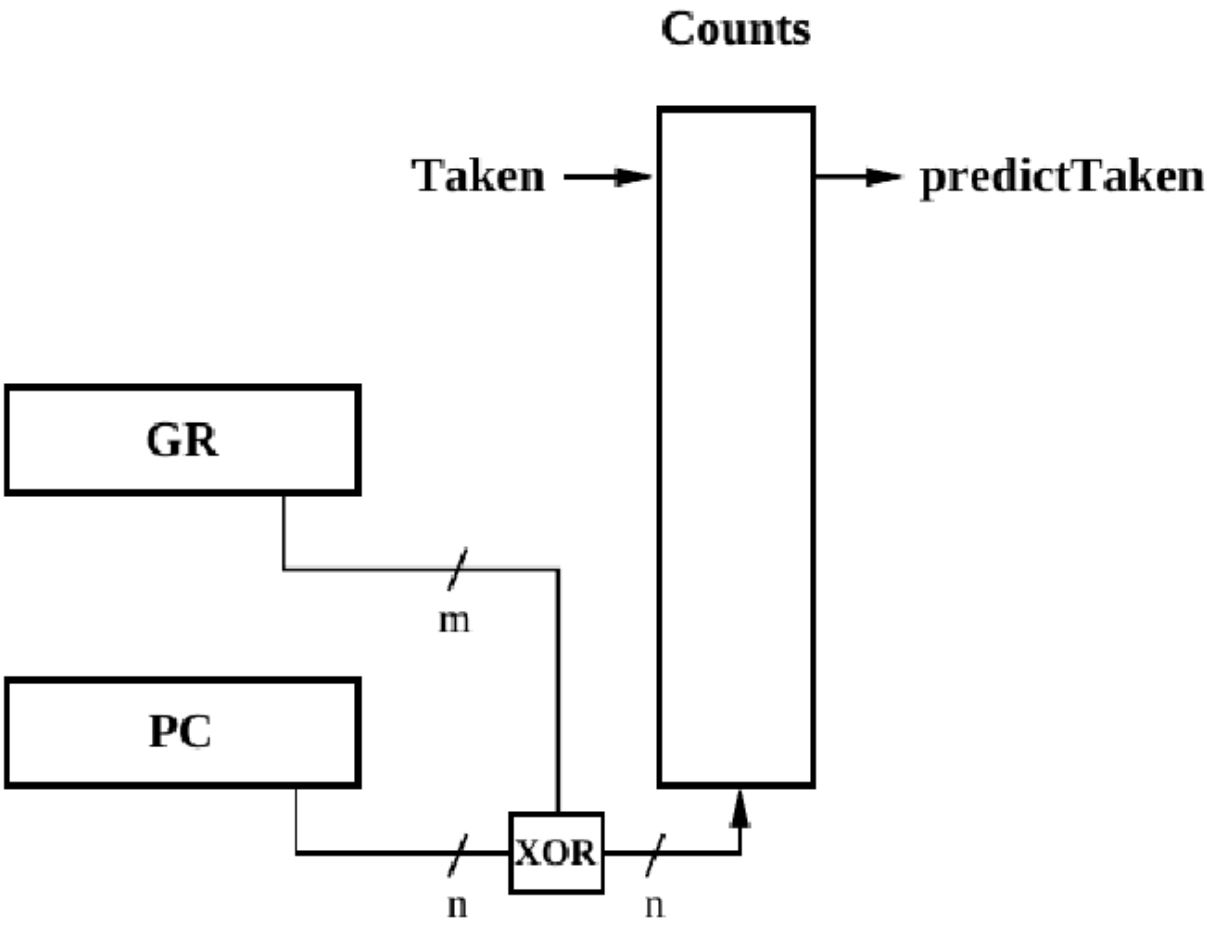
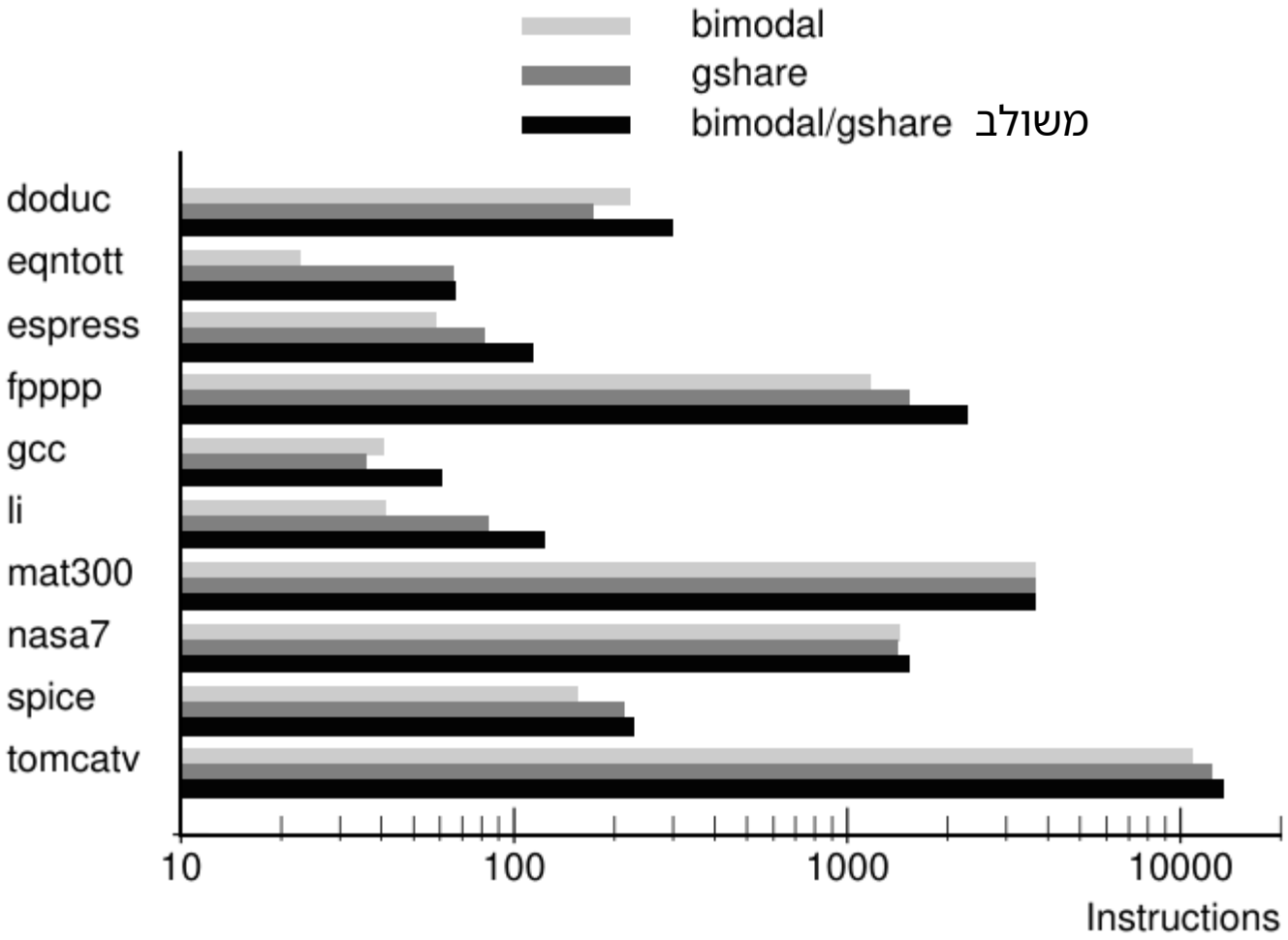


Figure 10: Global History Predictor with Index Sharing

Scott McFarling, “Combining Branch Predictors”



מספר ההוראות בין שתי החטאות בחיזוי

Figure 15: Instructions between Misspredicted Branches

Scott McFarling, “Combining Branch Predictors”

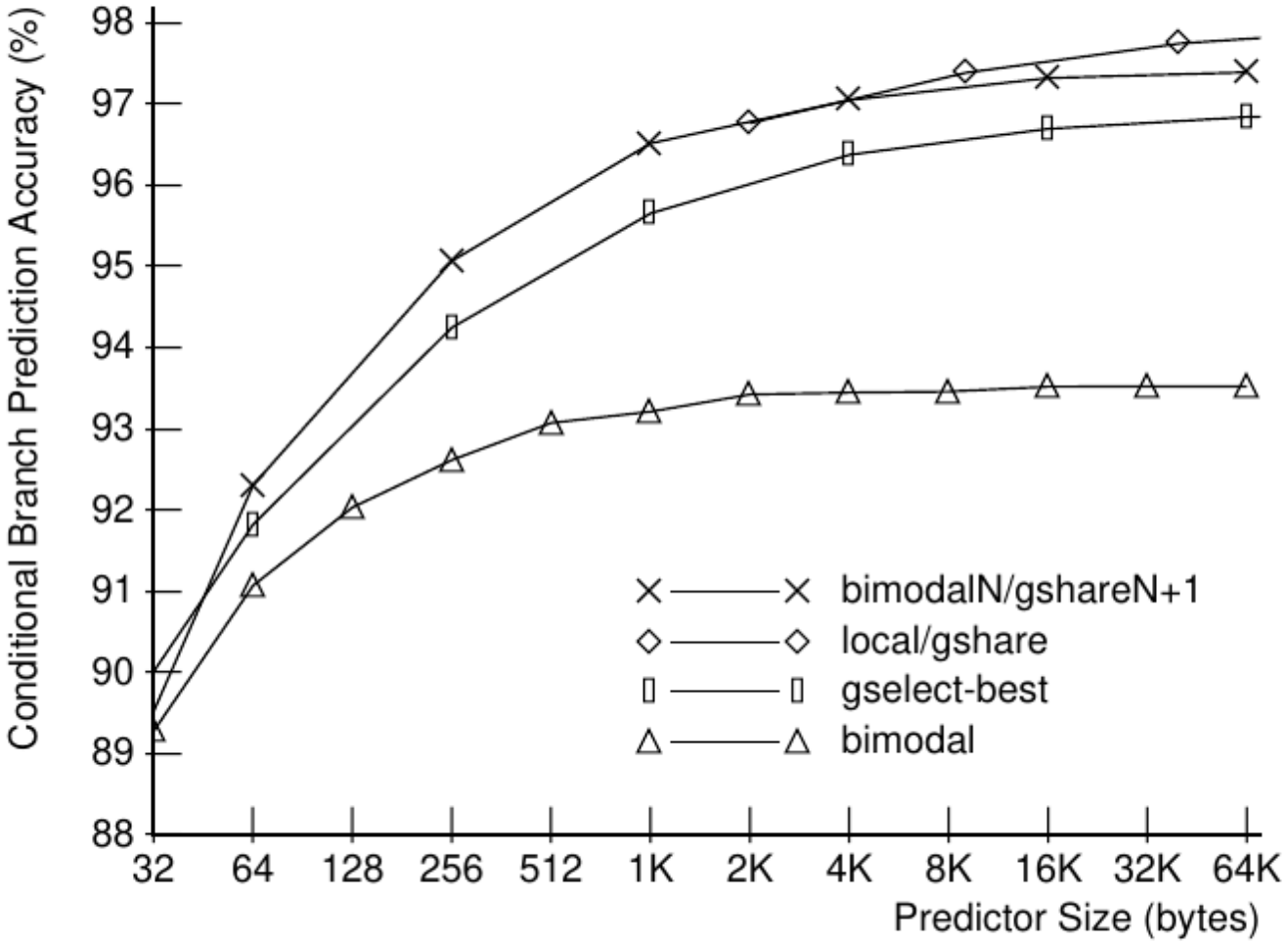


Figure 16: Combined Predictor Performance by Size

Can We Do Better?

- Last-time and 2BC predictors exploit only “last-time” predictability for a given branch

- Realization 1: A branch’s outcome can be correlated with other branches’ outcomes
  - Global branch correlation

- Realization 2: A branch’s outcome can be correlated with past outcomes of the same branch (in addition to the outcome of the branch “last-time” it was executed)
  - **Local branch correlation**

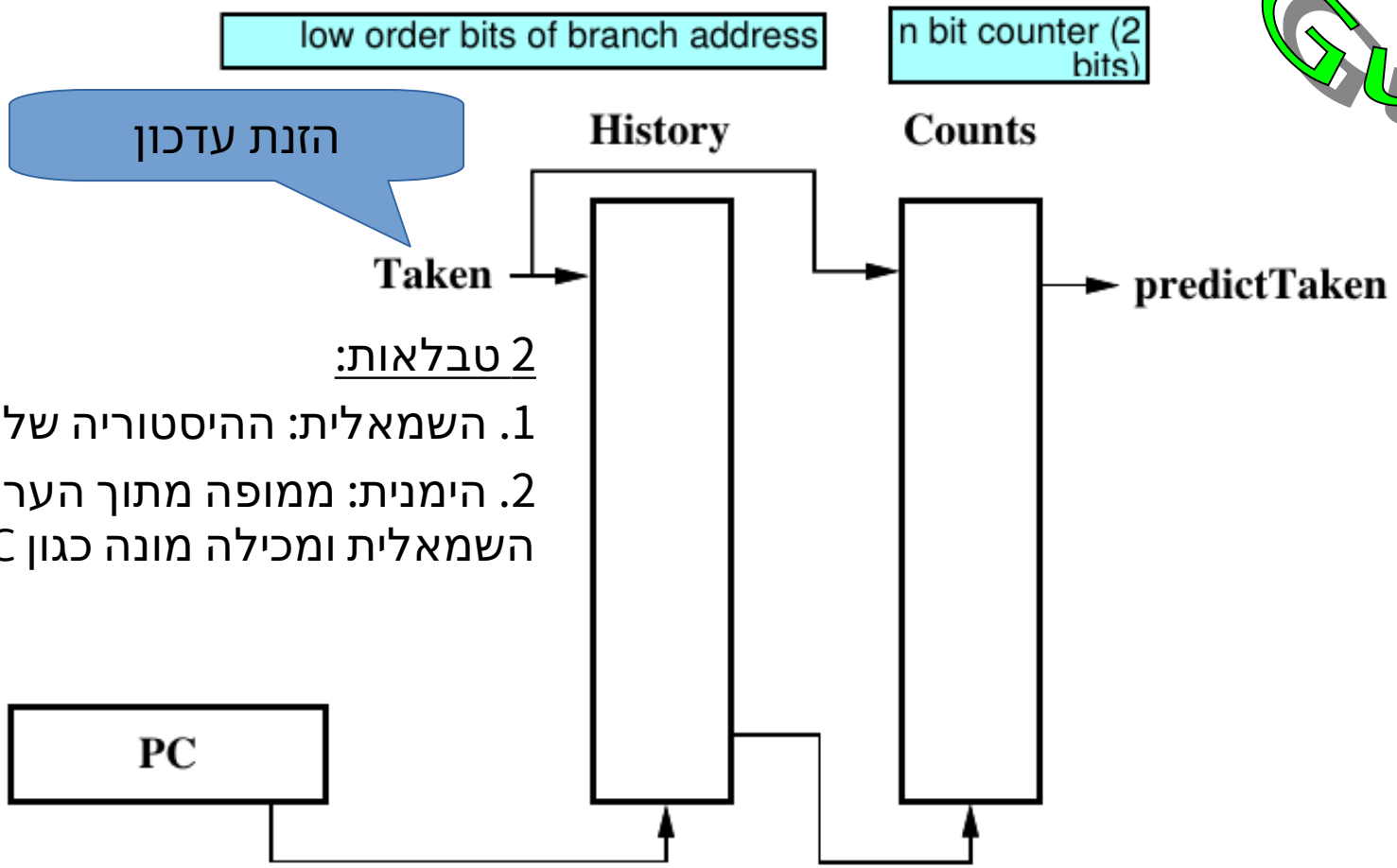
Local Branch Correlation

```
for (i=1; i<=4; i++) { }
```

If the loop test is done at the end of the body, the corresponding branch will execute the pattern $(1110)^n$, where 1 and 0 represent taken and not taken respectively, and n is the number of times the loop is executed. Clearly, if we knew the direction this branch had gone on the previous three executions, then we could always be able to predict the next branch direction.

- McFarling, "Combining Branch Predictors," DEC WRL TR 1993.

Scott McFarling, “Combining Branch Predictors”



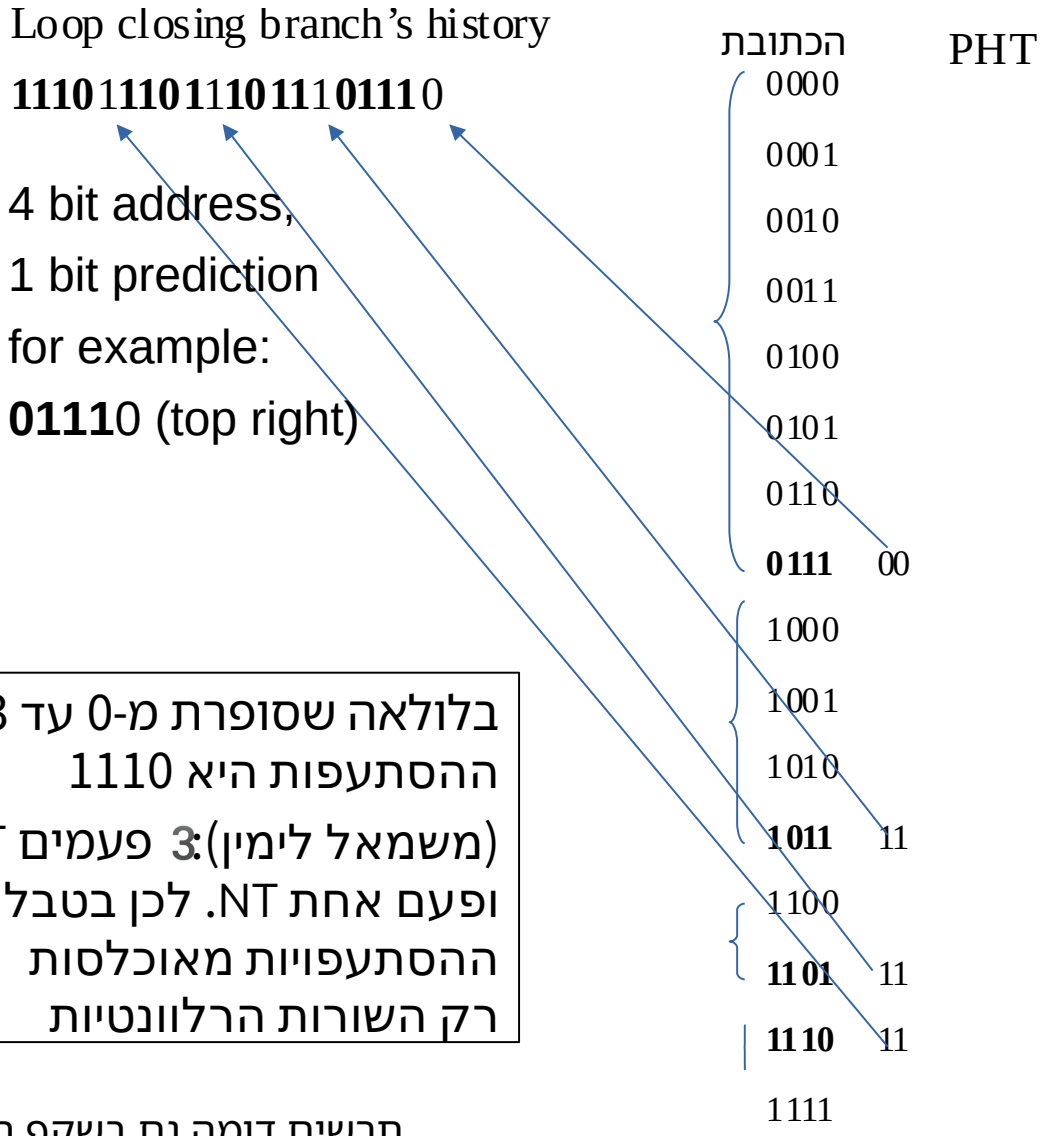
2 טבלאות:

1. השמאלית: ההיסטוריה של הסתעפויות
2. הימנית: ממופה מתוך הערך בטבלה השמאלית ומכילה מונה כגון 2BC.

Figure 4: Local History Predictor Structure

More Motivation for Local History

- To predict a loop branch “perfectly”, we want to identify the last iteration of the loop
- By having a separate PHT entry for each local history, we can distinguish different iterations of a loop
- Works for “short” loops

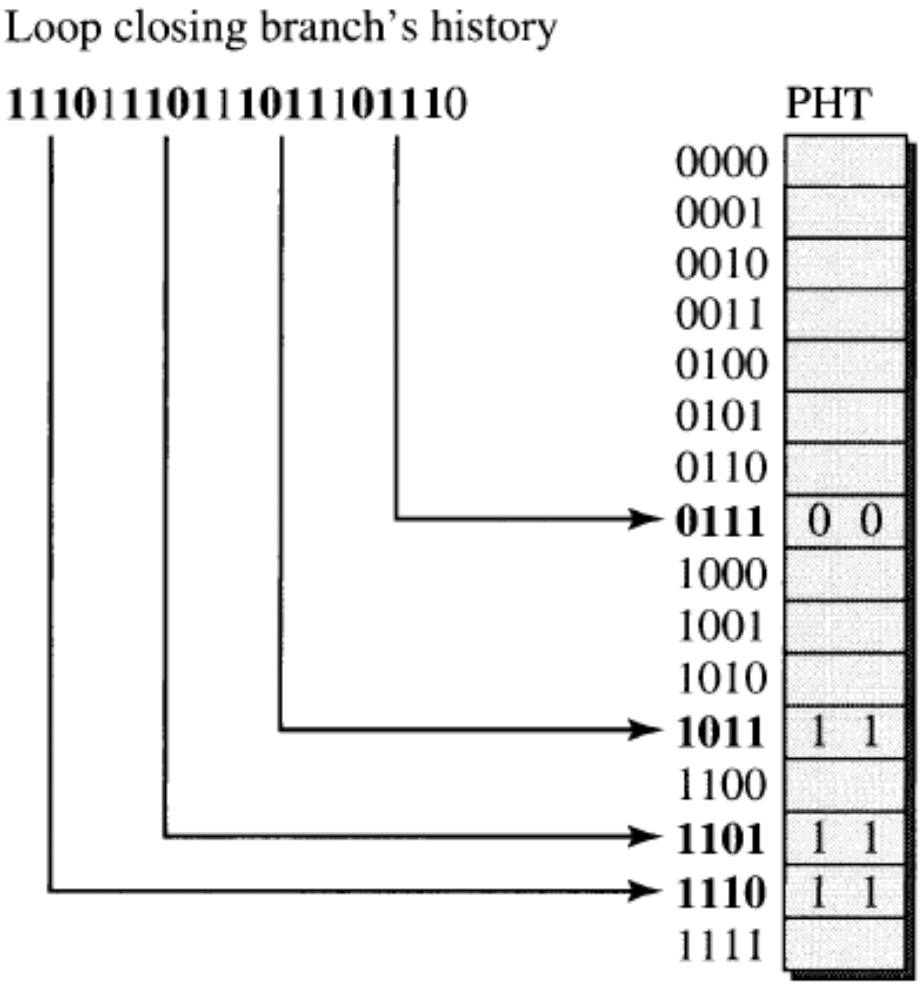


בלולאה שסופרת מ-0 עד 3 ההסתעפות היא 1110

(משמאל לימין): 3 פעמים T ופעם אחת NT. לכן בטבלת ההסתעפויות מאוכלסות רק השורות הרלוונטיות

תרשים דומה גם בשקף הבא...

Local History Predictor Example



תבנית של NT והמשך הלולאה

תבנית של סיום לולאה

Credit: Shen & Lipasti, Figure 9.7 :אותו התרשים, הפעם מתוך הספר

Capturing Local Branch Correlation

- Idea: Have a per-branch history register
  - Associate the predicted outcome of a branch with "T/NT history" of the same branch
- Make a prediction based on the outcome of the branch the last time the same local branch history was encountered

- Called the **local history/branch predictor**
- Uses two levels of history (Per-branch history register + history at that history register value)

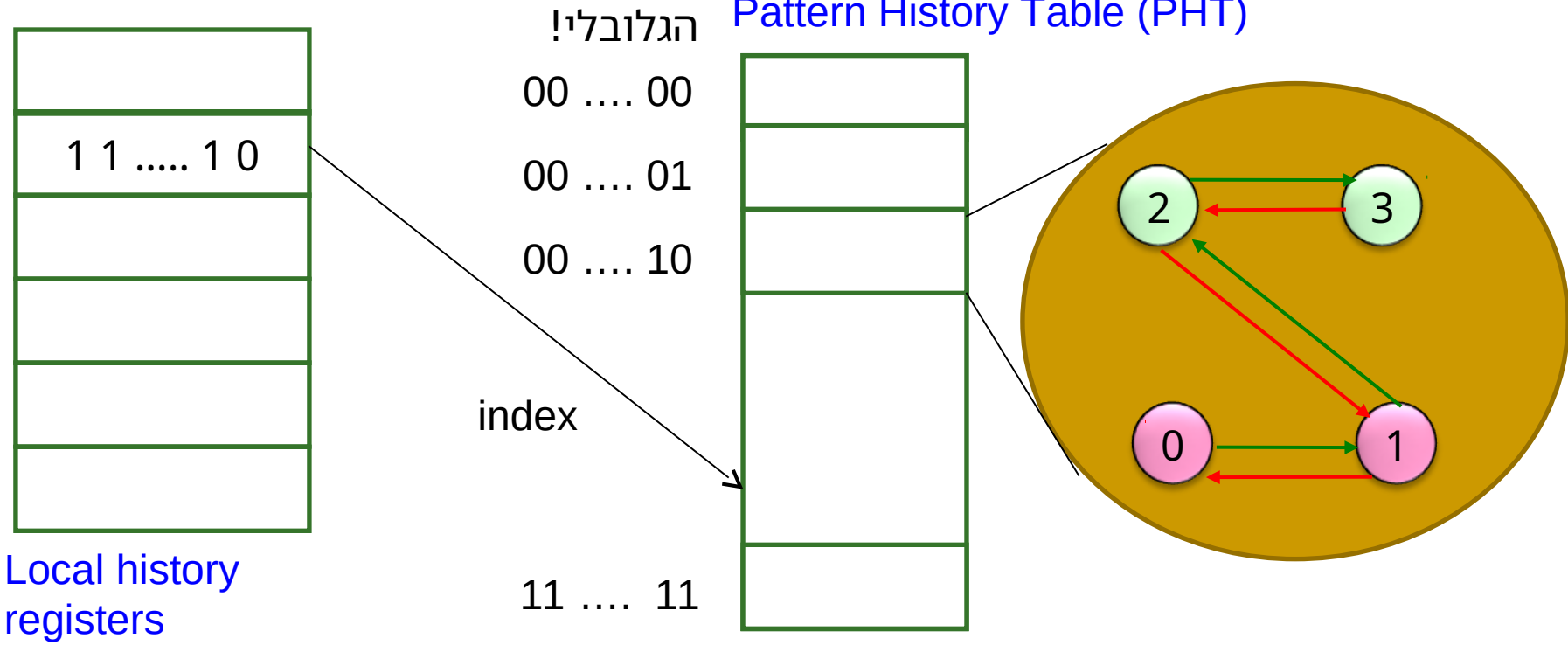
אנימציה

Two Level Local Branch Prediction

הסבר נוסף בשקף הבא

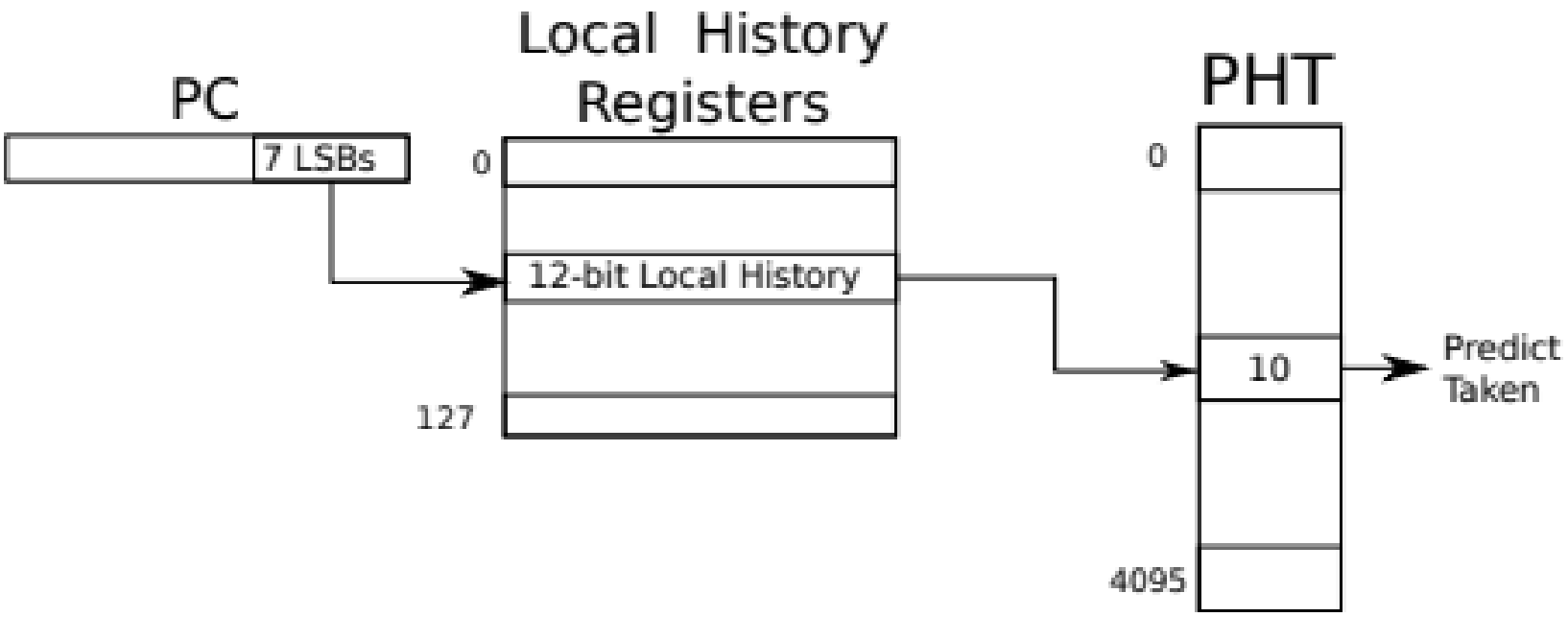
- First level: A set of local history registers (N bits each)
  - Select the history register based on the PC of the branch
- Second level: Table of saturating counters for each history entry
  - The direction the branch took the last time the same history was seen

הטיפול כאן דומה לזה שנעשה במטמון הגלובלי!



Yeh and Patt, "Two-Level Adaptive Training Branch Prediction," MICRO 1991.

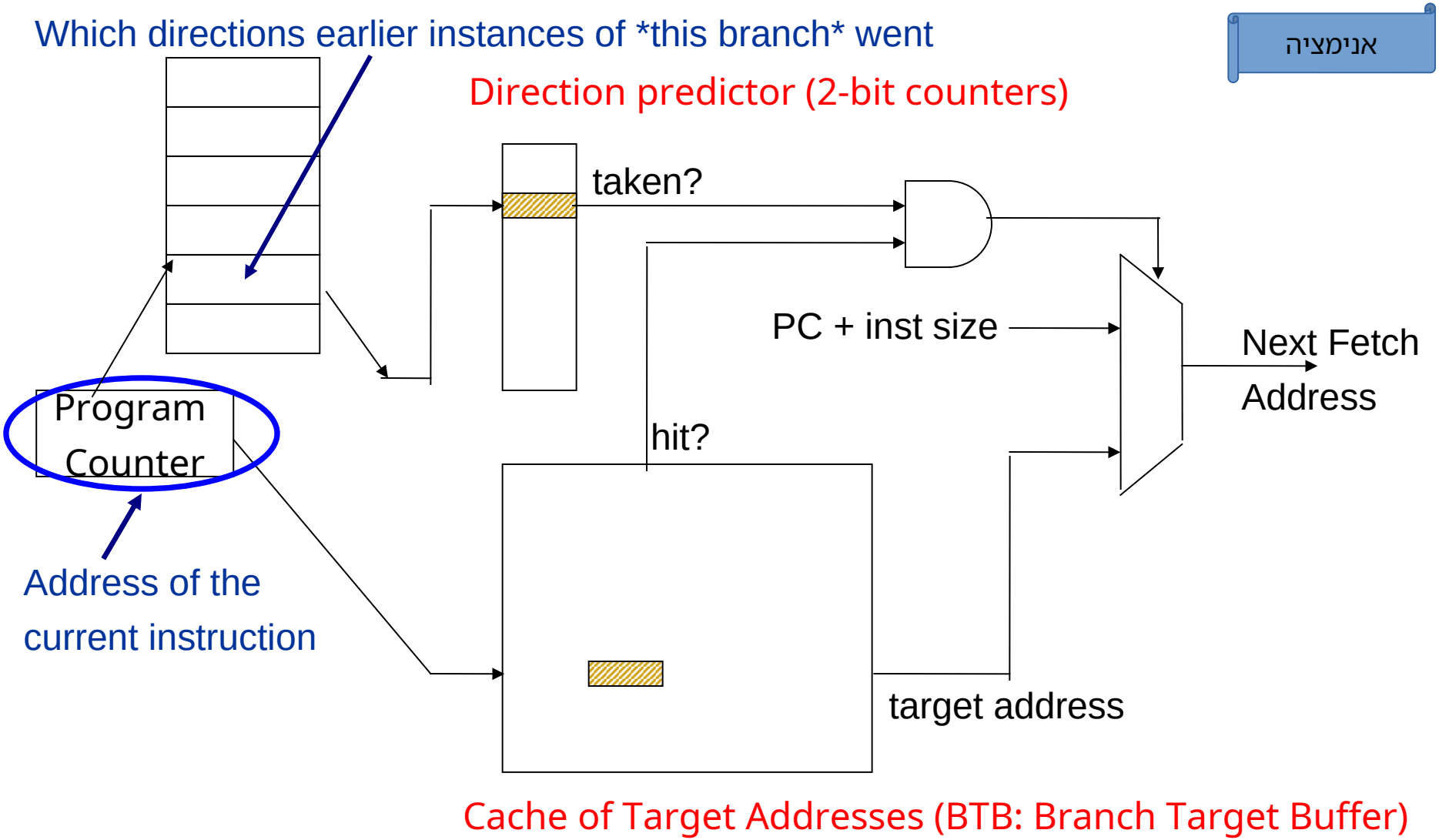
תרשים נוסף



2^7=128

2^7*12 + 2^12*2 = 1536 + 8192 = 9728bits=9.5kb

Two-Level Local History Branch Predictor



Can We Do Even Better?

- Predictability of branches varies

- Some branches are more predictable using local history
- Some branches are more predictable using global
- For others, a simple two-bit counter is enough
- Yet for others, a single bit is enough

- Observation: There is heterogeneity in predictability behavior of branches
  - **No one-size fits all** branch prediction algorithm for all branches
- Idea: Exploit that heterogeneity by designing heterogeneous (hybrid) branch predictors

Hybrid Branch Predictors

אנימציה

- 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:

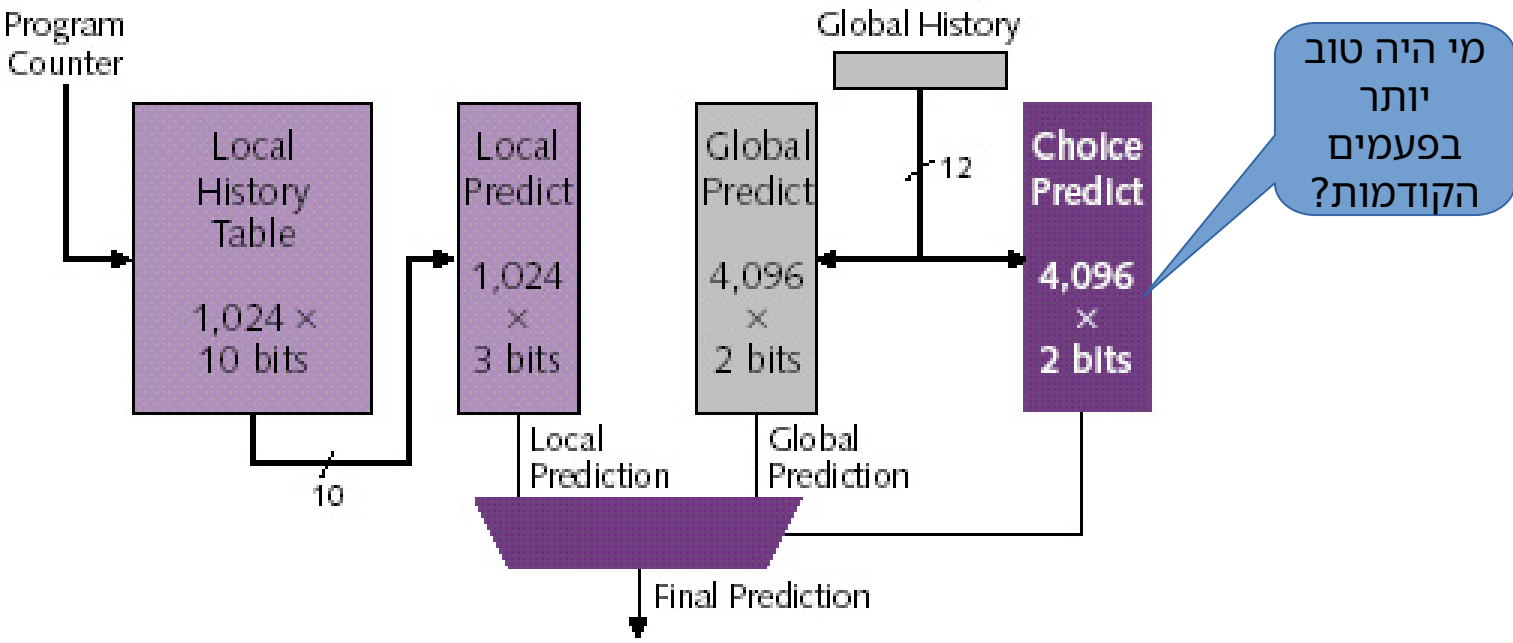
  -- Need "meta-predictor" or "selector"

  -- Longer access latency

  -- More hardware & complexity

  - McFarling, "Combining Branch Predictors," DEC WRL Tech Report, 1993.

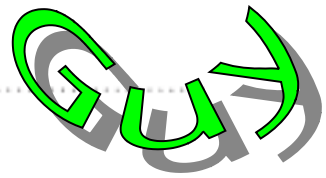
Alpha 21264 Tournament Predictor



בצבע אפור: חיזוי גלובלי

- Minimum branch penalty: 7 cycles
- Typical branch penalty: 11+ cycles
- 48K bits of target addresses stored in I-cache
- Predictor tables are reset on a context switch
- Kessler, "The Alpha 21264 Microprocessor," IEEE Micro 1999.

Kessler, "THE ALPHA 21264 MICROPROCESSOR"



ALPHA 21264

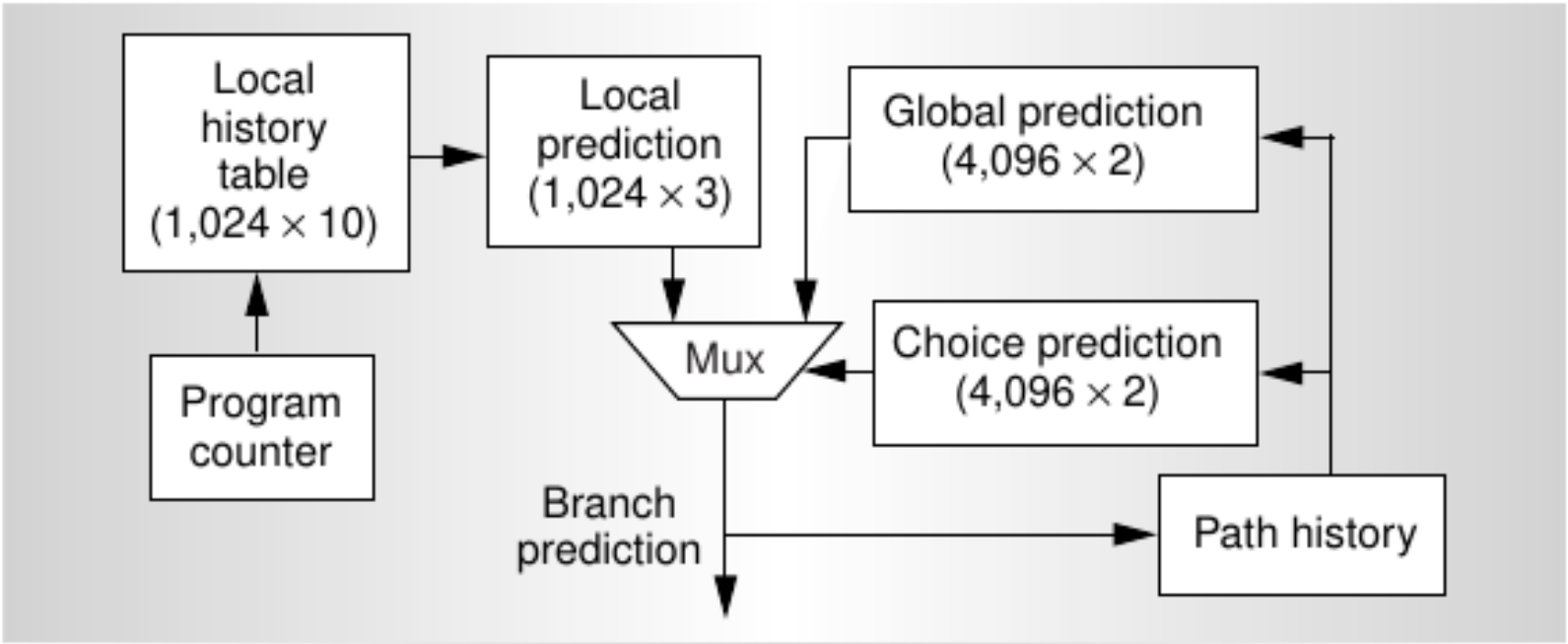


Figure 4. Block diagram of the 21264 tournament branch predictor. The local history prediction path is on the left; the global history prediction path and the chooser (choice prediction) are on the right.

Biased Branches and Branch Filtering

- Observation: Many branches are biased in one direction (e.g., 99% taken)

- Problem: These branches *pollute* the branch prediction structures → make the prediction of other branches difficult by causing "interference" in branch prediction tables and history registers

- Solution: Detect such biased branches, and predict them with a simpler predictor (e.g., last time, static, ...)

  נעדיף להשתמש במנגנון הפשוט ביותר לצורך החיזוי (אם ניתן)

- Chang et al., "Branch classification: a new mechanism for improving branch predictor performance," MICRO 1994.

Are We Done w/ Branch Prediction?

- Hybrid branch predictors work well
  - E.g., 90-95% prediction accuracy on average

- Some “difficult” workloads still suffer, though!
  - E.g., gcc
  - Max IPC with tournament prediction: 9
  - Max IPC with perfect prediction: 35

9/35 ≈ 26% prediction accutacy

Some Other Branch Predictor Types

- Loop branch detector and predictor – מנגנון חיזוי המתמחה בלולאות
  - Loop iteration count detector/predictor
  - Works well for loops with small number of iterations, where iteration count is predictable
  - Used in Intel Pentium M
- Perceptron branch predictor (Global)
  - Learns the *direction correlations* between individual branches
  - Assigns weights to correlations
  - Jimenez and Lin, "Dynamic Branch Prediction with Perceptrons," HPCA 2001.
  - Used in AMD Zen/Zen2 family
- Hybrid history length based predictor
  - Uses different tables with different history lengths
  - Seznec, "Analysis of the O-Geometric History Length branch predictor," ISCA 2005.
  - Used in AMD Zen/Zen2 family

Intel Pentium M Predictors: Loop and Jump

The advanced branch prediction in the Pentium M processor is based on the Intel Pentium® 4 processor's [6] branch predictor. On top of that, two additional predictors to capture special program flows, were added: a Loop Detector and an Indirect Branch Predictor.

A Single Entry of the Loop Predictor Table Used in the Pentium-M Processor.

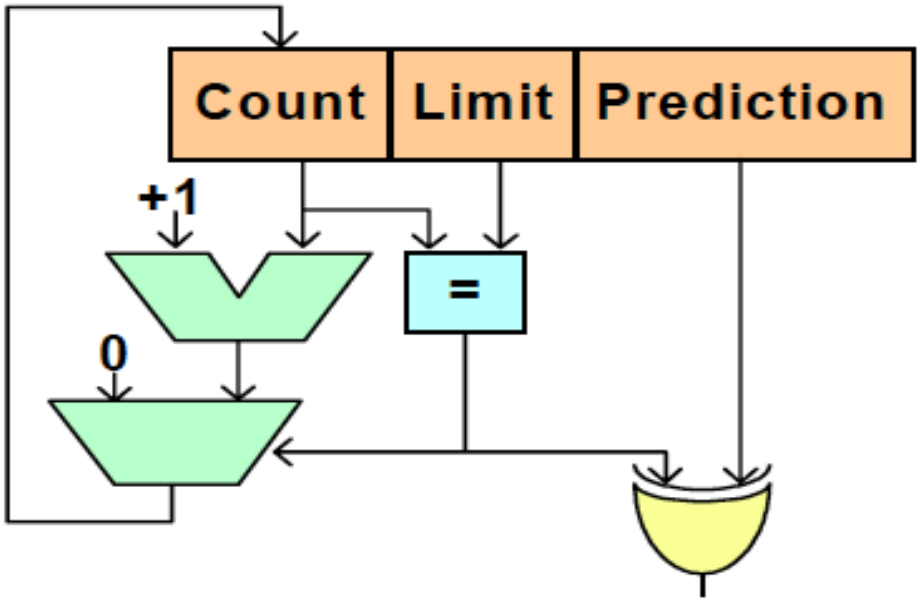


Figure 2: The Loop Detector logic

Gochman et al.,

"The Intel Pentium M Processor: Microarchitecture and Performance,"

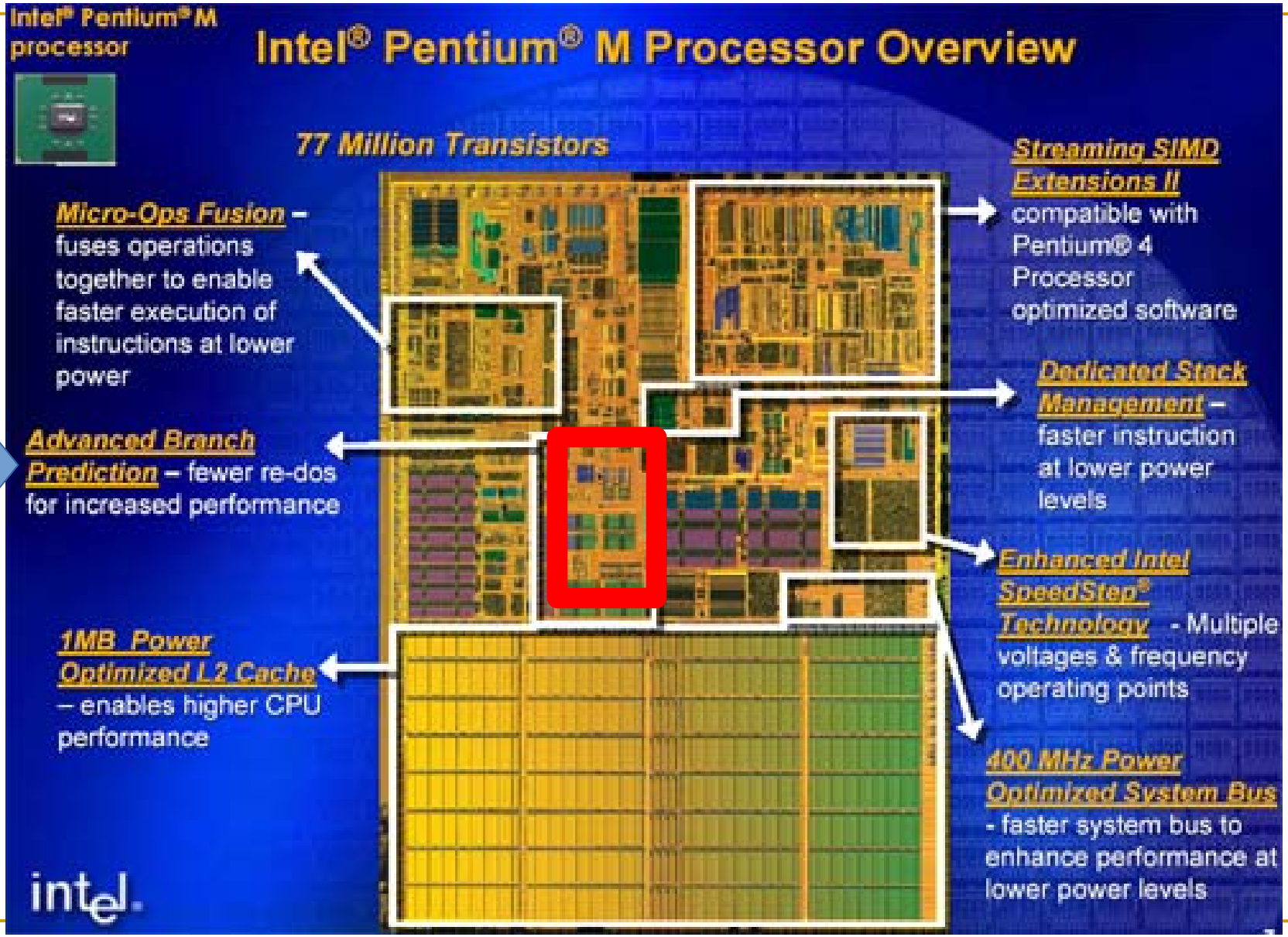
Intel Technology Journal, May 2003.

הסבר מתוך המאמר ניתן בשקף הבא

The Loop Detector (Figure 2) analyzes branches to see if they have loop behavior. Loop behavior is defined as moving in one direction (taken or not-taken) a fixed number of times interspersed with a single movement in the opposite direction. When such a branch is detected, a set of counters are allocated in the predictor such that the behavior of the program can be predicted completely accurately for larger iteration counts than typically captured by global or local history predictors.

Intel Pentium M (2003)

אנימציה



https://www.anandtech.com/show/1083/3

More Advanced Branch Prediction

Perceptrons for Learning Linear Functions

- A perceptron is a simplified model of a biological neuron
- It is also a simple **binary classifier**
- A perceptron maps an input vector X to a 0 or 1
  - Input = Vector X (סט של ערכים)
  - Perceptron learns the linear function (if one exists) of how each element of the vector affects the output (stored in an internal Weight vector)
  - Output = Weight·X + Bias, if > 0 → Output=1 else Output=0;

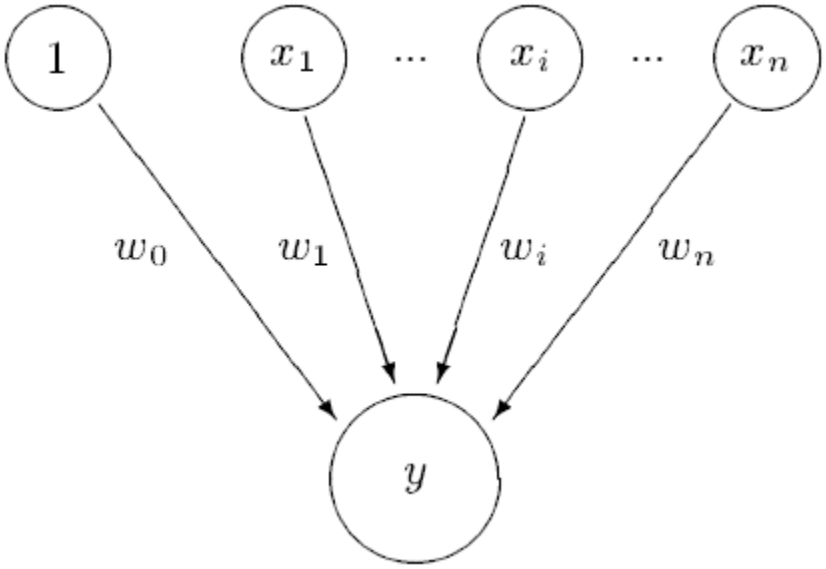
מתן משקלים להיסטוריה של ההסתעפויות

- In the branch prediction context
  - Vector X: Branch history register bits
  - Output: Prediction for the current branch, 1=Taken, 0=NT

Rosenblatt, "Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms," 1962

Perceptron Branch Predictor (I)

- Idea: Use a perceptron to learn the correlations between branch history register bits and branch outcome
- A perceptron learns a target Boolean function of N inputs



$$y = w_0 + \sum_{i=1}^{n} x_i w_i.$$

Each branch associated with a perceptron

A perceptron contains a set of weights wi

→Each weight corresponds to a bit in the GHR

→How much the bit is correlated with the direction of the branch

→ Positive correlation: large positive weight

→ Negative correlation: large negative weight

Prediction:

→ Express GHR bits as 1 (T) and -1 (NT)

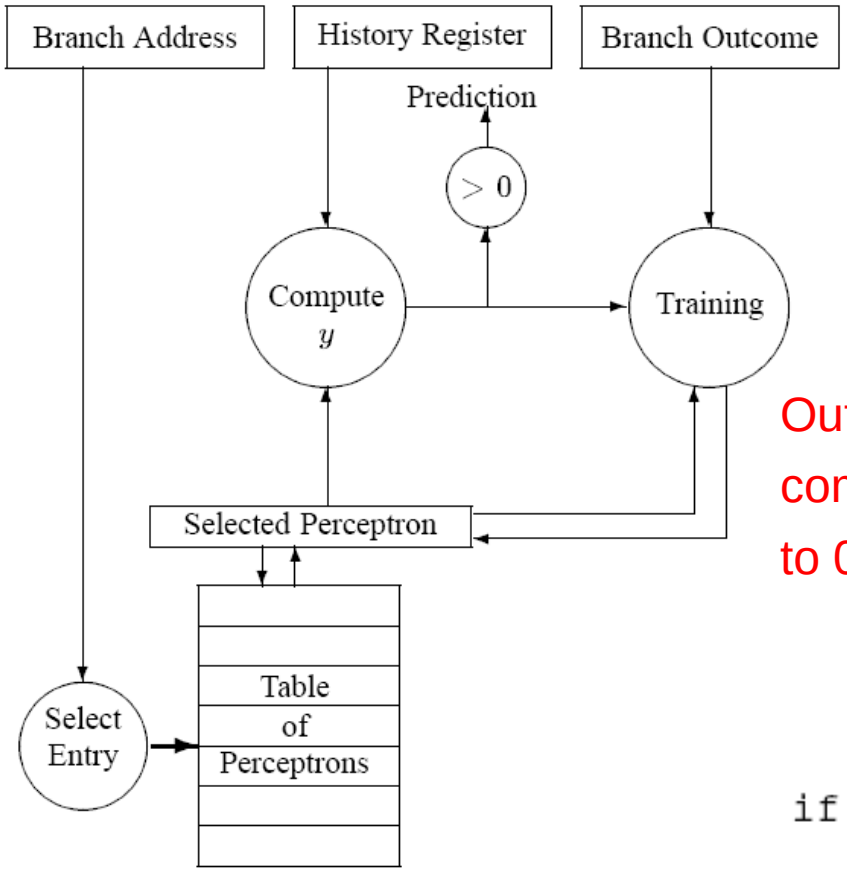
→ Take dot product of GHR and weights

→ If output > 0, predict taken

- Jimenez and Lin, "Dynamic Branch Prediction with Perceptrons," HPCA 2001.

Perceptron Branch Predictor (II)

אנימציה



Prediction function:

Dot product of GHR and perceptron weights

$$y = w_0 + \sum_{i=1}^{n} x_i w_i.$$

Output compared to 0

Bias weight (bias of branch, independent of the history)

t={ -1, +1}, let theta be the threshold, a parameter to the training algorithm used to decide when enough training has been done.

Training function:

```
if sign(y_out) ≠ t or |y_out| ≤ θ then
        for i := 0 to n do
                w_i := w_i + t x_i
        end for
end if
```

מורכבות גבוהה. יש צורך בביצוע פעולות כפל מרובות

Daniel A. Jiménez and Calvin Lin, “Dynamic Branch Prediction with Perceptrons”

Perceptron Branch Predictor (III)

- Advantages
  - + More sophisticated learning mechanism → better accuracy
  - + Enables long branch history lengths → better accuracy

- Disadvantages
  - -- Complexity (adder tree to compute perceptron output)
  - -- Can learn only linearly-separable functions
    - e.g., cannot learn XOR type of correlation between 2 history bits and branch outcome

A successful example of use of machine learning in processor design

See, e.g., Grayson+, "Evolution of the Samsung Exynos CPU Microarchitecture," ISCA 2020.

Recommended Reading

2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)

Evolution of the Samsung Exynos CPU Microarchitecture

Industrial Product

Brian Grayson*, Jeff Rupley†, Gerald Zuraski Jr.‡, Eric Quinnell§, Daniel A. Jiménez¶, Tarun Nakra‡‡, Paul Kitchin**, Ryan Hensley††, Edward Brekelbaum*, Vikas Sinha**, and Ankit Ghiya§

bgrayson@ieee.org, jrupley@austin.rr.com, gzuraskijr@gmail.com, eric.quinnell@gmail.com, djimenez@acm.org, Tarun.Nakra@amd.com, pkitchin@gmail.com, ryan.hensley@ieee.org, nedbrek@gmail.com, sinhavk@gmail.com, and mascot26@gmail.com

*Sifive †Centaur ‡Independent Consultant §ARM ¶Texas A&M University ‡‡AMD **Nuvia ††Goodix

***Abstract*—The Samsung Exynos family of cores are high-performance "big" processors developed at the Samsung Austin Research & Design Center (SARC) starting in late 2011. This paper discusses selected aspects of the microarchitecture of these cores - specifically perceptron-based branch prediction, Spectre v2 security enhancements, micro-operation cache algorithms, prefetcher advancements, and memory latency optimizations. Each micro-architecture item evolved over time, both as part of continuous yearly improvement, and in reaction to changing mobile workloads.**

***Index Terms*—microprocessor, superscalar, branch prediction, prefetching**

- Deep technical details within the microarchitecture.

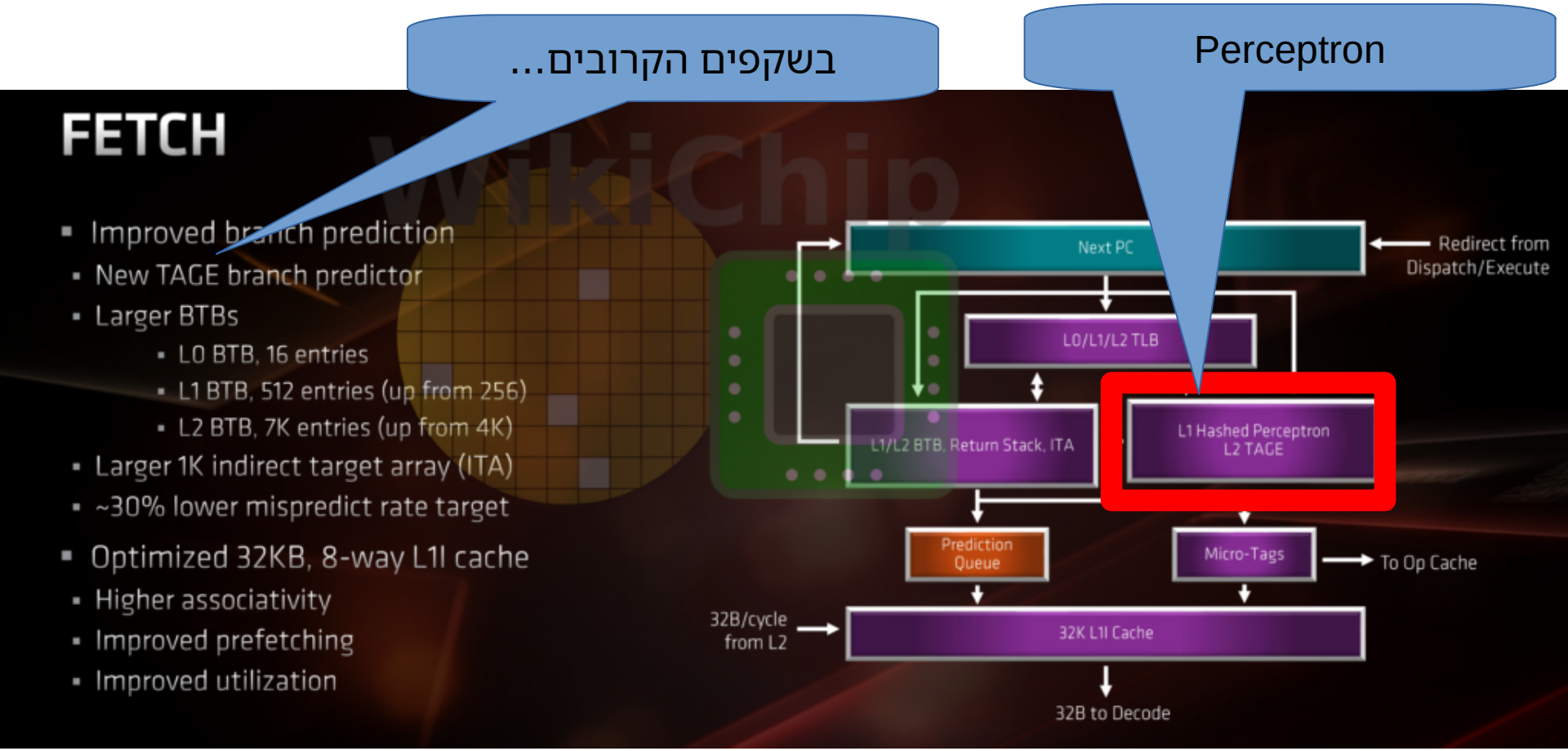
The remainder of this paper discusses several aspects of front-end microarchitecture (including branch prediction microarchitecture, security mitigations, and instruction supply) as well as details on the memory subsystem, in particular with regards to prefetching and DRAM latency optimization. The overall generational impact of these and other changes is presented in a cross-workload view of IPC.

II. Methodology

Grayson+, "Evolution of the Samsung Exynos CPU Microarchitecture," ISCA 2020.

AMD Piledriver/Zen/Zen2 (2012-Present)

- Employ a perceptron branch predictor



https://fuse.wikichip.org/news/2458/a-look-at-the-amd-zen-2-core/

Another Idea: TAGE

Prediction Using Multiple History Lengths

אורך היסטוריות מהקטן אל הגדול

- Observation: Different branches require different history lengths for better prediction accuracy

פרדיקטור פשוט

- Idea: Have multiple PHTs indexed with GHRs with different history lengths and intelligently allocate PHT entries to different branches

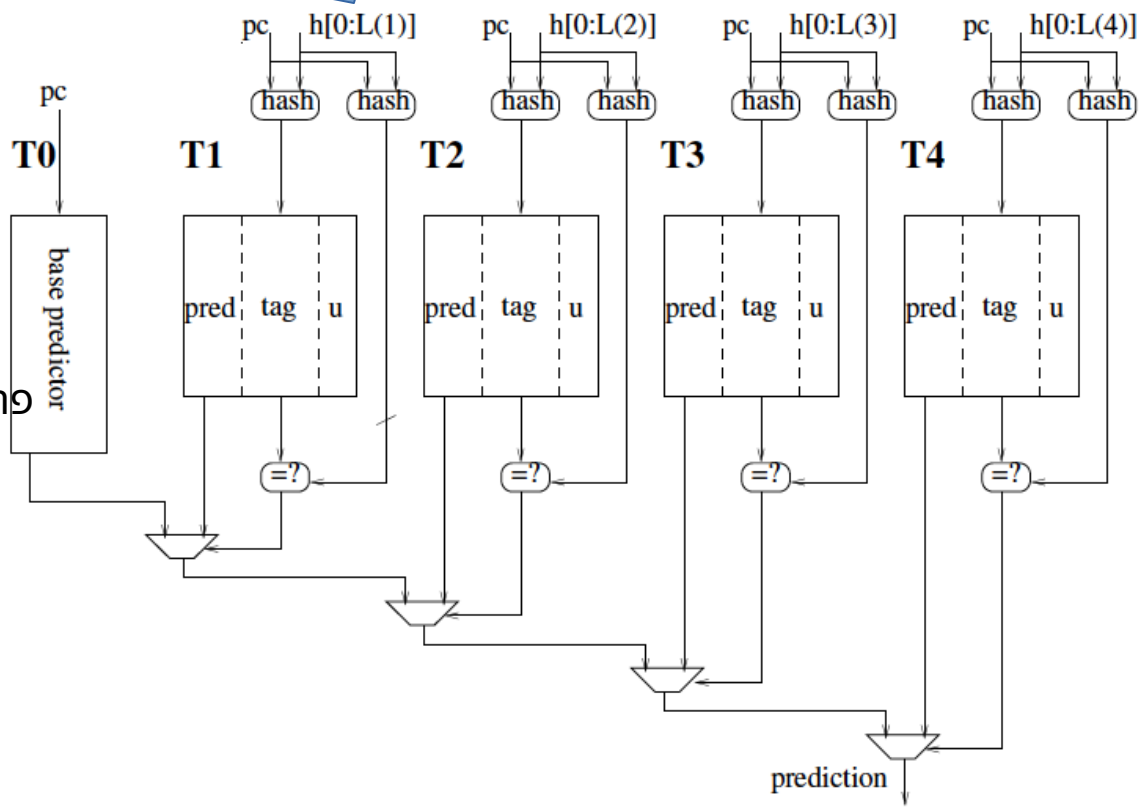
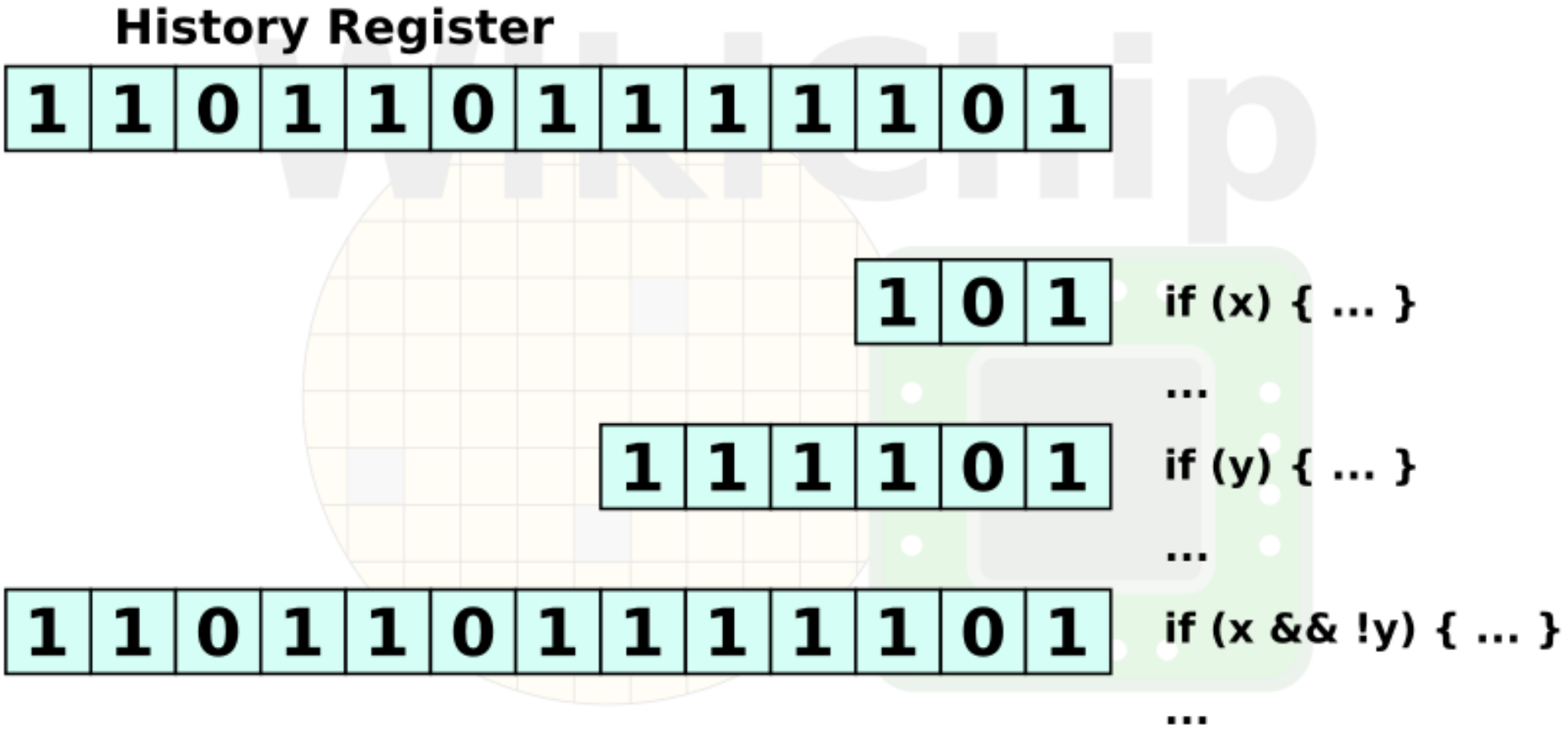


Figure 1: A 5-component TAGE predictor synopsis: a base predictor is backed with several tagged predictor components indexed with increasing history lengths

Seznec and Michaud, "A case for (partially) tagged Geometric History Length Branch Prediction," JILP 2006.

סדרה גדלה גיאומטרית של מנגנוני חיזוי על-פי היסטוריה

Different Branches: Different History Lengths



"...An entry on a tagged table consists of the partial tag, the jump target, the confidence bit and a 2-bit useful counter"

https://fuse.wikichip.org/news/2458/a-look-at-the-amd-zen-2-core/

TAGE Branch Predictor

- Advantages
  - + Chooses the “best” history length to predict each branch → better accuracy
  - + Enables long branch history lengths → better accuracy
- Disadvantages
  - -- Hardware (design) complexity is not low
  - -- Need to choose good hash functions and table sizes to maximize accuracy and minimize latency

A successful recent idea that is used in many modern processor designs

Seznec & Michaud

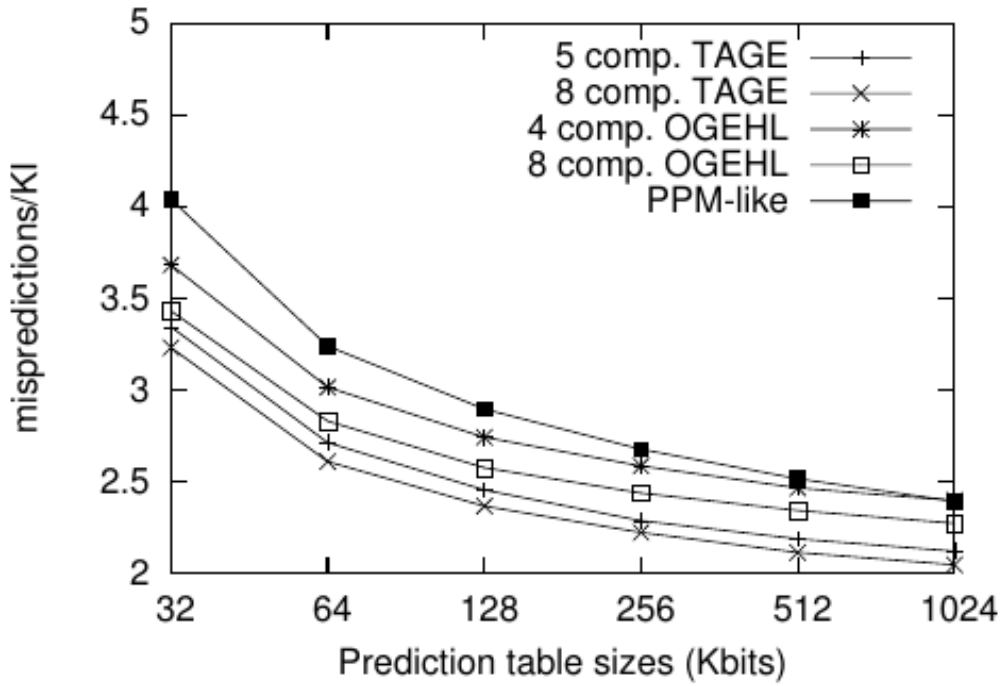
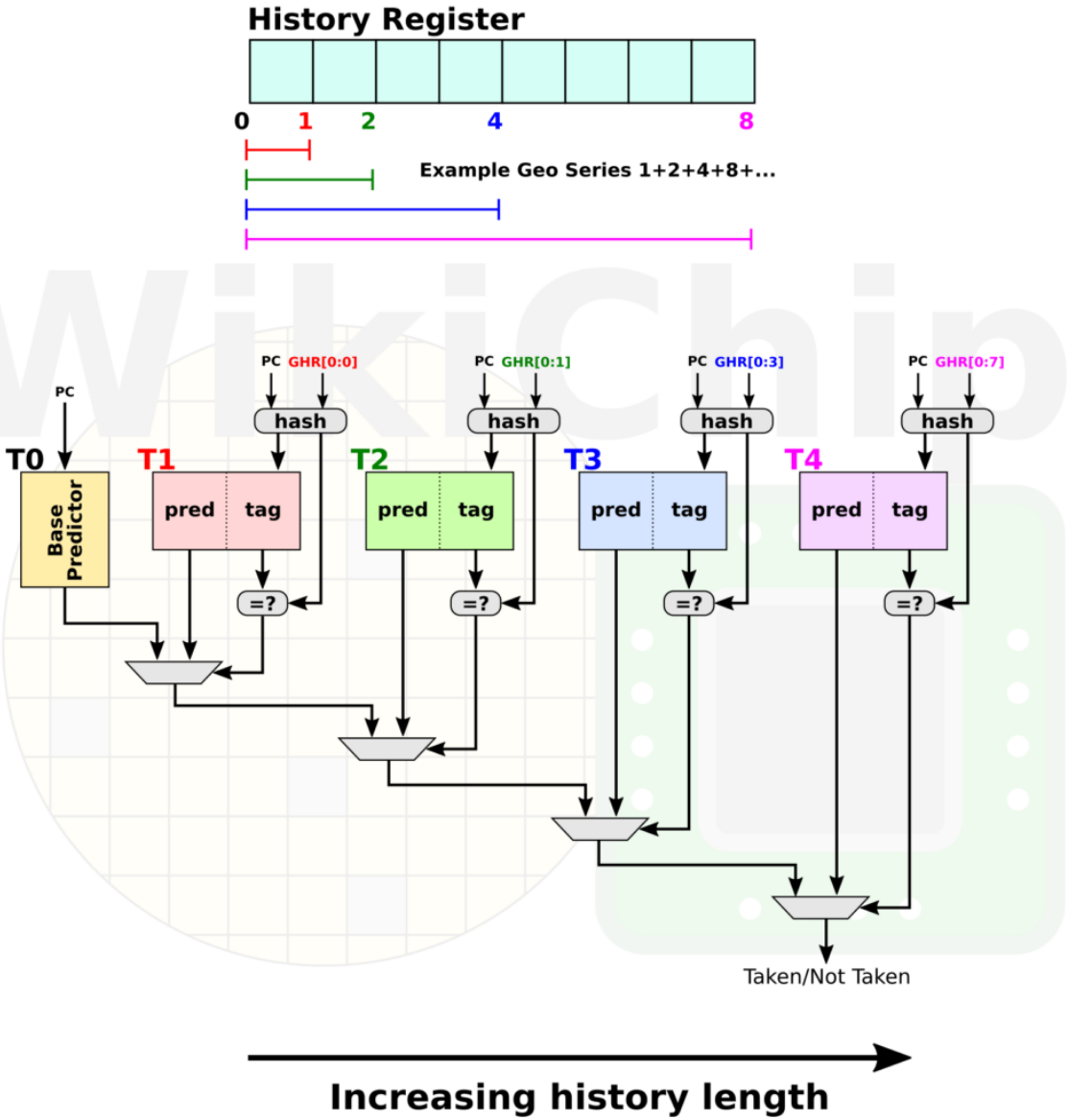


Figure 2: Conditional branch prediction accuracy on the TAGE predictor

comp = component

AMD Zen2 TAGE Predictor (2019)



https://fuse.wikichip.org/news/2458/a-look-at-the-amd-zen-2-core/

State of the Art in Branch Prediction

אנימציה

- See the Branch Prediction Championship
  - https://www.jilp.org/cbp2016/program.html

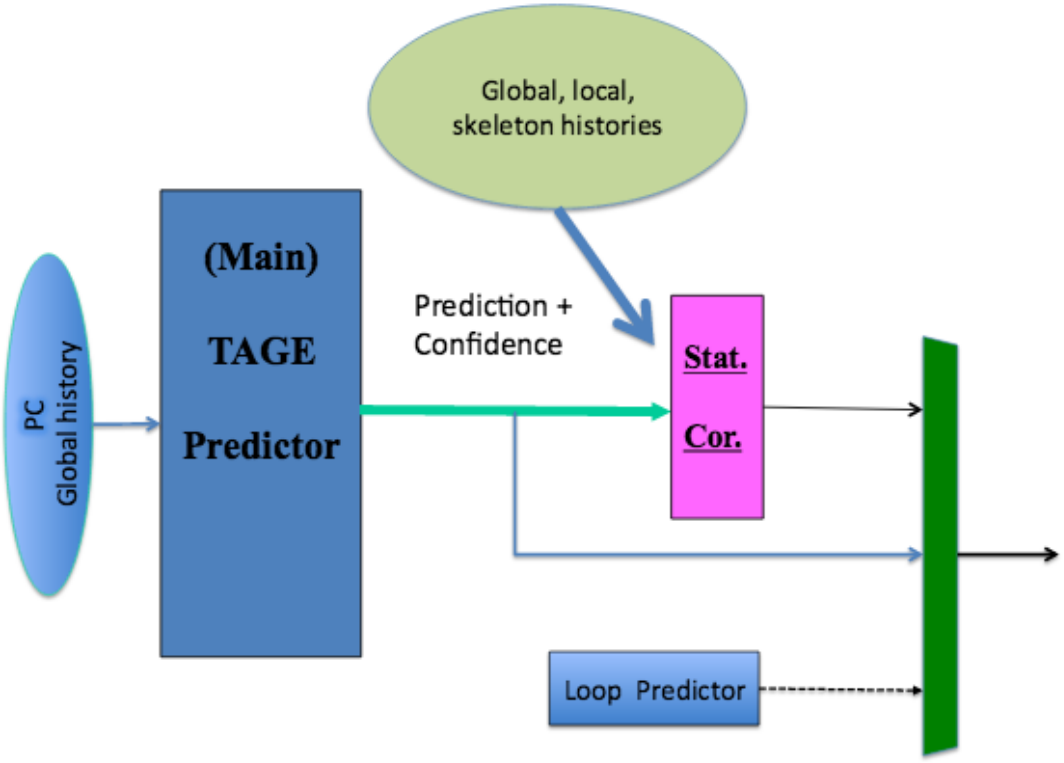
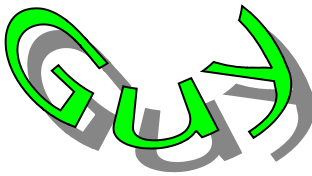


Figure 1. The TAGE-SC-L predictor: a TAGE predictor backed with a Statistical Corrector predictor and a loop predictor

Andre Seznec,
"TAGE-SC-L branch predictors,"
CBP 2014.

Andre Seznec,
"TAGE-SC-L branch predictors again," CBP 2016.

https://www.jilp.org/cbp2016/program.html



Championship Branch Prediction
Program
Saturday, June 18, 2016

Time	Title and Authors	Paper	Code	Presentation
2:00 - 2:15 pm	Introduction and Welcome *Trevor Mudge (The University of Michigan, Ann Arbor, USA)*			slides
2:15 - 2:30 pm	Discussion of trace selection *Max Breughe (Samsung SARC, Austin, USA)*			slides
2:30 - 2:50 pm	TAGE-SC-L Branch Predictors Again *Andre Seznec (INRIA/IRISA, France)*	pdf	tarball	slides
2:50 - 3:10 pm	Exploring branch predictability limits with the MTAGE+SC predictor *Andre Seznec (INRIA/IRISA, France)*	pdf	tarball	slides
3:10 - 3:30 pm	Multiperspective Perceptron Predictor *Daniel Jimenez (Texas A&M University, College Station, USA)*	pdf	tarball	slides
3:30 - 4:00 pm	==========BREAK==========			
4:00 - 4:20 pm	Multiperspective Perceptron Predictor with TAGE *Daniel Jimenez (Texas A&M University, College Station, USA)*	pdf	tarball	slides
4:20 - 4:40 pm	Dynamically Sizing the TAGE Branch Predictor *Stephen Pruett, Siavash Zangeneh, Ali Fakhrzadehgan, Ben Lin and Yale Patt (University of Texas, Austin, USA)*	pdf	tarball	slides
4:40 - 5:00 pm	Announcement of results and awards *James Dundas (Samsung SARC, Austin, USA)*			slides

“For pioneering contributions to branch prediction and cache memories.”

2025 ACM/IEEE CS ECKERT-MAUCHLY AWARD

The Eckert-Mauchly Award recognizes outstanding contributions to computer and digital systems architecture. The award was named for John Presper Eckert and John William Mauchly, who collaborated on the design and construction of the Electronic Numerical Integrator and Computer (ENIAC), the first large-scale computing machine, which was completed in 1947. A certificate and US$5,000 are awarded jointly by the ACM and the IEEE CS for outstanding contributions to the field of computer and digital systems architecture.

ANDRÉ SEZNEC
SiFive

2025 ACM/IEEE CS Eckert-Mauchly Award

“For pioneering contributions to branch prediction and cache memories.”

Branch Confidence Estimation

- Idea: Estimate if the prediction is likely to be correct
  - i.e., estimate how "confident" you are in the prediction

- Why?
  - Could be very useful in deciding how to speculate:
    - What predictor/PHT to choose/use
    - Whether to keep fetching on this path
    - Whether to switch to some other way of handling the branch, e.g. dual-path execution (eager execution) or dynamic predication
    - …

- Jacobsen et al., "Assigning Confidence to Conditional Branch Predictions," MICRO 1996.

Speculative Processor

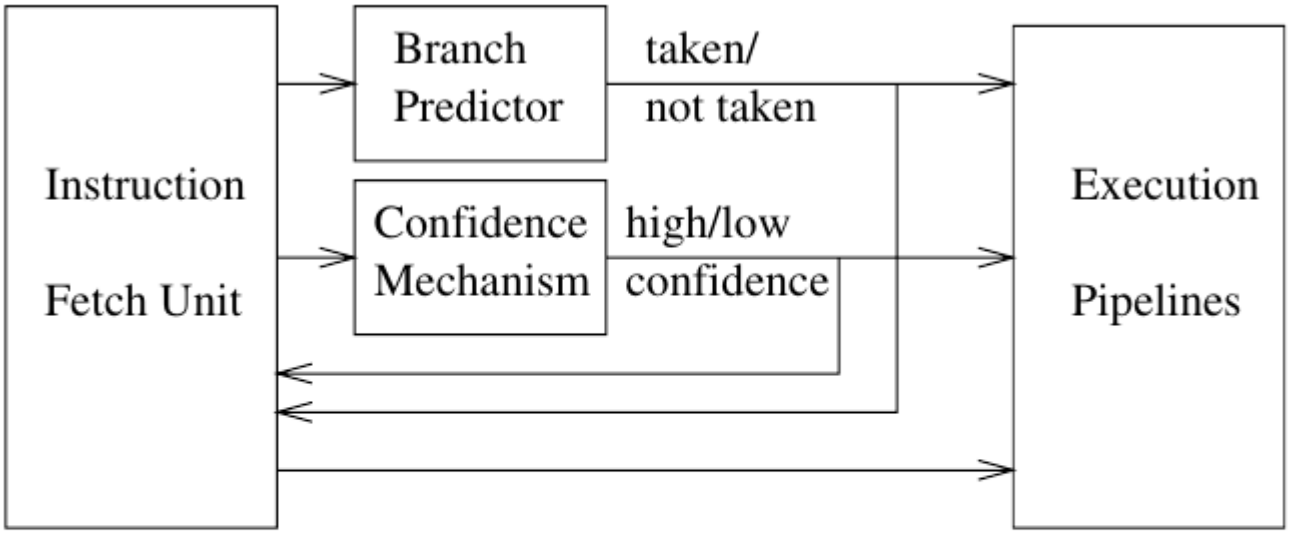
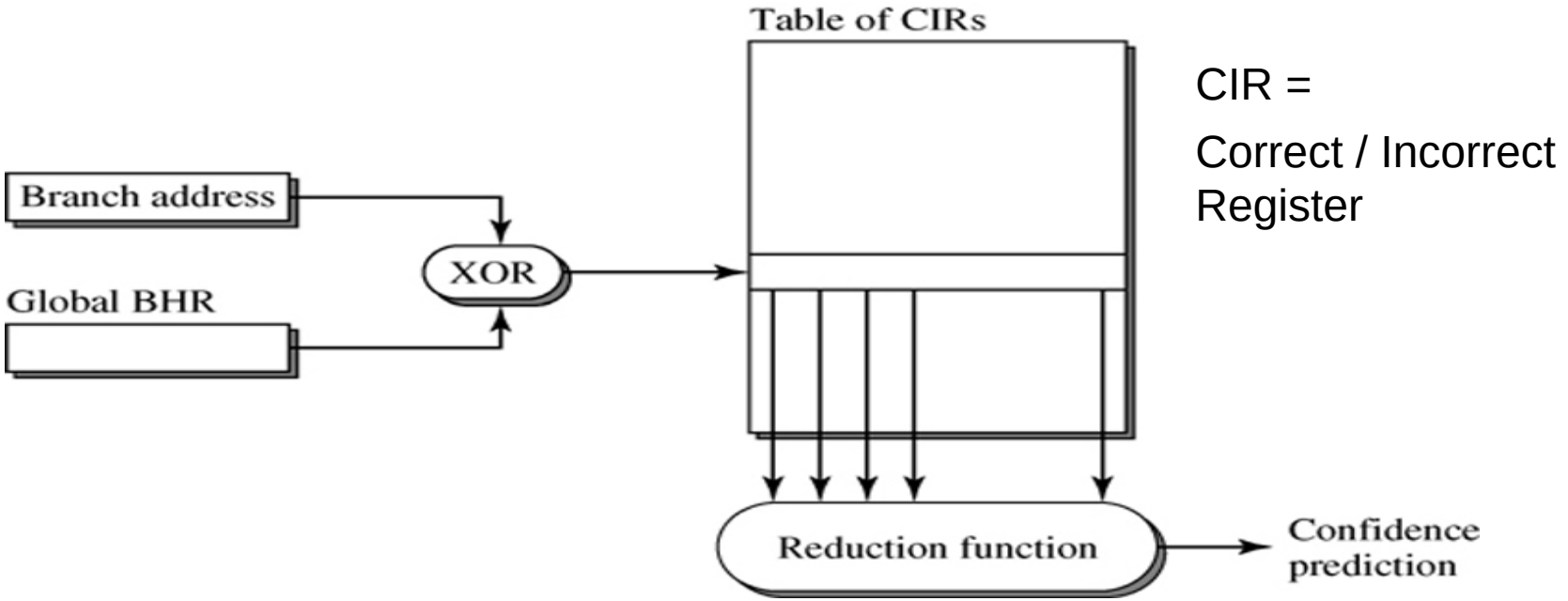


Fig. 1. A generic speculative processor containing a branch prediction signal paired with high/low confidence signal.

How to Estimate Confidence

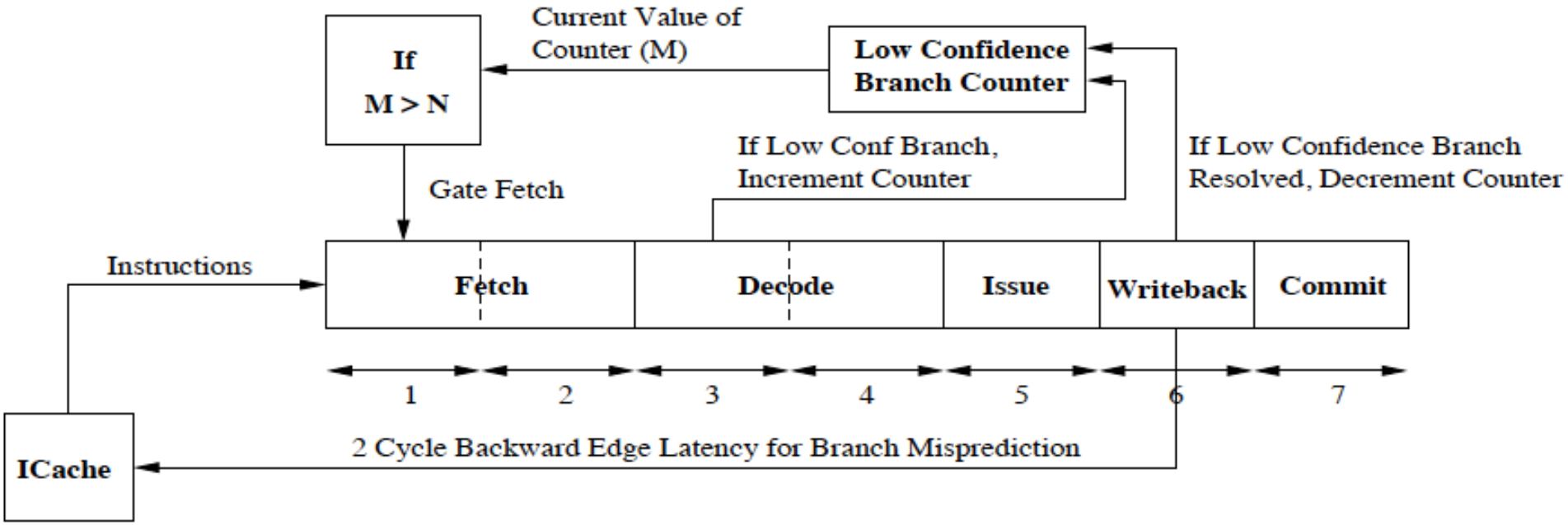
- An example estimator:
  - Keep a record of correct/incorrect outcomes for the past N instances of the “branch”
  - Based on the correct/incorrect patterns, guess if the current prediction will likely be correct/incorrect



Jacobsen et al., “Assigning Confidence to Conditional Branch Predictions,” MICRO 1996.

What to Do With Confidence Estimation?

- An example application: Pipeline Gating



מתן דרוג (ציון) לאמינות החיזוי.

N הוא ערך סף. אם M>N אז ייפסק הטיפול בחיזוי הנוכחי (כי החיזוי לא אמין). M גבוה פירושו חיזוי לא אמין.

Manne et al., "Pipeline Gating: Speculation Control for Energy Reduction," ISCA 1998.

מתוך המאמר:

5 Conclusion

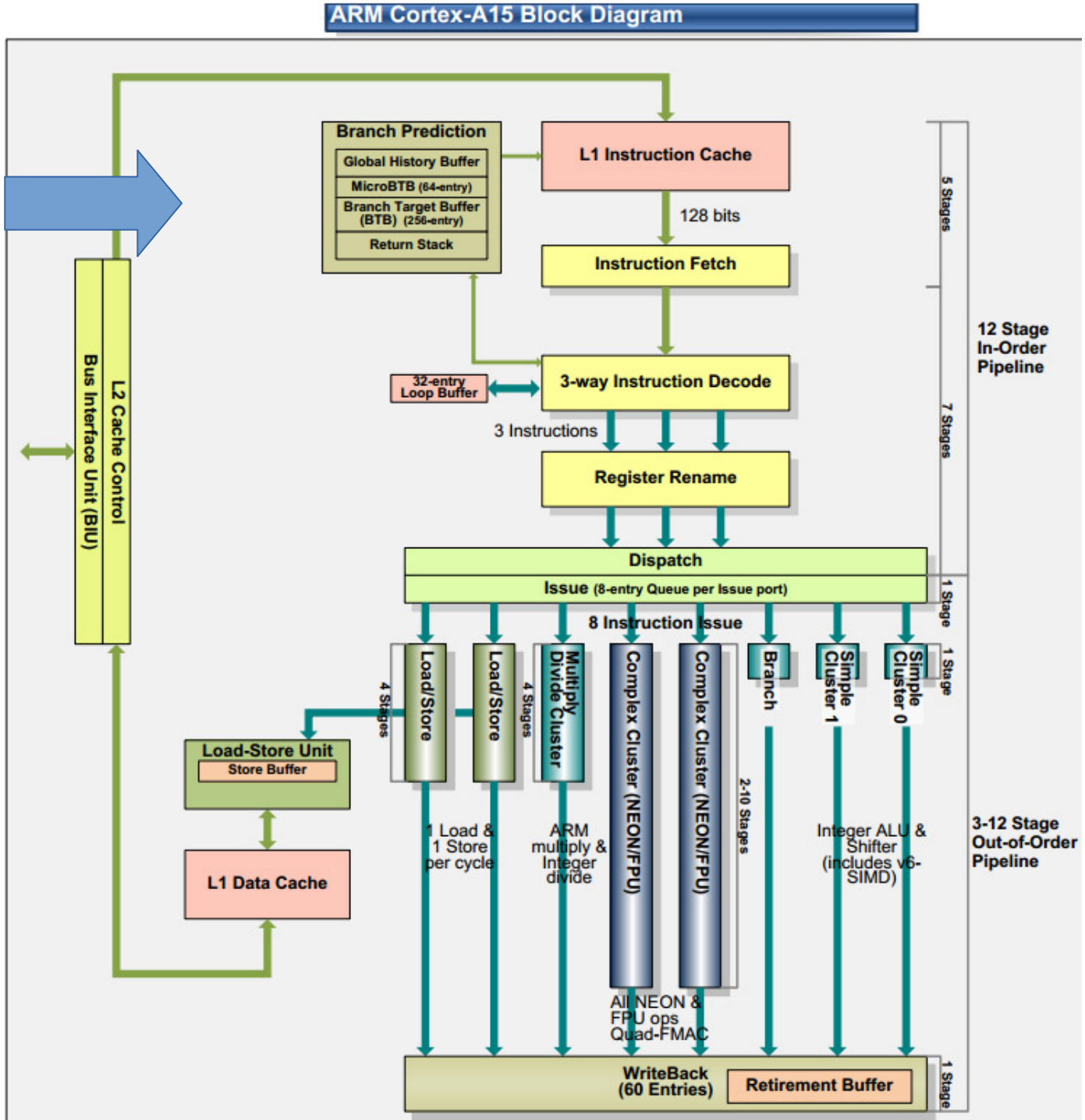
חסכון אנרגטי

We have looked at speculation control to reduce the amount of energy consumed in a speculative, multi-issue, out-of-order processor. We introduced a new mechanism, pipeline gating, which results in a reduction of instructions in the pipeline without significantly altering performance. We have shown results for different branch predictors and confidence estimators, and implemented inexpensive dynamic confidence estimation methods that do a reasonable job of reducing unnecessary work. Furthermore, we presented a practical configuration for the JRS confidence estimator that successfully reduces energy without a large hardware penalty. Most importantly, we showed that inexpensive, dynamic confidence estimation mechanisms exist which, at worst, do not impact performance for highly predictable programs, and at best, reduce work by a measurable amount for programs with a large misprediction rate.

יתרון השיטה

חסכון אנרגטי

Architectural level power reduction in high performance processors is a broad field and one that is in its infancy. We have presented an innovative method for reducing power, and there is much work left to be done in this area. With wider width processors and hyper speculation in the foreseeable future [9], pipeline gating methods will become even more essential for no-risk energy reduction in high performance processors.


ARM Cortex-A15 Block Diagram
Branch Prediction
Global History Buffer
MicroBTB (64-entry)
Branch Target Buffer (BTB) (256-entry)
Return Stack
L1 Instruction Cache
128 bits
Instruction Fetch
32-entry Loop Buffer
3-way Instruction Decode
3 Instructions
Register Rename
Dispatch
Issue (8-entry Queue per Issue port)
8 Instruction Issue
5 Stages
7 Stages
1 Stage
12 Stage In-Order Pipeline
Bus Interface Unit (BIU)
L2 Cache Control
Load/Store
Load/Store
Multiply Divide Cluster
Complex Cluster (NEON/FPU)
Complex Cluster (NEON/FPU)
Branch
Simple Cluster 1
Simple Cluster 0
4 Stages
4 Stages
2-10 Stages
Load-Store Unit
Store Buffer
L1 Data Cache
1 Load & 1 Store per cycle
ARM multiply & Integer divide
All NEON & FPU ops Quad-FMAC
Integer ALU & Shifter (includes v6-SIMD)
3-12 Stage Out-of-Order Pipeline
WriteBack (60 Entries)
Retirement Buffer

עד כאן
Dynamic branch prediction

נחזור ל-
Predicated execution

It was already discussed in previous lecture but here we add a few more details

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
- Guess the next fetch address (branch prediction)
- Employ delayed branching (branch delay slot)
- Do something else (fine-grained multithreading)
- Eliminate control-flow instructions (predicated execution)
- Fetch from both possible paths (if you know the addresses of both possible paths) (multipath execution)

Review: Predicate Combining (*not* Predicated Execution)

- Complex predicates are converted into multiple branches
  - if ((a == b) && (c < d) && (a > 5000)) { ... }
    - 3 conditional branches
- Problem: This increases the number of control dependencies
- Idea: Combine predicate operations to feed a single branch instruction
  - Predicates stored and operated on using condition registers
  - A single branch checks the value of the combined predicate

+ Fewer branches in code → fewer mipredictions/stalls

-- Possibly unnecessary work

-- If the first predicate is false, no need to compute other predicates

- Condition registers exist in IBM RS6000 and the POWER architecture

Predication (Predicated Execution)

- Idea: Convert control dependence to data dependence

- Simple example: Suppose we had a Conditional Move instruction…
  - CMOV condition, R1 ← R2
  - R1 = (condition == true) ? R2 : R1
  - Employed in most modern ISAs (x86, Alpha)
- Code example with branches vs. CMOVs

  if (a == 5) {b = 4;} else {b = 3;}

  CMPEQ condition, a, 5;

  CMOV condition, b ← 4;

  CMOV !condition, b ← 3;

Conditional Move Operations

- Very limited form of predicated execution

- CMOV R1 ← R2
  - R1 = (ConditionCode == true) ? R2 : R1
  - Employed in most modern ISAs (x86, Alpha)

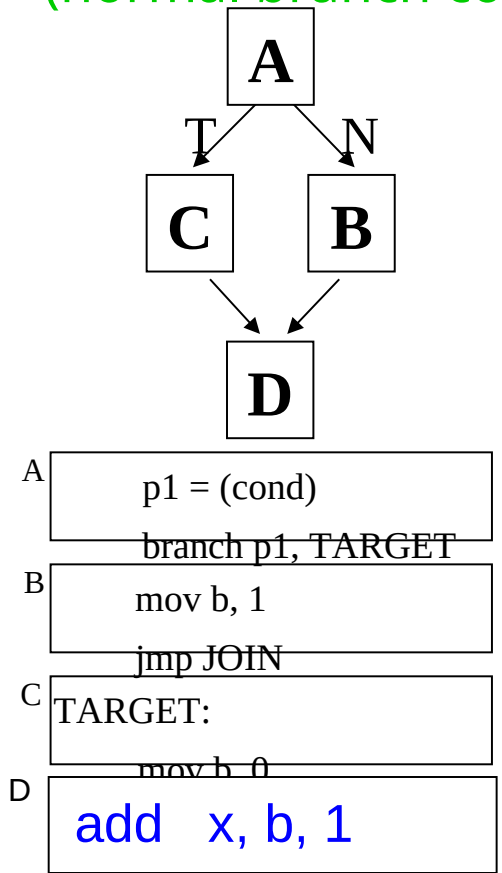
אנימציה

Predication (Predicated Execution)

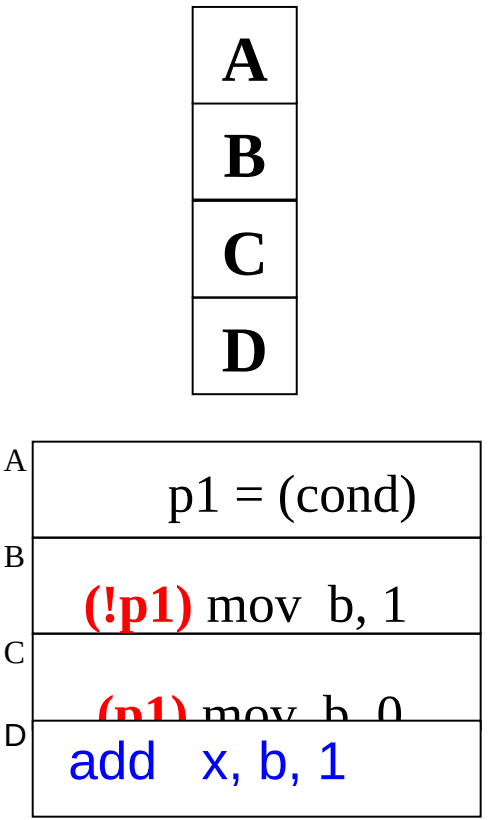
- Idea: Compiler converts control dependence into data dependence → branch is eliminated
  - Each instruction has a predicate bit set based on the predicate computation
  - Only instructions with TRUE predicates are committed (others turned into NOPs)

```
if (cond) {
    b = 0;
}
else {
    b = 1;
}
```

(normal branch code)



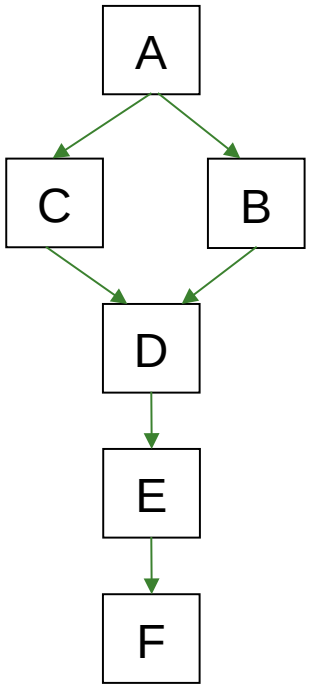
(predicated code)



Predicated Execution (II)

אנימציה

- Predicated execution can be high performance and energy-efficient



Predicated Execution

Fetch Decode Rename Schedule RegisterRead Execute

nop

Branch Prediction

Fetch Decode Rename Schedule RegisterRead Execute

Pipeline flush!!

Predicated Execution (III)

- Eliminates branches → enables straight line code (i.e., larger basic blocks in code)
- Advantages:

  + Eliminates mispredictions for hard-to-predict branches

    + No need for branch prediction for some branches

    + Good if misprediction cost > useless work due to predication

  + Enables code optimizations hindered (מופרע ע"י) by the control dependency

    + Can move instructions more freely within predicated code

- Disadvantages:

  -- Causes useless work for branches that are easy to predict

    -- Reduces performance if misprediction cost < useless work

    -- Adaptivity: Static predication is not adaptive to run-time branch behavior. Branch behavior changes based on input set, program phase, control-flow path.

  -- Additional hardware and ISA support

  -- Cannot eliminate all hard to predict branches

    -- Loop branches

Predicated Execution vs. Branch Prediction

+ Eliminates mispredictions for hard-to-predict branches
  + No need for branch prediction for some branches
  + Good if misprediction cost > useless work due to predication

-- Causes useless work for branches that are easy to predict
  -- Reduces performance if misprediction cost < useless work
  -- Adaptivity: Static predication is not adaptive to run-time branch behavior. Branch behavior changes based on input set, program phase, control-flow path.

Intel Itanium

Intel Itanium Architecture Software Developer's Manual

The Intel® Itanium® architecture is a unique combination of innovative features such as explicit parallelism, predication, speculation and more. The architecture is designed to be highly scalable to fill the ever increasing performance requirements of various server and workstation market segments. The Itanium architecture features a revolutionary 64-bit instruction set architecture (ISA) which applies a new processor architecture technology called EPIC, or Explicitly Parallel Instruction Computing. A key feature of the Itanium architecture is IA-32 instruction set compatibility.

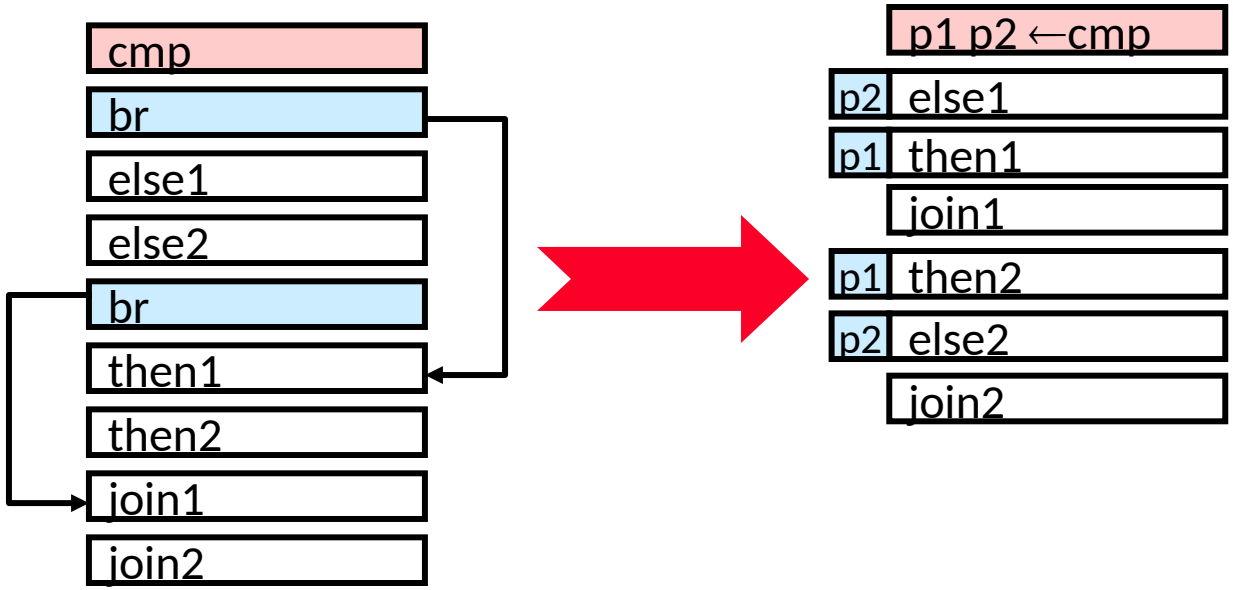
פרק 4 במדריך עוסק בפירוט מגוון האפשרויות של הטיפול
בהסתעפויות במנגנון ה-Predication

Predicated Execution in Intel Itanium

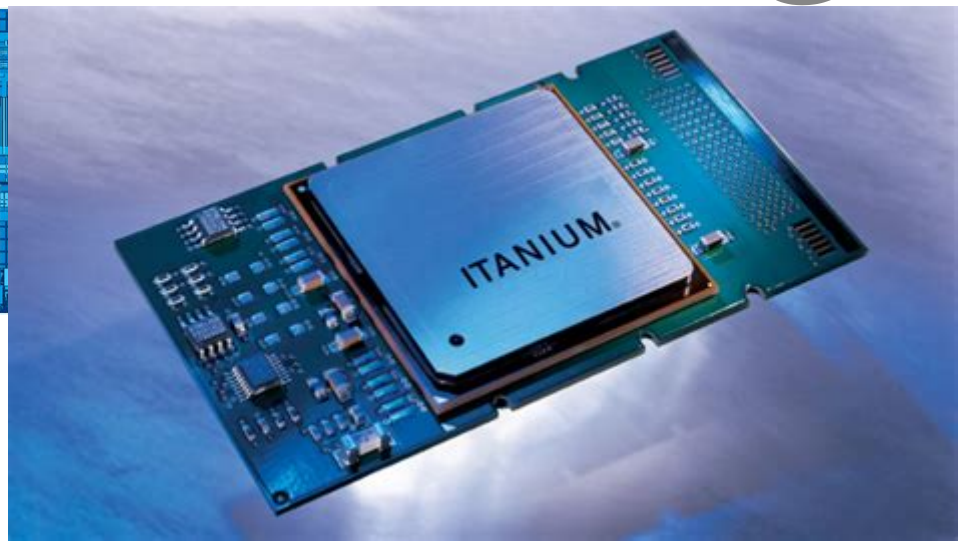
- Each instruction can be separately predicated
- 64 one-bit predicate registers

  each instruction carries a 6-bit predicate field

- An instruction is effectively a NOP if its predicate is false



Intel Itanium




```
if (a) {
    b = c + d;
}
if (e) {
    h = i + j;
}
```

```
cmp.ne p1,p2=a,r0       // p1 <- a!= 0
cmp.ne  p3,p4=e,r0 ;;   // p3 <- e != 0
(p1)add b=c,d           // If a!= 0 then add
(p3)sub h=i,j           // If e!= 0 then sub
```

p1, p3 הם התנאים

Reference:

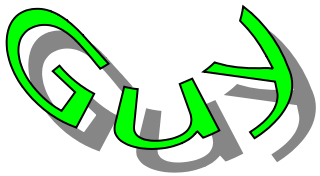
Intel® Itanium ® Architecture

Software Developer's Manual. Volume 1: Application Architecture

Revision 2.3, May 2010

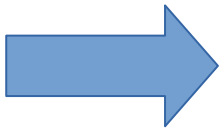
Section 2.5 - Predication

movn and movz in MIPS



```
int bar(int,int);

int foo(int a, int b){
    if (a <= b)
      a = 0;
    else
      b = 0;
    return bar(a, b);
}
```



```
foo:
        slt     $2,$5,$4
        movn    $5,$0,$2
        movz    $4,$0,$2
        j       bar
        nop
```

בצע הדגמה. לחץ על הקישור

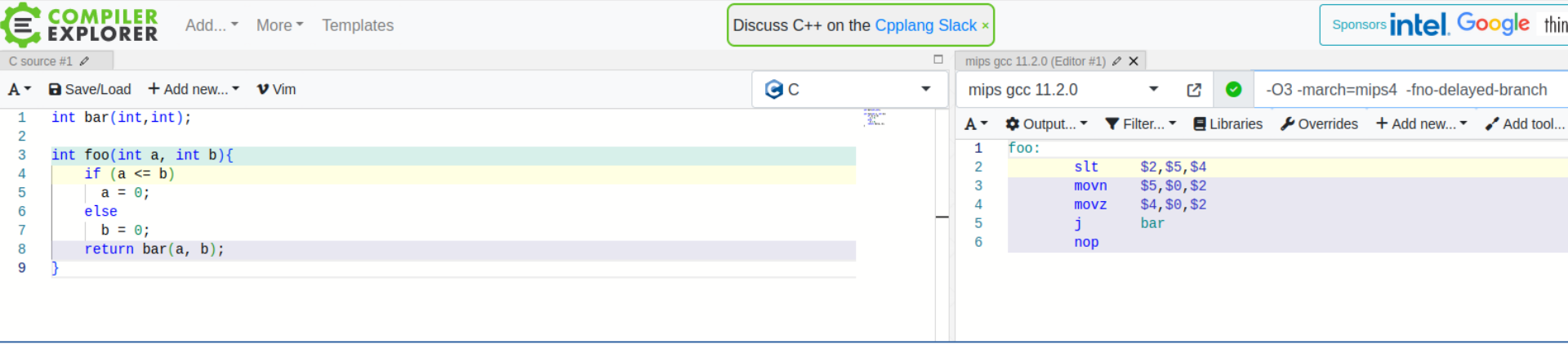
Version 1: -O0 -march=mips32 -fno-delayed-branch

Version 2: -O3 -march=mips32 -fno-delayed-branch

דוגמה דומה:

Show a demo:

:~/science/Teaching/CPU/lectures/10/code/movz




```
$ cat ./run_me.sh
#!/bin/sh
mips-linux-gnu-gcc -S -O0 -march=mips32 -fno-delayed-branch ./test1.c
mv test1.s test1_not_optimized.s
mips-linux-gnu-gcc -S -O3 -march=mips32 -fno-delayed-branch ./test1.c
mv test1.s ./test1_optimized.s
```

```
foo:
        .frame  $sp,0,$31               # vars= 0, regs= 0/0, args= 0, gp= 0
        .mask   0x00000000,0
        .fmask  0x00000000,0
        .set    noreorder
        .set    nomacro
        lui     $28,%hi(__gnu_local_gp)
        addiu   $28,$28,%lo(__gnu_local_gp)
        slt     $2,$5,$4
        lw      $25,%call16(bar)($28)
        movn    $5,$0,$2
        movz    $4,$0,$2
        .reloc  1f,R_MIPS_JALR,bar
1:      jr      $25
        nop
```

Conditional Execution in the ARM ISA

- Almost all ARM instructions can include an optional condition code.
- An instruction with a condition code is executed only if the condition code flags in the CPSR meet the specified condition.

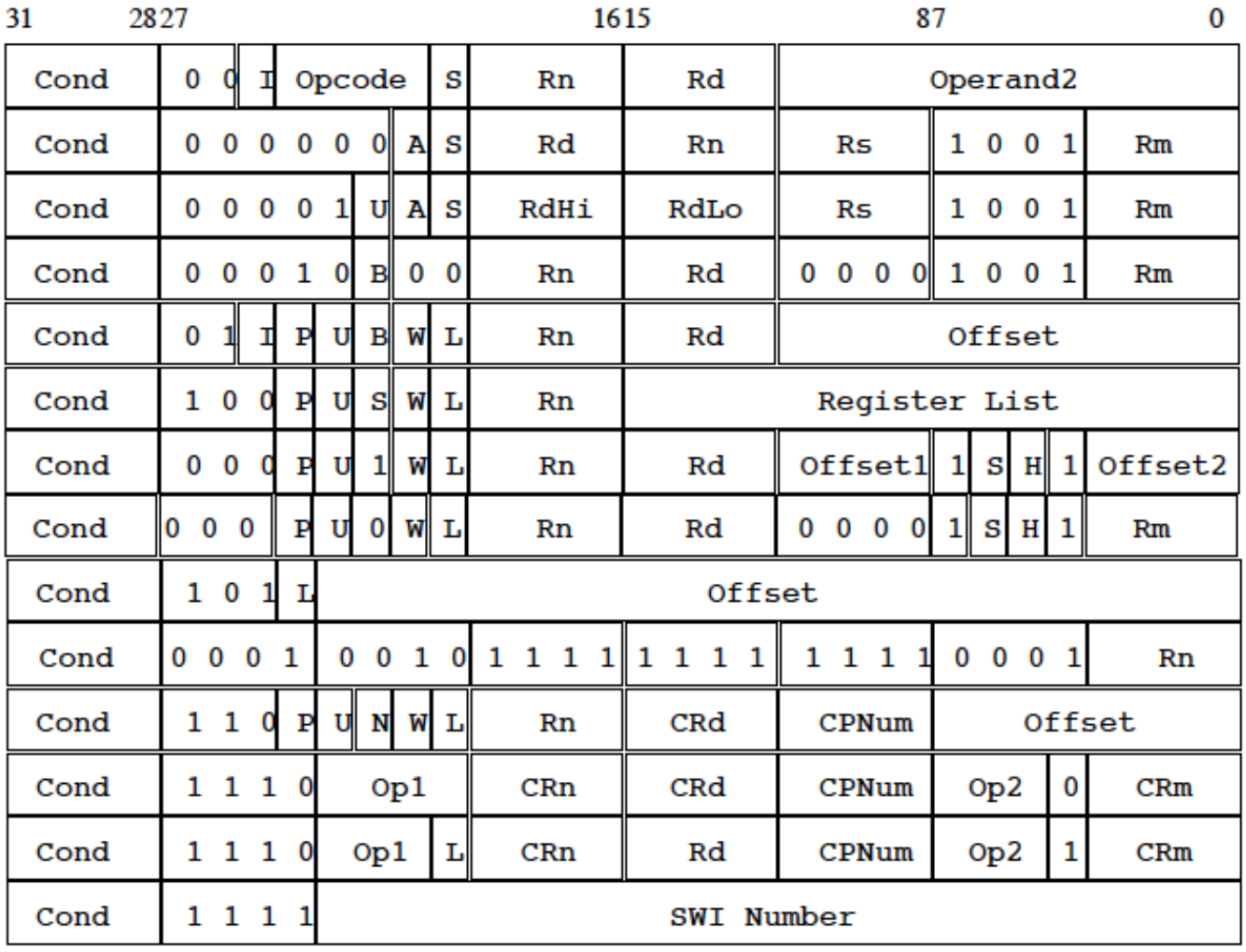
CPSR = Current Program Status Register

At any given moment, you have access to 16 registers (R0-R15) and the Current Program Status Register (CPSR).

(In User mode, a restricted form of the CPSR called the **Application Program Status Register (APSR)** is accessed instead.)

Source: ARM Cortex-A Series Programmer's Guide for ARMv7-A. https://developer.arm.com/documentation/den0013/d/ARM-Processor-Modes-and-Registers/Registers/Program-Status-Registers

Conditional Execution in ARM ISA



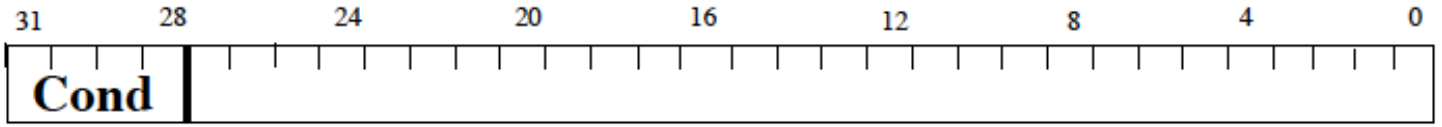
Instruction type

- Data processing / PSR Transfer
- Multiply
- Long Multiply **(v3M / v4 only)**
- Swap
- Load/Store Byte/Word
- Load/Store Multiple
- Halfword transfer : Immediate offset **(v4 only)**
- Halfword transfer: Register offset **(v4 only)**
- Branch
- Branch Exchange **(v4T only)**
- Coprocessor data transfer
- Coprocessor data operation
- Coprocessor register transfer
- Software interrupt

4 bit

4 הביטים של ה- COND הם חלק מקידוד ההוראות שעושות שימוש ברגיסטר הסטטוס

Conditional Execution in ARM ISA



0000 = EQ - Z set (equal)

0001 = NE - Z clear (not equal)

0010 = HS / CS - C set (unsigned higher or same)

0011 = LO / CC - C clear (unsigned lower)

0100 = MI -N set (negative)

0101 = PL - N clear (positive or zero)

0110 = VS - V set (overflow)

0111 = VC - V clear (no overflow)

1000 = HI - C set and Z clear (unsigned higher)

1001 = LS - C clear or Z (set unsigned lower or same)

1010 = GE - N set and V set, or N clear and V clear (>or =)

1011 = LT - N set and V clear, or N clear and V set (>)

1100 = GT - Z clear, and either N set and V set, or N clear and V set (>)

1101 = LE - Z set, or N set and V clear,or N clear and V set (<, or =)

1110 = AL - always

1111 = NV - reserved.

Conditional Execution in ARM ISA

* **To execute an instruction conditionally, simply postfix it with the appropriate condition:**
  - For example an add instruction takes the form:
    - `ADD r0,r1,r2 ; r0 = r1 + r2 (ADDAL)`

    ADD Always
    ללא תנאי
  - To execute this only if the zero flag is set:
    - `ADDEQ r0,r1,r2 ; If zero flag set then...`
      `; ... r0 = r1 + r2`

    in the CPSR
* **By default, data processing operations do not affect the condition flags (apart from the comparisons where this is the only effect). To cause the condition flags to be updated, the S bit of the instruction needs to be set by postfixing the instruction (and any condition code) with an "S".**
  - For example to add two numbers and set the condition flags:
    - `ADDS r0,r1,r2 ; r0 = r1 + r2`
      `; ... and set flags` in the CPSR

Reference (Guy):
https://developer.arm.com/documentation/ddi0344/b/ch16s02s01

ADDAL – ADD Always regardless of the condition flag

ADDEQ (Add if Equal)

ADDNE (Add if Not Equal)

ADDGT (Add if Greater Than)

ADDVS (Add if Overflow Set)

ADDS – S suffux: this means that the instruction will update the condition flags (N,Z,C,V) in the CPSR according to the result of the addition.

N – Negative

Z – Zero

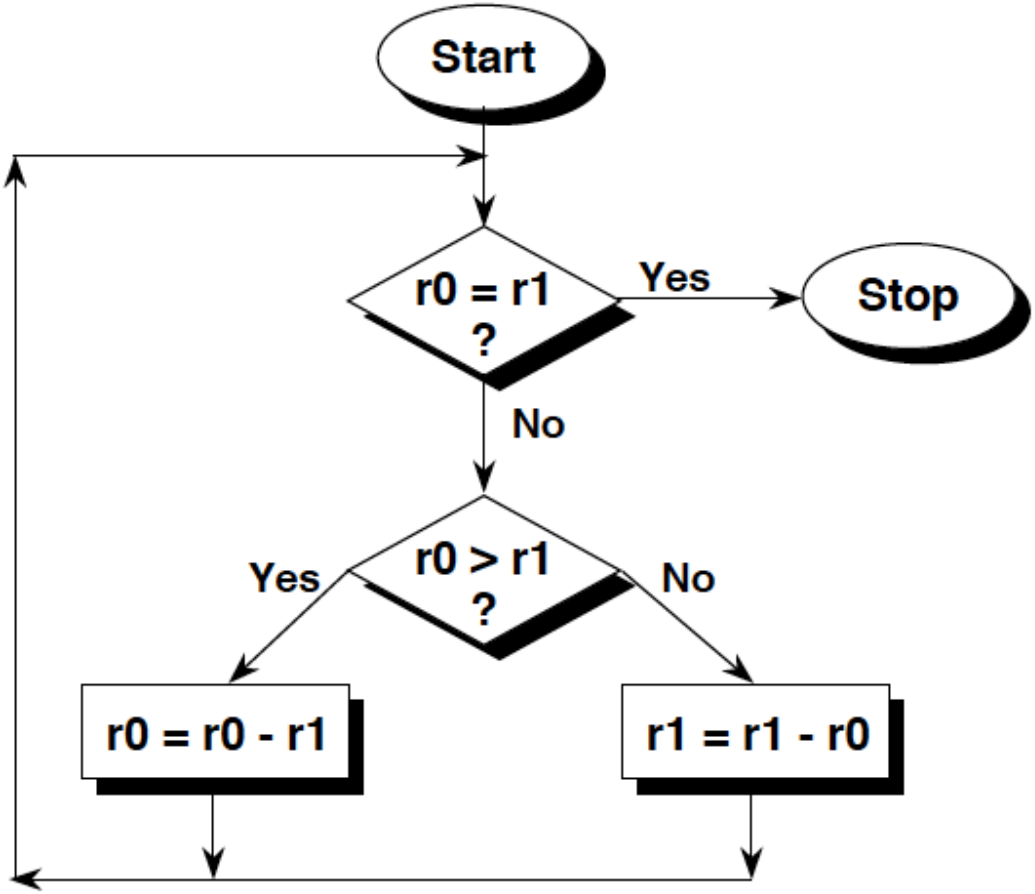
C – Carry

V - Overflow

השוואה בין מוד רגיל ובין מוד המעדכן את הסטטוס רגיסטר

Feature	`ADD` (e.g., `ADD R0, R1, R2`)	`ADDS` (e.g., `ADDS R0, R1, R2`)
Operation	Adds two operands	Adds two operands
Result Storage	Stores result in destination register	Stores result in destination register
Condition Flags	**Does NOT** update N, Z, C, V flags	**UPDATES** N, Z, C, V flags
Purpose	Simple arithmetic, no immediate conditional branch needed	Arithmetic where the result needs to be tested for conditional branching or other flag-dependent operations

Conditional Execution in ARM ISA



* **Convert the GCD algorithm given in this flowchart into**
  1) "Normal" assembler, where only branches can be conditional.
  2) ARM assembler, where all instructions are conditional, thus improving code density.

* **The only instructions you need are CMP, B and SUB.**

הקוד בשקף הבא.

GCD = Greatest Common Divisor

The greatest common divisor (GCD) of integers a and b, at least one of which is nonzero, is the greatest positive integer d such that d is a divisor of both a and b; that is, there are integers e and f such that a = de and b = df, and d is the largest such integer. The GCD of a and b is generally denoted gcd(a, b).[8]

Conditional Execution in ARM ISA

"Normal" Assembler

```
gcd     cmp r0, r1        ;reached the end?
        beq stop
        blt less          ;if r0 > r1
        sub r0, r0, r1    ;subtract r1 from r0
        bal gcd
less    sub r1, r1, r0    ;subtract r0 from r1
        bal gcd
stop
```

ARM Conditional Assembler

```
gcd     cmp   r0, r1      ;if r0 > r1
        subgt r0, r0, r1  ;subtract r1 from r0
        sublt r1, r1, r0  ;else subtract r0 from r1
        bne   gcd         ;reached the end?
```

פעולות השוואה מפעילות אוטומטית
את ה-conditional bits

How to Handle Control Dependences

אנימציה

- 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
- Guess the next fetch address (branch prediction)
- Employ delayed branching (branch delay slot)
- Do something else (fine-grained multithreading)
- Eliminate control-flow instructions (predicated execution)
- Fetch from both possible paths (if you know the addresses of both possible paths) (multipath execution)

Review of Last Few Lectures

- Control dependence handling in pipelined machines
  - Delayed branching
  - Fine-grained multi-threading
  - Branch prediction
    - Compile time (static)
      - Always NT, Always T, Backward T Forward NT, Profile based
    - Run time (dynamic)
      - Last time predictor
      - Hysteresis: 2BC predictor
      - Global branch correlation → Two-level global predictor
      - Local branch correlation → Two-level local predictor
      - Hybrid branch predictors
  - Predicated execution
  - Multi-path execution – מאוד בקצרה
  - Return address stack & Indirect branch prediction

עד כאן מצגת זו