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

Based on slides by Prof. Onur Mutlu

Carnegie Mellon University

Spring 2015

Dr. Guy Tel-Zur

Control Dependence Handling

Agenda for Today & Next Few Lectures

- Single-cycle Microarchitectures
- Multi-cycle and Microprogrammed Microarchitectures
- Pipelining
- Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, ...
- Out-of-Order Execution
- Issues in OoO Execution: Load-Store Handling, ...

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.
- Kessler, "The Alpha 21264 Microprocessor," IEEE Micro 1999.

Readings for Next Few Lectures

כתובת להורדה:

ftp://ftp.cs.wisc.edu/sohi/papers/1995/ieee-proc.superscalar.pdf

The Microarchitecture of Superscalar Processors

James E. Smith
Department of Electrical and Computer Engineering
1415 Johnson Drive
Madison, WI 53706
ph: (608)-265-5737
fax:(608)-262-1267
email: jes@ece.wisc.edu

Gurindar S. Sohi
Computer Sciences Department
1210 W. Dayton
Madison, WI 53706
ph: (608)-262-7985
email: sohi@cs.wisc.edu

Aug. 20, 1995

James Smith

Gurindar S. Sohi

תקציר המאמר

Abstract - ההדגשות הן שלי

Superscalar processing is the latest in a long series of innovations aimed at producing ever-faster microprocessors.

By exploiting **instruction-level parallelism**, superscalar processors are capable of executing more than one instruction in a clock cycle. **This paper discusses the microarchitecture of superscalar processors**. We begin with a discussion of the general problem solved by superscalar processors: converting an ostensibly sequential program into a more parallel one. The principles underlying this process, and the constraints that must be met, are discussed.

The paper then provides a description of the specific implementation techniques used in the important phases of superscalar processing. The major phases include: i) instruction fetching and conditional branch processing, ii) the determination of data dependences involving register values, iii) the initiation, or issuing, of instructions for parallel execution, iv) the communication of data values through memory via loads and stores, and v) committing the process state in correct order so that precise interrupts can be supported.

Examples of recent superscalar microprocessors,

the MIPS R10000, the DEC 21164, and the AMD K5 are used to illustrate a variety of superscalar methods.

תרגום התקציר

עיבוד סופר-סקלרי הוא האחרון בסדרה ארוכה של חידושים שמטרתם לייצר מיקרו-מעבדים מהירים יותר ויותר.

על ידי ניצול מקביליות ברמת הוראות, מעבדים סופר-סקלריים מסוגלים לבצע יותר מהוראה אחת במחזור שעון. מאמר זה דן במיקרו-ארכיטקטורה של מעבדים סופר-סקלריים. אנו מתחילים בדיון בבעיה הכללית הנפתרת על ידי מעבדים סופר-סקלריים: המרת תוכנית לכאורה רציפה לתוכנית מקבילה יותר. העקרונות העומדים בבסיס תהליך זה והאילוצים שיש לעמוד בהם, נדונים.

לאחר מכן, המאמר מספק תיאור של טכניקות היישום הספציפיות המשמשות בשלבים החשובים של עיבוד סופר-סקלרי. השלבים העיקריים כוללים: i) אחזור הוראות ועיבוד ענפים מותנה, ii) קביעת תלות נתונים הכוללת ערכי אוגר, iii) ייזום או הנפקה של הוראות לביצוע מקביל, iv) תקשורת ערכי נתונים דרך זיכרון באמצעות טעינות ומאחסנים, ו-v) ביצוע מצב התהליך בסדר הנכון כך שניתן יהיה לתמוך בפסיקות מדויקות.

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).

Implementing Precise Interrupts in Pipelined Processors

JAMES E. SMITH, MEMBER, IEEE, AND ANDREW R. PLESZKUN, MEMBER, IEEE

Abstract—**This paper describes and evaluates solutions to the precise interrupt problem in pipelined processors. An interrupt is *precise* if the saved process state corresponds with a sequential model of program execution where one instruction completes before the next begins. In a pipelined processor, precise interrupts are difficult to implement because an instruction may be initiated before its predecessors have completed.**

The precise interrupt problem is described, and five solutions are discussed in detail. The first solution forces instructions to complete and modify the process state in architectural order. The other four solutions allow instructions to complete in any order, but additional hardware is used so that a precise state can be restored when an interrupt occurs. All the methods are discussed in the context of a parallel pipeline structure. Simulation results based on the CRAY-1S scalar architecture are used to show that the first solution results in a performance degradation of at least 16 percent. The remaining four solutions offer better performance, and three of them result in as little as a 3 percent performance loss. Several extensions, including vector architectures, virtual memory, and linear pipeline structures, are briefly discussed.

model, then the interrupt is *precise*. To be more specific, the saved state should reflect the following conditions.

1) All instructions preceding the instruction indicated by the saved program counter have been executed and have modified the process state correctly.

2) All instructions following the instruction indicated by the saved program counter are unexecuted and have not modified the process state.

3) If the interrupt is caused by an exception condition raised by an instruction in the program, the saved program counter points to the interrupted instruction. The interrupted instruction may or may not have been executed, depending on the definition of the architecture and the cause of the interrupt. Whichever is the case, the interrupted instruction has either completed, or has not started execution.

If the saved process state is inconsistent with the sequential architectural model and does not satisfy the above conditions, then the interrupt is *imprecise*.

Recap of Last Lecture

- Data Dependence Handling
  - Data Forwarding/Bypassing
  - In-depth Implementation
    - MIPS Pipelining
    - Register dependence analysis
    - Stalling
    - Performance analysis with and without forwarding
  - Questions to Ponder
    - HW vs. SW handling of data dependences
    - Static versus dynamic scheduling
    - What makes compiler based instruction scheduling difficult?
    - Profiling (representative input sets needed; dynamic adaptation difficult) – **demo**: see next slide

  Introduction to static instruction scheduling (e.g., fix-up code)

Control Dependence Handling

Review: Control Dependence

- Question: What should the fetch PC be in the next cycle?
- Answer: The address of the next instruction
  - All instructions are control dependent on previous ones. Why?
- If the fetched instruction is a **non-control-flow** instruction:
  - Next Fetch PC is the address of the next-sequential instruction
  - Easy to determine if we know the size of the fetched instruction
- If the instruction that is fetched is a control-flow instruction:
  - How do we determine the next Fetch PC?
- In fact, how do we even know whether or not the fetched instruction is a control-flow instruction?

Branch Types

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

Different branch types are being handled differently

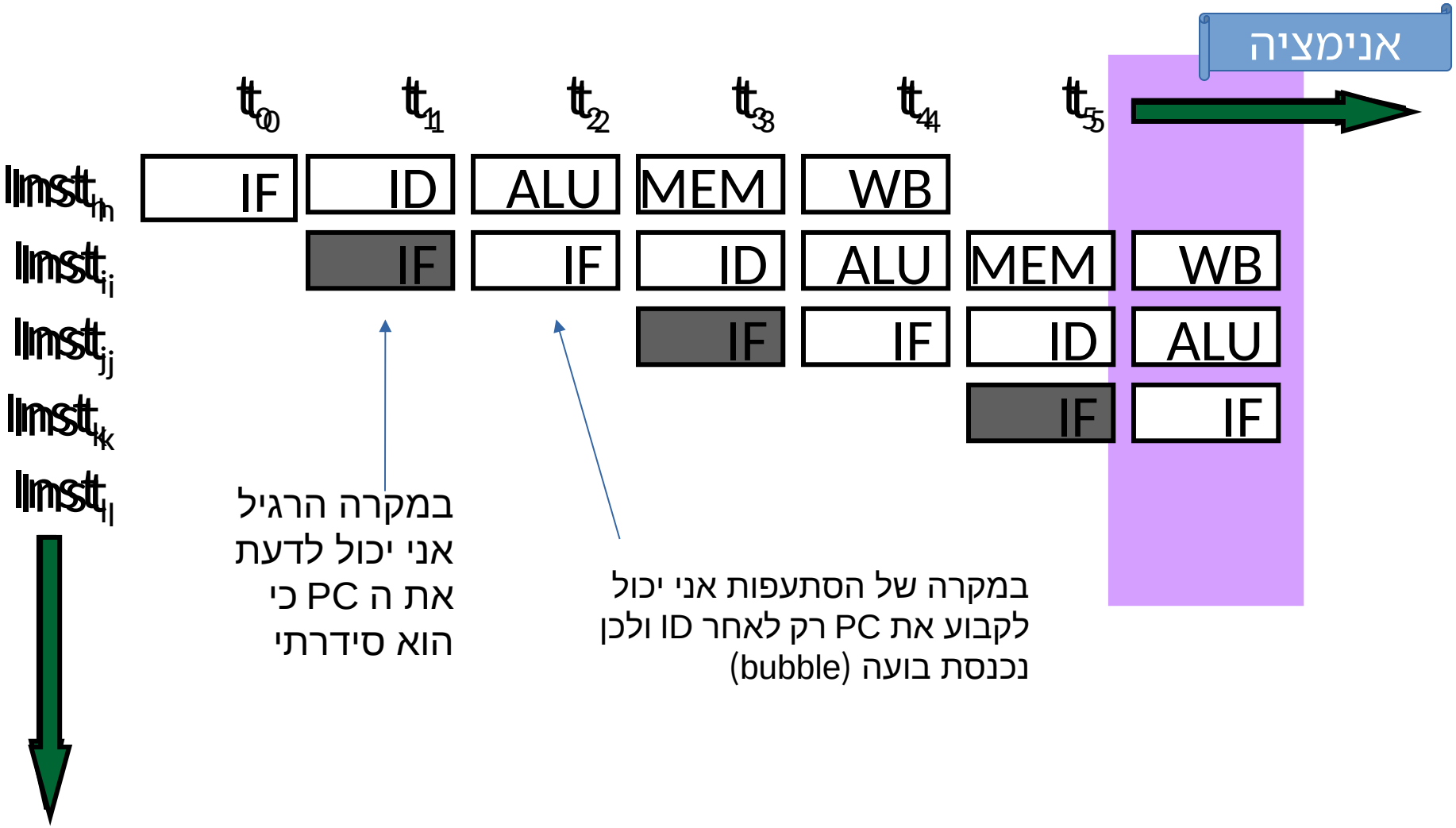
How to Handle Control Dependences

- Critical to keep the pipeline full with correct sequence of dynamic instructions.
- Potential solutions if the instruction is a control-flow instruction: שישה פתרונות אפשריים:
- Stall the pipeline until we know the next fetch address — פתרון לא טוב
- Guess the next fetch address (branch prediction) — H/W
- Employ delayed branching (branch delay slot) — S/W
- 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)

נתאר בקצרה את הפתרונות 1, 3-6 ובהמשך נתמקד ב- 2

Stall Fetch Until Next PC is Available: Good Idea?

האפשרות הראשונה מהשקף הקודם. בשקף נראית סידרה של הסתעפויות בלתי מותנות.



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

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) — פתרון לא טוב
- Employ delayed branching (branch delay slot)
- Do something else (fine-grained multithreading) } נדבר על כך בפרק הבא
- Eliminate control-flow instructions (predicated execution)
- Fetch from both possible paths (if you know the addresses of both possible paths) (multipath execution)

Doing Better than Stalling Fetch …

- Rather than waiting for true-dependence on PC to resolve, just guess nextPC = PC+4 to keep fetching every cycle

תחזית מהסוג הכי בסיסי: להניח תמיד PC+4

Is this a good guess?

What do you lose if you guessed incorrectly?

- ~20% of the instruction mix is control flow
  - ~50 % of "forward" control flow (i.e., if-then-else) is taken
  - ~90% of "backward" control flow (i.e., loop back) is taken

Overall, typically ~70% taken and ~30% not taken

[Lee and Smith, 1984]

Not Take ==>
next PC = PC+4

- Expect "nextPC = PC+4" ~86% of the time, but what about the remaining 14%?

80% + 20%*30% = 86%, 100%-86% = 14% הסבר

Guessing NextPC = PC + 4

- Always predict the next sequential instruction is the next instruction to be executed
- This is a form of next fetch address prediction (and branch prediction)
- How can you make this more effective?
- Idea: Maximize the chances that the next sequential instruction is the next instruction to be executed
  - Software: Lay out the control flow graph such that the "likely next instruction" is on the not-taken path of a branch
    - **Profile** guided code positioning → Pettis & Hansen, PLDI 1990.

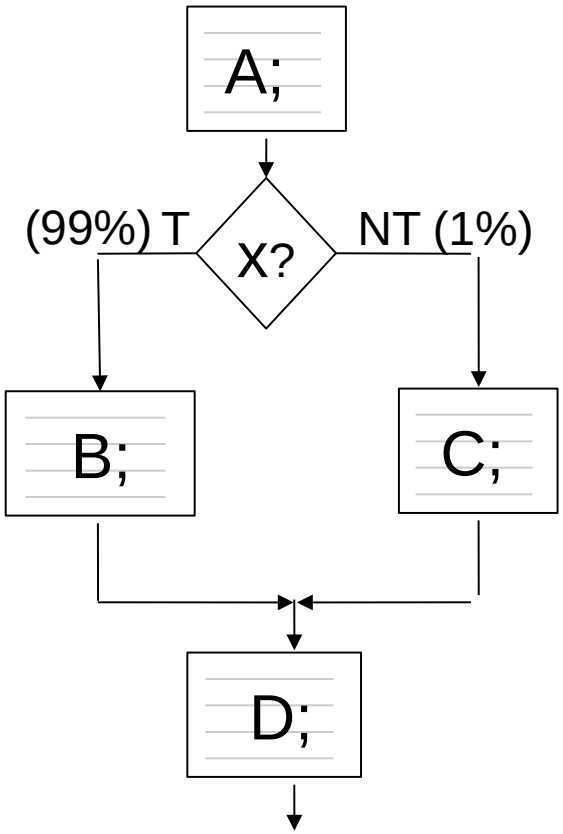
השקף הבא

  - Hardware: ??? (how can you do this in hardware…)
    - **Cache traces** of executed instructions → Trace cache

Adjust code according to profiling - SW

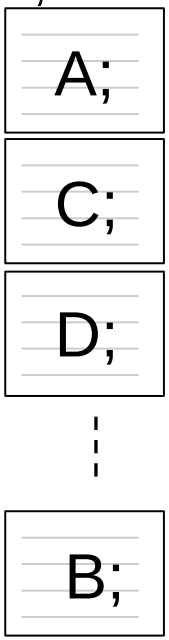
אנימציה

Compiler profile:



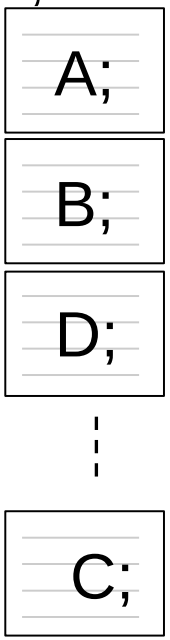
Option I:

```
A;
if (x)  { B }
else  {C};
D;
```



Option II:

```
A;
if (!x)  { C }
else  {B};
D;
```



ארגון הקוד הסידרתי כך שיהיה רציף בסבירות גבוהה

Option II is better since in 99% of the cases NT is PC+4:

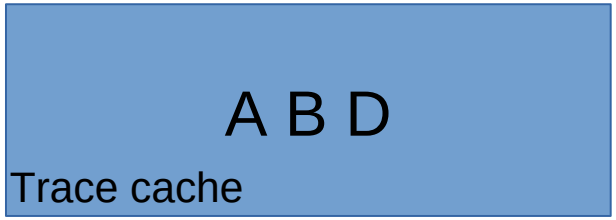
Trace Cache (adjust by HW)

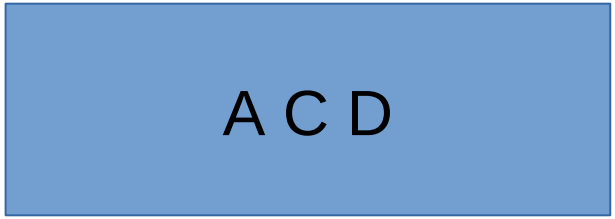
נניח 5 הוראות בכל בלוק

סה"כ 15 הוראות

A B D

5 5 5





המטמון שומר את העקבות - traces. מה שמבוצע סידרתית גם יכתב סידרתית גם במטמון.

נשתמש בו לניחוש (prediction) של הפקודה הבאה (הערך הבא ב-PC).

צריך עוד שדות חוץ מה-PC הבא (וגם לטפל בטעות).

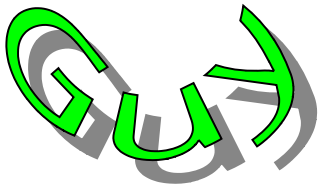
אם יחול שינוי בהמשך התכנית, המטמון יכול להסתגל ולשנות את תוכנו. (זה גם חלק מהטיפול בטעות)

בשקף הקודם ראינו פתרון בתוכנה שמבוצע ע"י הקומפיילר הכותב לזיכרון. עתה הפתרון הוא בחומרה ונכתב במטמון.

ה- Trace cache יכול להסתגל לשינויים - לפיכך זהו פתרון דינמי, תוצאתי.

מתבצע כך ב**פנטיום 4**.

Trace Cache - read



אנא ראו, ווקיפדיה: https://en.wikipedia.org/wiki/Trace_cache

Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching

Eric Rotenberg
Computer Science Dept.
Univ. of Wisconsin - Madison
ericro@cs.wisc.edu

Steve Bennett
Intel Corporation
sbennett@ichips.intel.com

James E. Smith
Dept. of Elec. and Comp. Engr.
Univ. of Wisconsin - Madison
jes@ece.wisc.edu

Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching

by Rotenberg, Bemmett & Smith

http://www.eecs.harvard.edu/cs146-246/micro.trace-cache.pdf

Abstract

As the issue width of superscalar processors is increased, instruction fetch bandwidth requirements will also increase. It will become necessary to fetch multiple basic blocks per cycle. Conventional instruction caches hinder this effort because long instruction sequences are not always in contiguous cache locations.

We propose supplementing the conventional instruction cache with a trace cache. This structure caches traces of the dynamic instruction stream, so instructions that are otherwise noncontiguous appear contiguous. For the Instruction Benchmark Suite (IBS) and SPEC92 integer benchmarks, a 4 kilobyte trace cache improves performance on average by 28% over conventional sequential fetching. Further, it is shown that the trace cache's efficient, low latency approach enables it to outperform more complex mechanisms that work solely out of the instruction cache.

1. Introduction

High performance superscalar processor organizations divide naturally into an instruction fetch mechanism and an instruction execution mechanism (Figure 1). The fetch and execution mechanisms are separated by instruction issue buffer(s), for example queues, reservation stations, etc. Conceptually, the instruction fetch mechanism acts as a "producer" which fetches, decodes, and places instructions into the buffer. The instruction execution engine is the "consumer" which removes instructions from the buffer and executes them, subject to data dependence and resource constraints. Control dependences (branches and jumps) provide a feedback mechanism between the producer and consumer.

Processors having this organization employ aggressive techniques to exploit instruction-level parallelism. Wide dispatch and issue paths place an upper bound on peak instruction throughput. Large issue buffers are used to maintain a *window* of instructions necessary for detecting parallelism, and a large pool of physical registers provides destinations for all the in-flight instructions issued from the window. To enable concurrent execution of instructions, the execution engine is composed of many parallel functional units. The fetch engine speculates past multiple branches in order to supply a continuous instruction stream to the window.

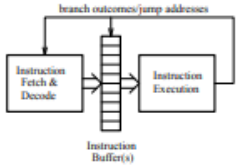
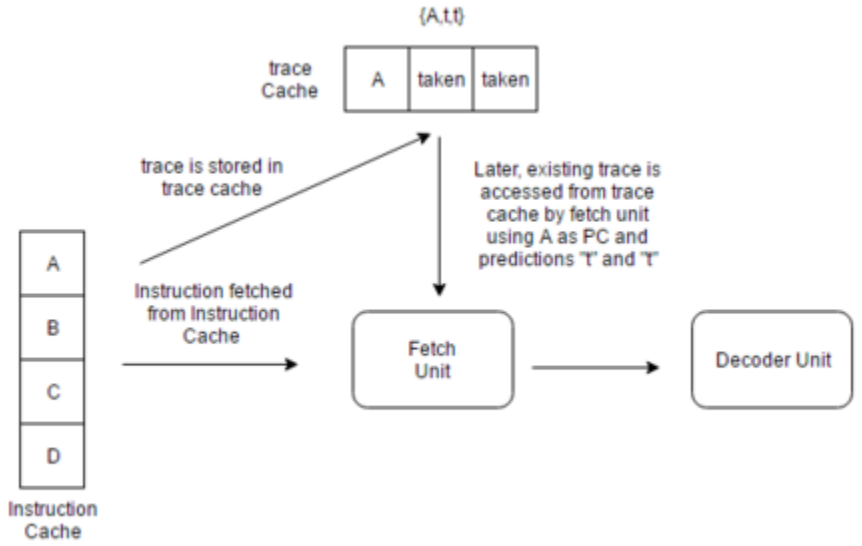


Figure 1. Decoupled fetch/execute engines.

The trend in superscalar design is to increase the scale of these techniques: wider dispatch/issue, larger windows, more physical registers, more functional units, and deeper speculation. To maintain this trend, it is important to balance all parts of the processor – any bottlenecks diminish the benefit of aggressive ILP techniques.

In this paper, we are concerned with instruction fetch bandwidth becoming a performance bottleneck. Instruction fetch performance depends on a number of factors. Instruction cache hit rate and branch prediction accuracy have long been recognized as important problems in fetch performance and are well-researched areas. In this paper, we are interested in additional factors that are only now emerging as processor issue rates exceed four instructions per cycle:

- *branch throughput* – If only one conditional branch is predicted per cycle, then the window can grow at the rate of only one basic block per cycle. Predicting multiple branches per cycle allows the overall instruction throughput to be correspondingly higher.
- *noncontiguous instruction alignment* – Because of branches and jumps, instructions to be fetched during any given cycle may not be in contiguous cache locations. Hence, there must be adequate paths and logic available to fetch and align noncontiguous basic blocks and pass them up the pipeline. That is, it is not enough



Pentium 4

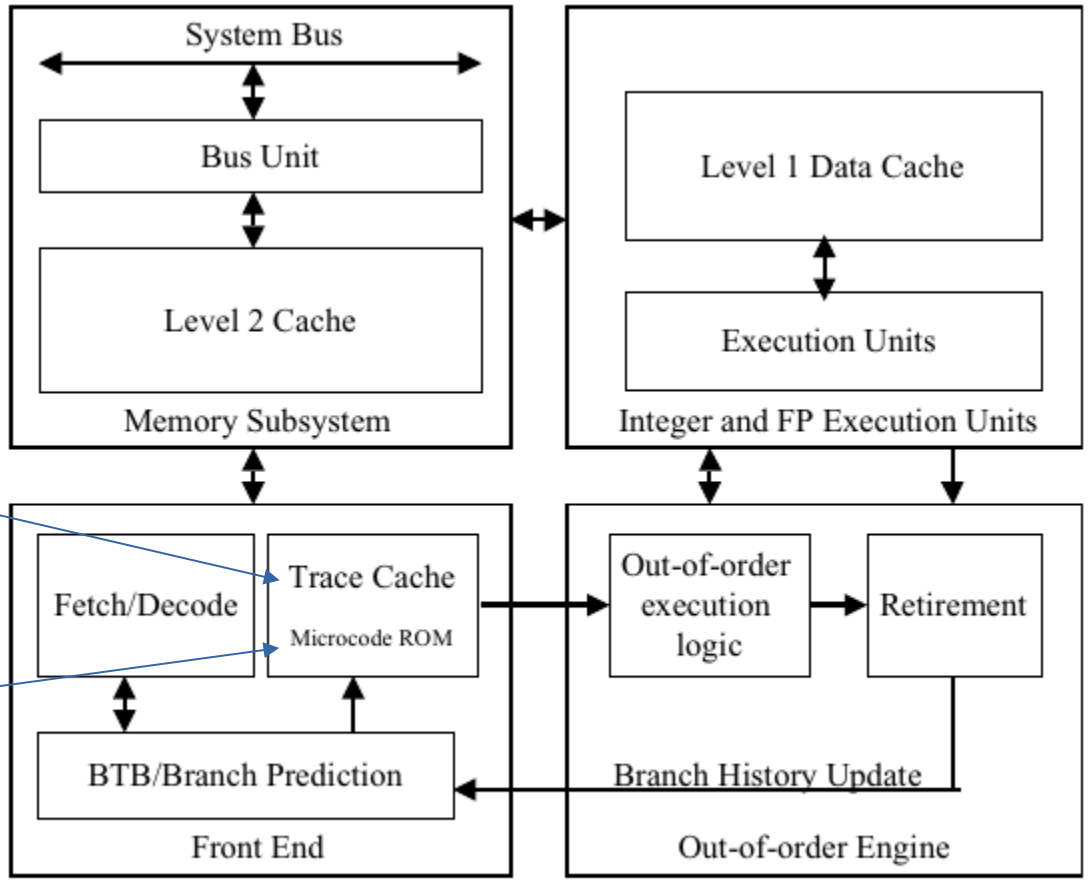


Figure 1: Basic block diagram

Source: Hinton et al, "The Microarchitecture of the Pentium 4 Processor", Intel Technology Journal Q1, 2001

מתוך המאמר

The Microarchitecture of the Pentium® 4 Processor

Glenn Hinton, Desktop Platforms Group, Intel Corp.
Dave Sager, Desktop Platforms Group, Intel Corp.
Mike Upton, Desktop Platforms Group, Intel Corp.
Darrell Boggs, Desktop Platforms Group, Intel Corp.
Doug Carmean, Desktop Platforms Group, Intel Corp.
Alan Kyker, Desktop Platforms Group, Intel Corp.
Patrice Roussel, Desktop Platforms Group, Intel Corp.

Index words: Pentium® 4 processor, NetBurst™ microarchitecture, Trace Cache, double-pumped ALU, deep pipelining

ABSTRACT

This paper describes the Intel® NetBurst™ microarchitecture of Intel's new flagship Pentium® 4 processor. This microarchitecture is the basis of a new family of processors from Intel starting with the Pentium 4 processor. The Pentium 4 processor provides a substantial performance gain for many key application areas where the end user can truly appreciate the

provides an in-depth examination of the features and functions of the Intel NetBurst microarchitecture.

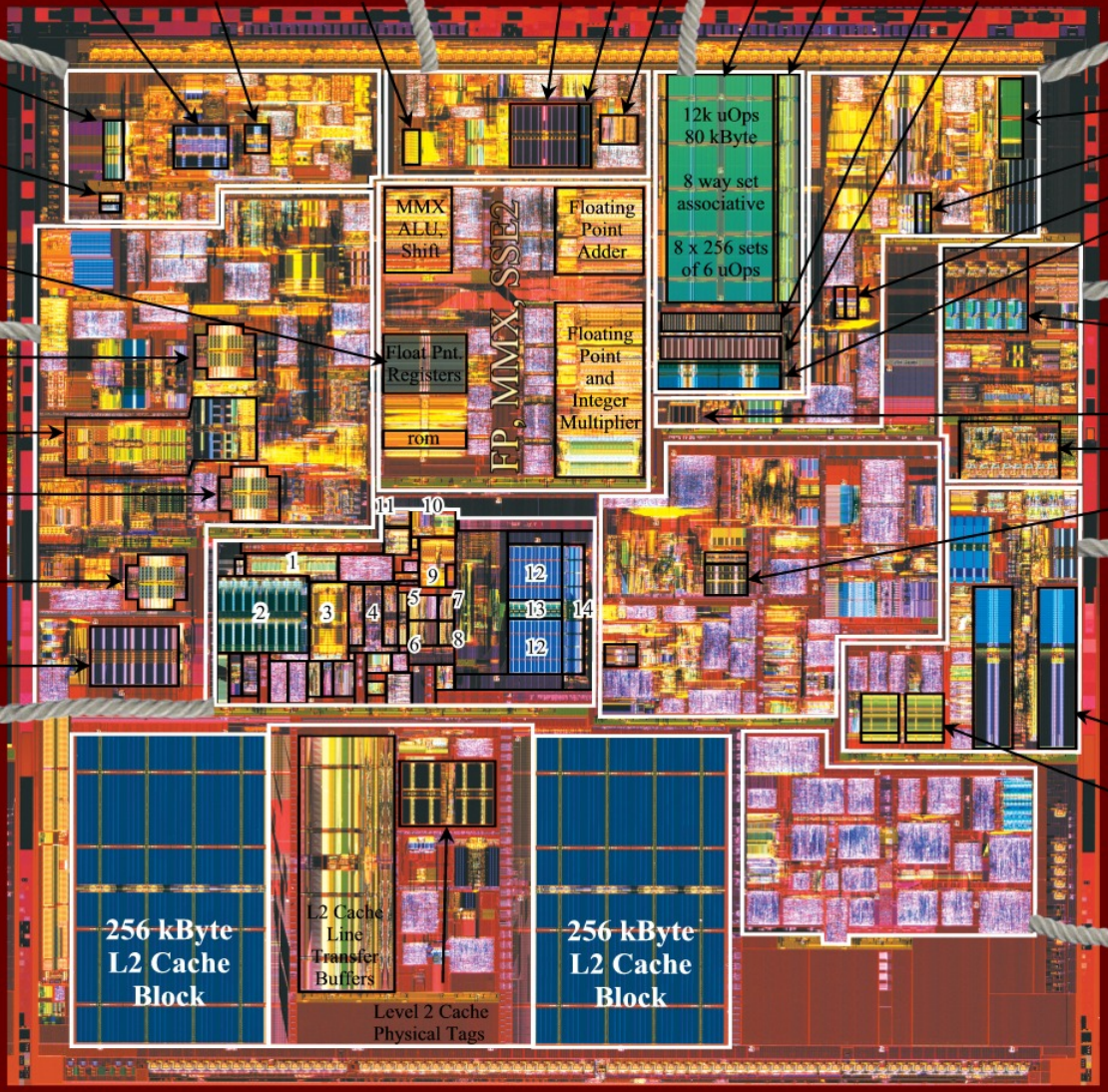
The Pentium 4 processor is designed to deliver performance across applications where end users can truly appreciate and experience its performance. For example, it allows a much better user experience in areas such as Internet audio and streaming video, image processing, video content creation, speech recognition, 3D

These traces consist of uops running sequentially down the predicted path of the IA-32 program execution. This allows the target of a branch to be included in the same trace cache line as the branch itself even if the branch and its target instructions are thousands of bytes apart in the program.

Note to self: /home/telzur/science/Teaching/CPU/lectures/09/extra_reading/Pentium4.pdf

Credi: http://www.chip-architect.org/news/Northwood_130nm_die_text_1600x1200.jpg

Intel Pentium 4 Northwood



Buffer Allocation & Register Rename

Instruction Queue (for less critical fields of the uOps)

General Instruction Address Queue & Memory Instruction Address Queue (queues register entries and latency fields of the uOps for scheduling)

Floating Point, MMX, SSE2 Renamed Register File 128 entries of 128 bit.

uOp Schedulers

FP Move Scheduler: (8x8 dependency matrix)

Parallel (Matrix) Scheduler for the two double pumped ALU's

General Floating Point and Slow Integer Scheduler: (8x8 dependency matrix)

Load / Store uOp Scheduler: (8x8 dependency matrix)

Load / Store Linear Address Collision History Table

Integer Execution Core

(1) uOp Dispatch unit & Replay Buffer Dispatches up to 6 uOps / cycle
(2) Integer Renamed Register File 128 entries of 32 bit + 6 status flags 12 read ports and six write ports
(3) Databus switch & Bypasses to and from the Integer Register File.
(4) Flags, Write Back
(5) Double Pumped ALU 0
(6) Double Pumped ALU 1
(7) Load Address Generator Unit
(8) Store Address Generator Unit
(9) Load Buffer (48 entries)
(10) Store Buffer (24 entries)

Execution Pipeline Start

Instruction Trace Cache

Trace Cache Access, next Address Predict

Trace Cache Branch Prediction Table (BTB), 512 entries.

Return Stacks (2x16 entries)

Trace Cache next IP's (2x)

Miscellaneous Tag Data

Instruction Decoder

Up to 4 decoded uOps/cycle out. (from max. one x86 instr/cycle) Instructions with more than four are handled by Micro Sequencer

Trace Cache LRU bits

Raw Instruction Bytes in

Data TLB, 64 entry fully associative, between threads dual ported (for loads and stores)

Instruction Fetch from L2 cache and Branch Prediction

Front End Branch Prediction Tables (BTB), shared, 4096 entries in total

Instruction TLB's 2x64 entry, fully associative for 4k and 4M pages. In: Virtual address [31:12] Out: Physical address [35:12] + 2 page level bits

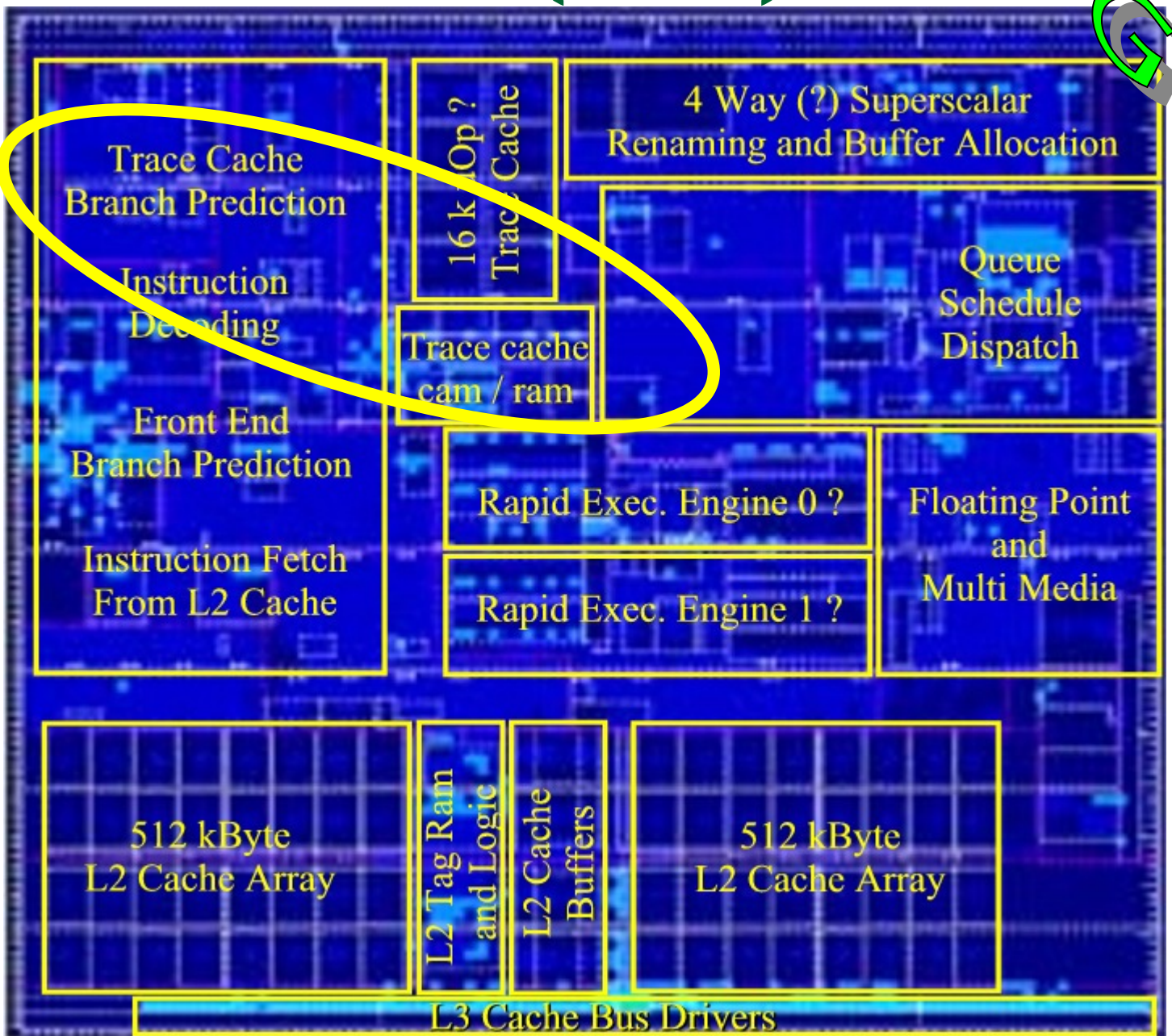
Front Side Bus Interface, 400..800 MHz

(11) ROB Reorder Buffer 3x42 entries
(12) 8 kByte Level 1 Data cache four way set associative. 1R/1W
(13) Summed Address Index decode and Way Predict
(14) Cache Line Read / Write Transferbuffers and 256 bit wide bus to and from L2 cache

April 19, 2003 www.chip-architect.com

Intel's Prescott (2003)

Credit:
http://www.chip-architect.org/news/2003_03_06_Looking_at_Intels_Prescott.html#(1)%20%20Instruction%20Trace%20Cache%20Extended



Guessing NextPC = PC + 4 , more!

- How else can you make this more effective?

- Idea: Get rid of control flow instructions (or minimize their occurrence)

- How?

איחוד תנאים =

1. Get rid of unnecessary control flow instructions → combine predicates (**predicate combining**)

פקודות מותנות =

2. Convert control dependences into data dependences → **predicated execution** = ביצוע קבוע מראש

This is actually #5 - Eliminate control-flow instructions

Predicate Combining (*not* Predicated Execution)

איחוד תנאים

- 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

הסבר בשקף הבא

- Idea: Combine predicate operations to feed a single branch instruction instead of having one branch for each
  - 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

החיסרון

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

- Condition registers exist in IBM RS6000 and the POWER architecture

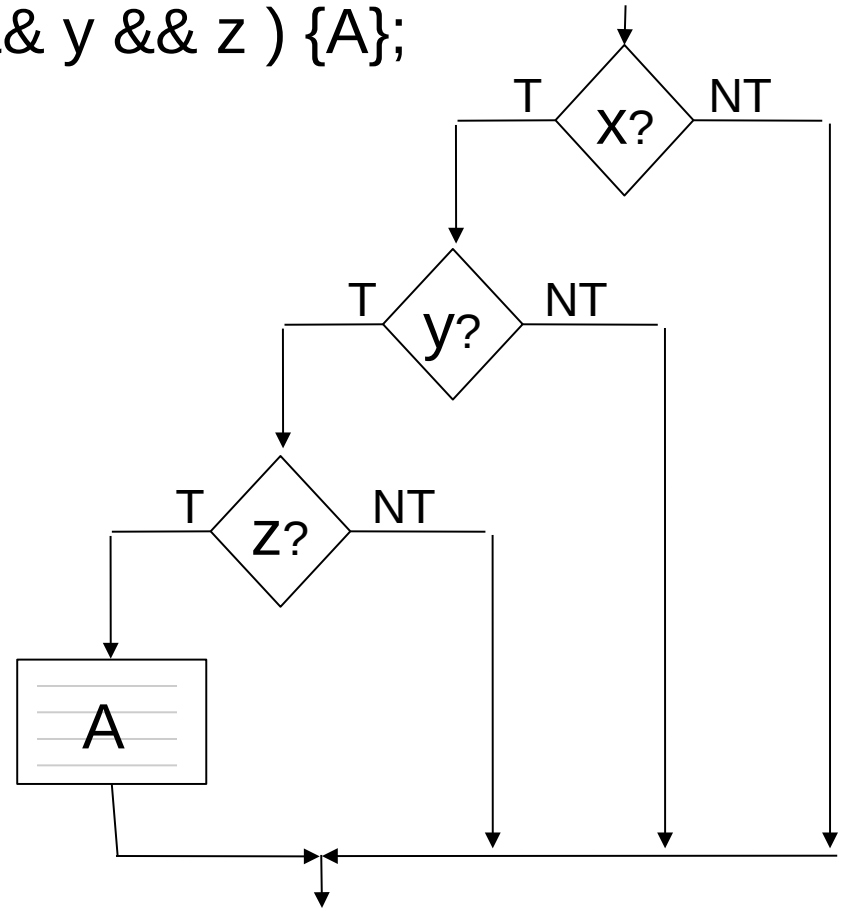
Predicate Combining
המחשת הקוד הלא יעיל בהסתעפות מורכבת

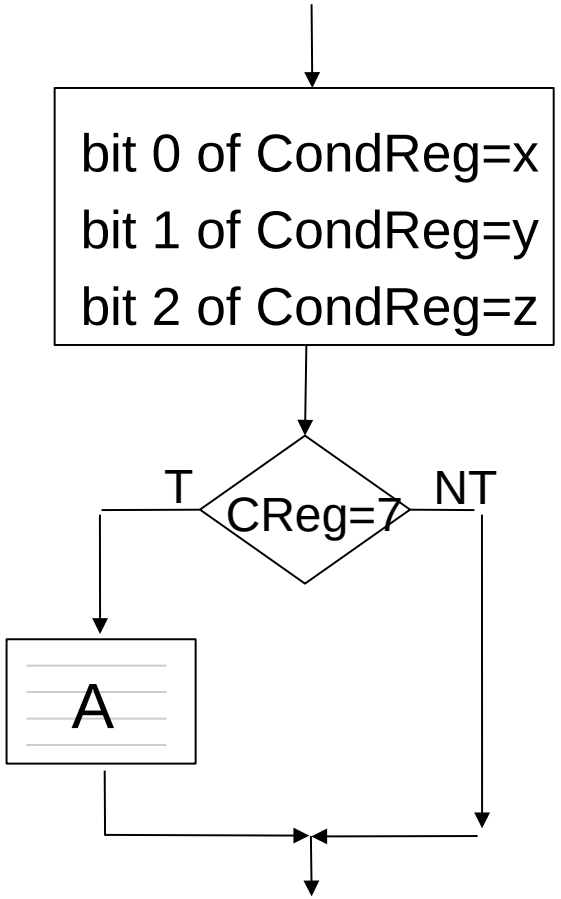
```
if (a != b) j X
if (c>=d) j Y
if (a<=5000) j Z
do A
.
.
X:
.
.
Y:
.
.
Z:
```

שקול לתנאי השקף הקודם

1) Predicate Combining – איחוד תנאים

If(x && y && z) {A};



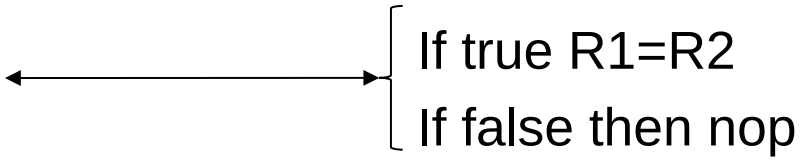


2) Predicated Execution ביצוע מותנה

אנימציה

- 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, ARM)



לכן באמצעות הוראת CMOV ניתן להעלים הסתעפויות

- 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;

כל ההוראות מתבצעות ברצף וללא קפיצות. לכן אין שינוי ב- PC

דוגמה בשפות C ו- C#

Examples of conditional expressions

The following expression determines which variable has the greater value, y or z, and assigns the greater value to the variable x:

```
x = (y > z) ? y : z;
```

The following statement is equivalent to the previous expression.

```
if (y > z)
   x = y;
else
   x = z;
```

The syntax for a conditional ref expression is as follows:

C#

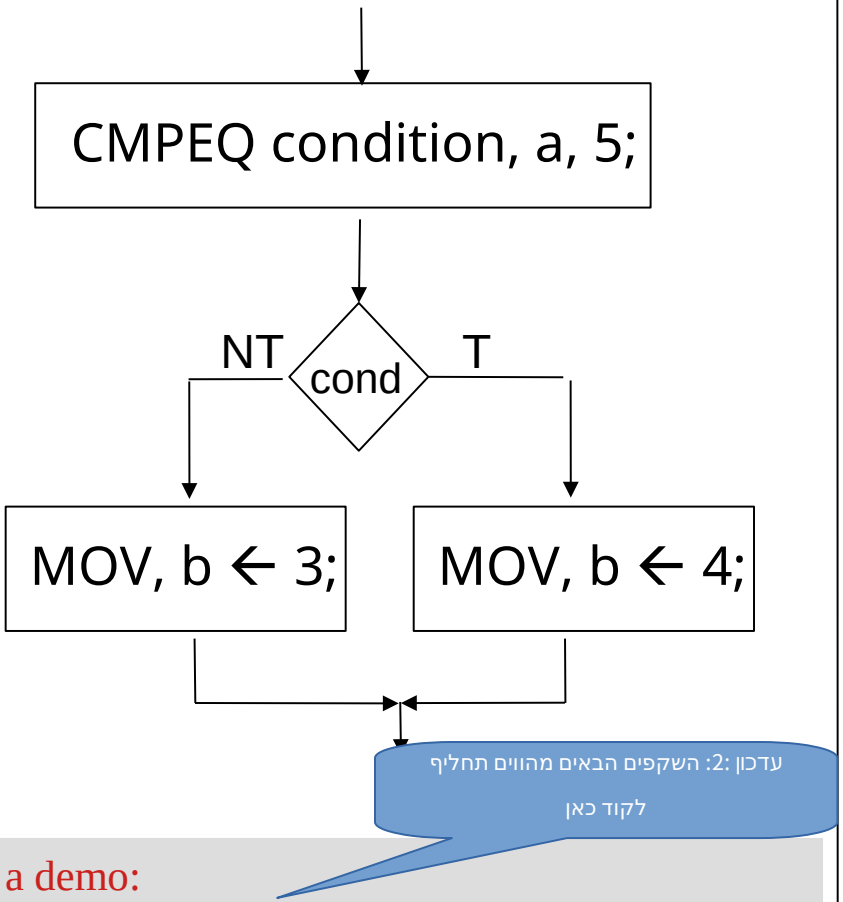
```
condition ? ref consequent : ref alternative
```

Demo: /home/telzur/science/Teaching/CPU/lectures/09/code/predicate_exe

Predicate Execution

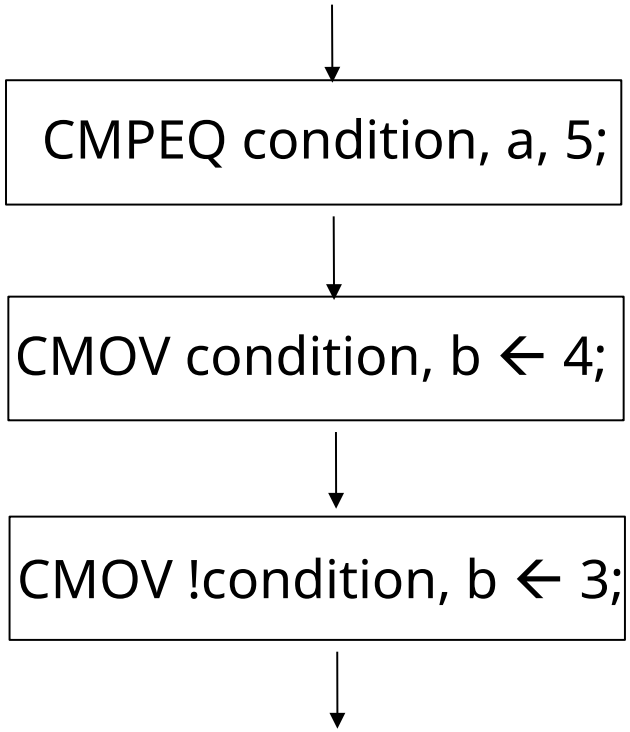
אנימציה

Instead of:



עדכון :2: השקפים הבאים מהווים תחליף לקוד כאן

We have:



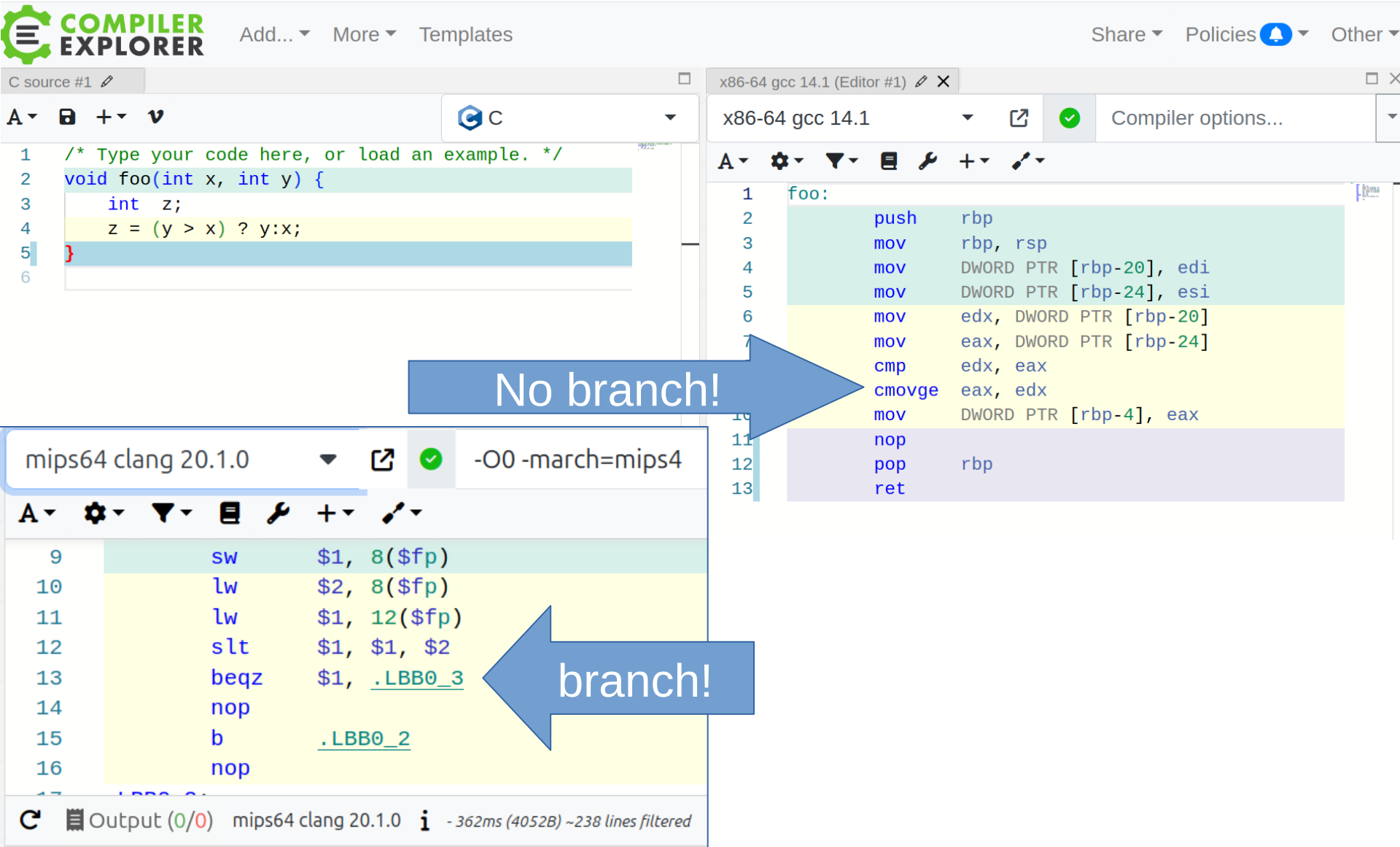
Both CMOV instructions are executed, one of them becomes NOP

Show a demo:

/home/telzur/science/Teaching/CPU/lectures/09/code/predicate_exe

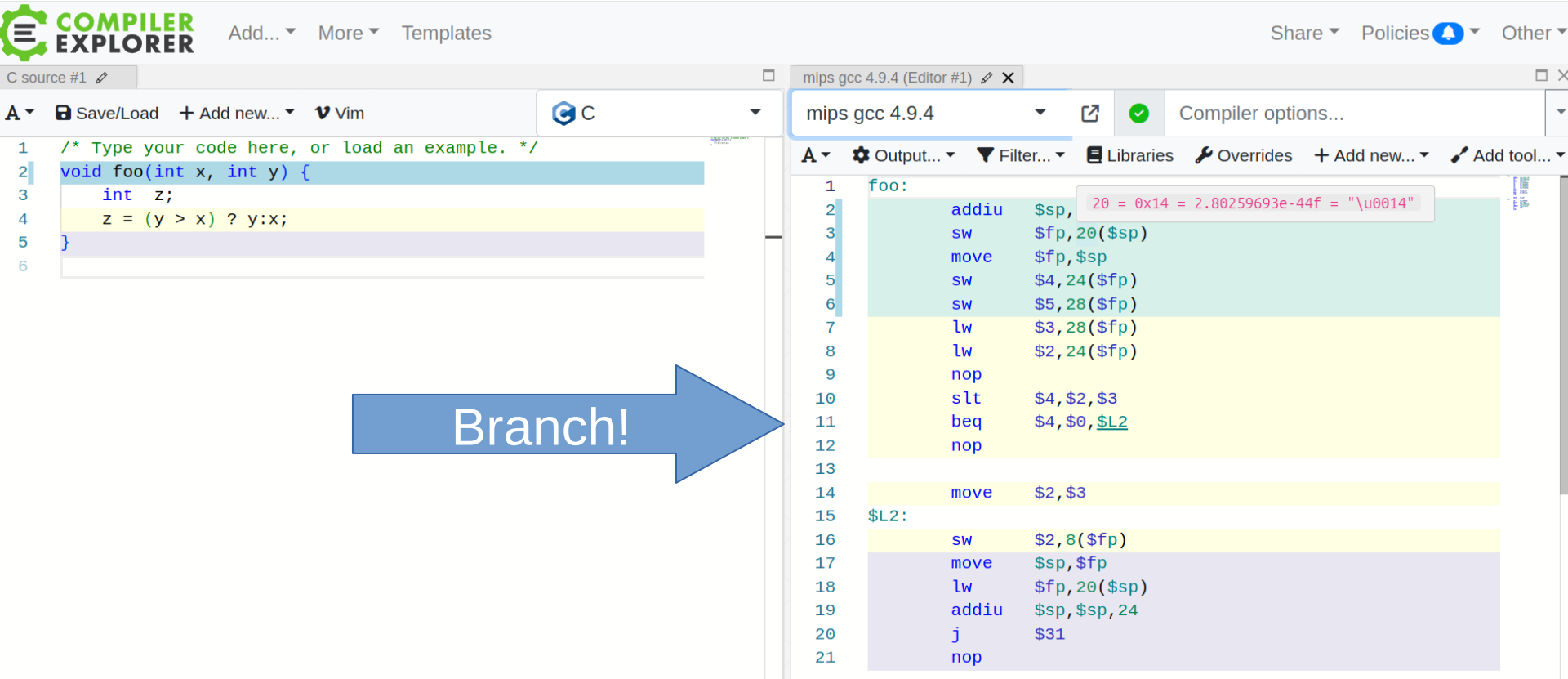
Let’s check with Compiler Explorer

X86 → 13 LOC



Let’s check with Compiler Explorer

Old MIPS → 21 LOC



Conditional execution

- Conditional execution

```
        bne r1 r2 skip
        r4=r5+r6
skip:   r7=r4+r12
```

```
r8=cmp(r1,r2)
if(r8) r4=r5 + r6
r7=r4+r12
```

-or-

```
r8=cmp(r1,r2)
r9=r5+r6
cmov (r8, r4 ← r9)
r7=r4+r12
```

Predicated Execution - summary

- Eliminates branches → enables straight line code (i.e., larger basic blocks in code)
- Advantages
  - Always-not-taken prediction works better (no branches)
  - Compiler has more freedom to optimize code (no branches)
    - control flow does not hinder (להפריע) inst. reordering optimizations
    - code optimizations hindered only by data dependencies
- Disadvantages
  - Useless work: some instructions fetched/executed but discarded (especially bad for easy-to-predict branches)
  - Requires additional ISA support
  - גרף מורכב של הסתעפויות (הגדלת הקוד)
- Q: Can we eliminate all branches this way?

A: No (loops….)

How to Handle Control Dependences

- Critical to keep the pipeline full with correct sequence of dynamic instructions.

חשוב מאוד

- Potential solutions if the instruction is a control-flow instruction:

שישה פתרונות אפשריים:

- Stall the pipeline until we know the next fetch address
- Guess the next fetch address (branch prediction)

זה הפוקוס של ההרצאה הנוכחית

- 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: Guessing NextPC = PC + 4

- Always predict the next sequential instruction is the next instruction to be executed
- This is a form of next fetch address prediction (and branch prediction)
- How can you make this more effective?
- Idea: Maximize the chances that the next sequential instruction is the next instruction to be executed
  - Software: Lay out the control flow graph such that the "likely next instruction" is on the not-taken path of a branch
    - **Profile** guided code positioning → Pettis & Hansen, PLDI 1990.
  - Hardware: ??? (how can you do this in hardware…)
    - Cache traces of executed instructions → **Trace cache**

Review: Guessing Next PC = PC + 4

- How else can you make this more effective?

- Idea: Get rid of control flow instructions (or minimize their occurrence)

- How?

  1. Get rid of unnecessary control flow instructions →

  predicate combining → condition registers

  2. Convert control dependences into data dependences → predicated execution

על אלה כבר דיברנו בשקפים הקודמים

Reducing number of branches executed

- **Loop unrolling**

```
for(i=0;i<10000;i++)
{
   A[i]=B[i]+C[i];
}

for(i=0;i<10000;i=i+2)
{
   A[i]=B[i]+C[i];
   A[i+1]=B[i+1]+C[i+1];
}
```

Demo: ~/science/Teaching/CPU/lectures/09/code/loop_unrolling

caution: יש עוד שחקנים שנהנים מהשינוי, יתכן שהשינוי מטיב עם המטמון וזו הסביה לקיצור הדרמטי בזמן הריצה ולא צימצום ההסתעפויות

Credit:
https://www.eecs.umich.edu/courses/eecs470/OLD/w14/lectures/470L17W14.pdf

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)

Delayed Branching (I)

- Change the semantics of a branch instruction
  - Branch takes effect after N instructions (לא קורה מיידית)
  - Or the branch takes effect after N cycles זו אמירה שקולה לקודמת
- Idea: Delay the execution of a branch. N instructions (delay slots) that come after the branch are **always** executed regardless of branch direction.

In MIPS N=1 (1 delay slot)

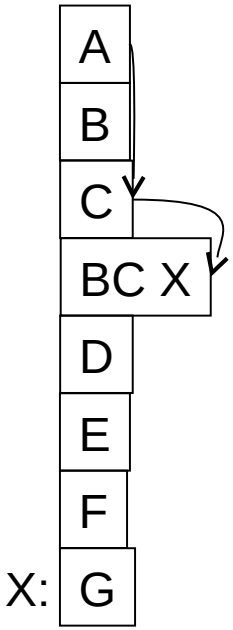
- Problem: How do you find instructions to fill the delay slots?
  - Branch must be independent of delay slot instructions

- Unconditional branch: Easier to find instructions to fill the delay slot
- Conditional branch: Condition computation should not depend on instructions in delay slots → difficult to fill the delay slot

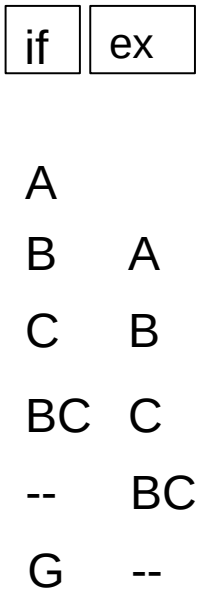
Delayed Branching (II)

אנימציה

Normal code:

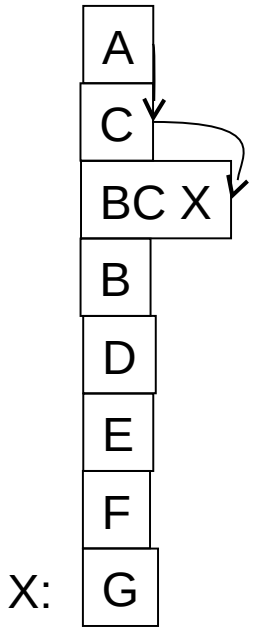


Timeline:

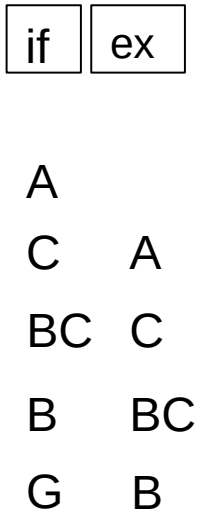


6 cycles

Delayed branch code:



Timeline:



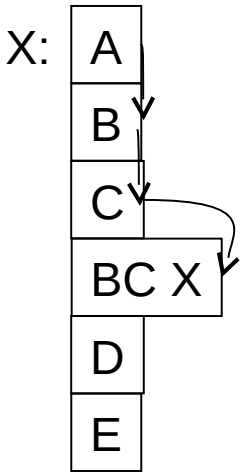
5 cycles

שיפור ביצועים ב- 16.67%. העברנו את B לאחר ה-Branch

Fancy Delayed Branching (III)

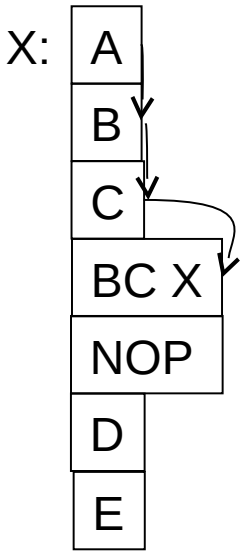
- Delayed branch with squashing
  - In SPARC
  - Semantics: If the branch falls through (i.e., it is not taken), the delay slot instruction is not executed
  - Why could this help?

Normal code:



כאן אין הוראה פנויה למלא את ה- delay slot

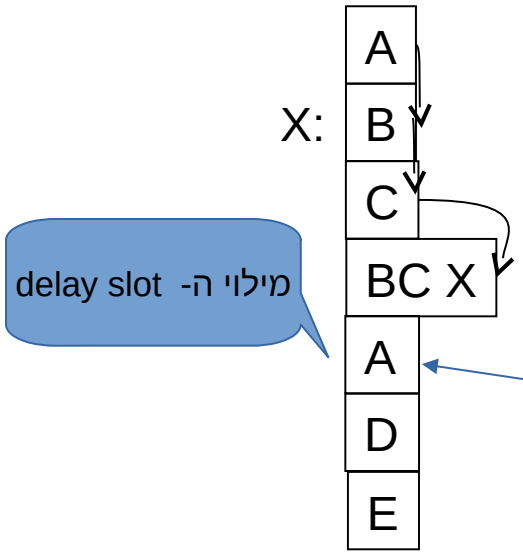
Delayed branch code:



הדרך של MIPS

Delayed branch w/ squashing:

מעוך



אם הbranch יילקח נחשב את A ואם לא A ימעך

זה טוב במקרה של לולאה

Delayed Branching (IV)

- Advantages:

  + Keeps the pipeline full with useful instructions in a simple way assuming

  1. Number of delay slots == number of instructions to keep the pipeline full before the branch resolves (לכן ככל שהפייפליין גדול יותר קשה יותר לספק את התנאי למילוי הפייפליין)

  2. All delay slots can be filled with useful instructions

- Disadvantages:

  -- Not easy to fill the delay slots (even with a 2-stage pipeline)

  1. Number of delay slots increases with pipeline depth, superscalar execution width (במעבד עוצמתי עדכני המספר נע בין 0 1 ל- 0 2 וזה לא מעשי)

  2. Number of delay slots should be variable with variable latency operations. Why?

  -- Ties ISA semantics to hardware implementation

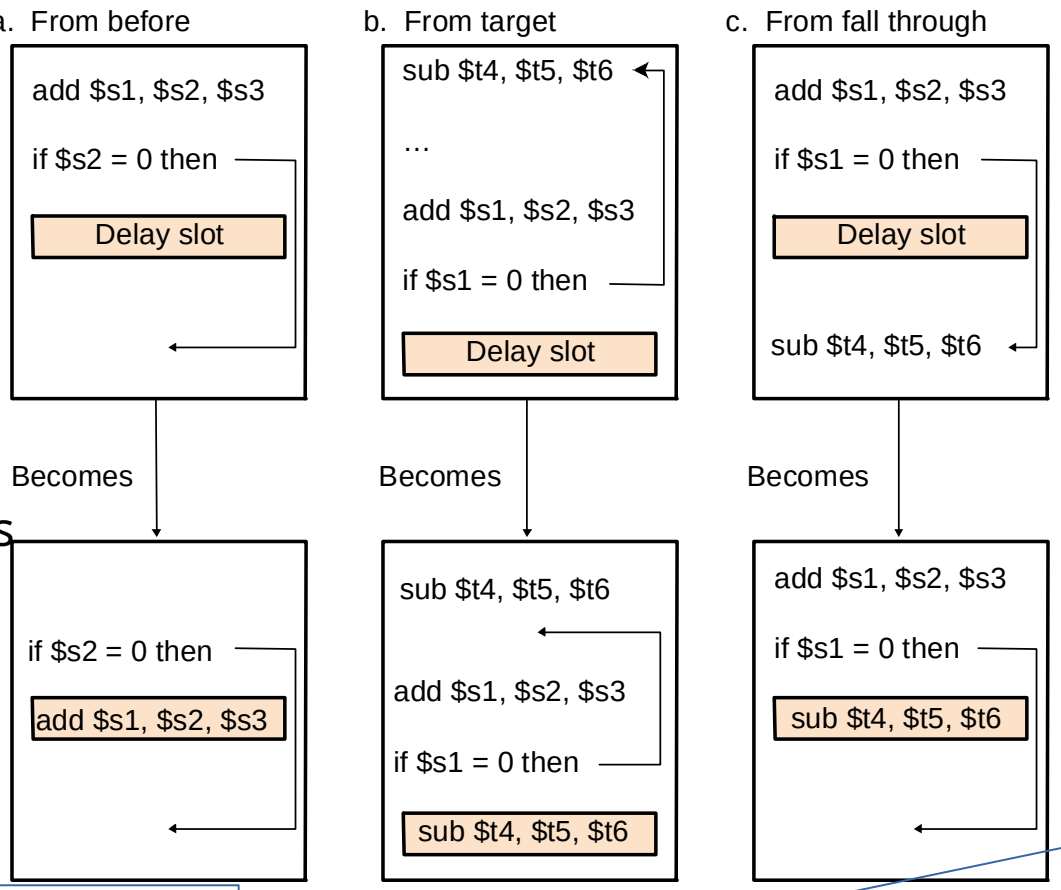
  -- SPARC, MIPS, HP-PA: 1 delay slot (להסתכל "אחורה", זוכרים?)

  -- What if pipeline implementation changes with the next design?

צריך לקמפל מחדש אם המימוש משתנה בדור הבא של המחשב

An Aside: Filling the Delay Slot

reordering data independent (RAW, WAW, WAR) instructions does not change program semantics



מה קורה אם במקום sub יש div והחלוקה היא באפס? דבר זה מייצר exception. האם ה-exception יתנהג אותו הדבר במקום של ה delay slot????

Safe?

within same basic block

For correctness: add a new instruction to the not-taken path?

For correctness: add a new instruction to the taken path?

Fix-up code (compensation code)

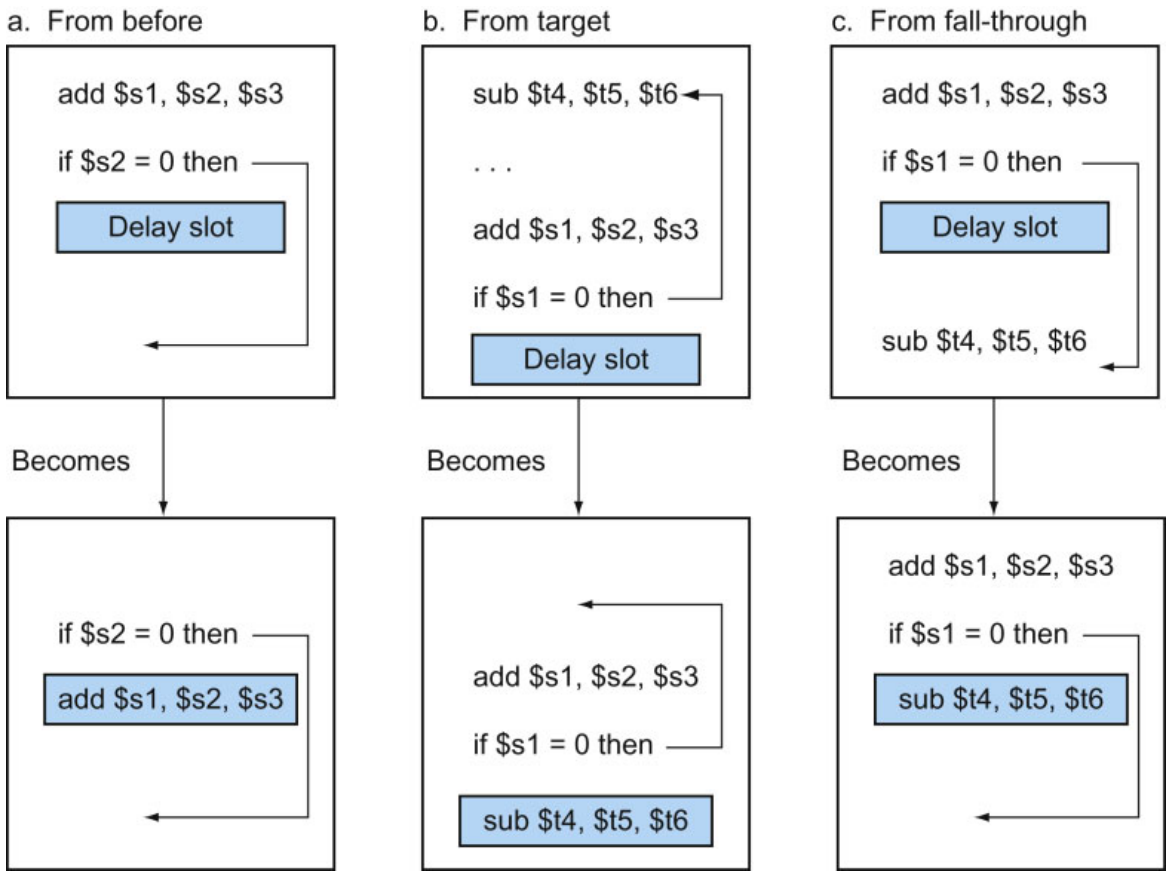


FIGURE 4.64 Scheduling the branch delay slot. The top box in each pair shows the code before scheduling; the bottom box shows the scheduled code. In (a), the delay slot is scheduled with an independent instruction from before the branch. This is the best choice. Strategies (b) and (c) are used when (a) is not possible. In the code sequences for (b) and (c), the use of $s1 in the branch condition prevents the add instruction (whose destination is $s1) from being moved into the branch delay slot. In (b) the branch delay slot is scheduled from the target of the branch; usually the target instruction will need to be copied because it can be reached by another path. Strategy (b) is preferred when the branch is taken with high probability, such as a loop branch. Finally, the branch may be scheduled from the not-taken fall-through as in (c). To make this optimization legal for (b) or (c), it must be OK to execute the sub instruction when the branch goes in the unexpected direction. By "OK" we mean that the work is wasted, but the program will still execute correctly. This is the case, for example, if $t4 were an unused temporary register when the branch goes in the unexpected direction.

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)

Fine-grained Multi-Threading

1. **Fine-grained Multithreading**: switching among threads happens at each instruction, independently from the the fact that the thread instruction has caused a cache miss.

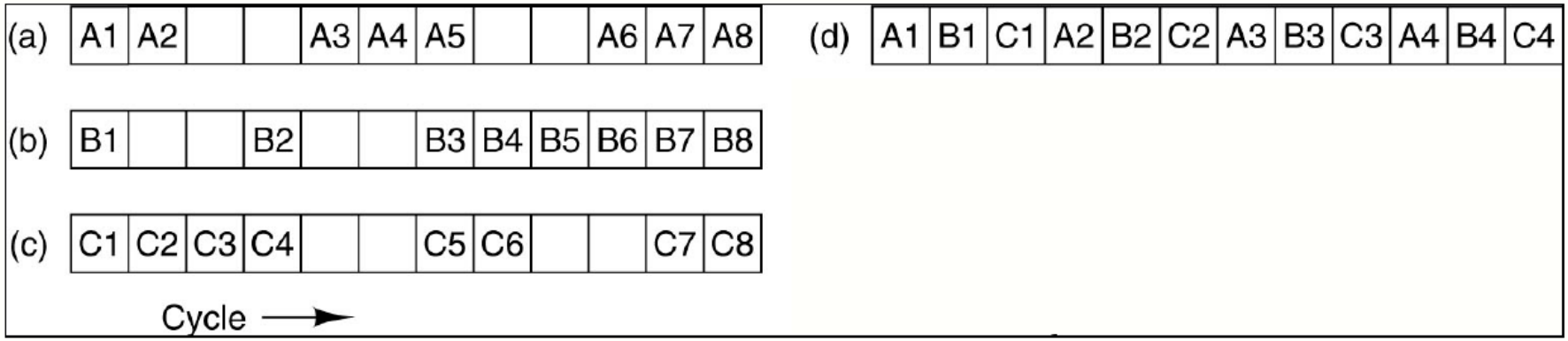
- Instructions "scheduling" among threads obeys a round robin policy, and the CPU must carry out the switch with *no* overhead, since overhead cannot be tolerated

- If there is a sufficient number of threads, it is likely that at least one is active (not stalled), and the CPU can be kept running.

יש כאן מענה ל-Control Dependence אבל מה עם Data Dependence?????

Fine-grained Multi-Threading

- (a)-(c) three threads and associated stalls (empty slots).
  (d) Fine-grained multithreading. Each slot is a clock cycle, and we assume for simplicity that each instruction can be completed in a clock cycle, unless a stall happens.
  (Tanenbaum, Fig. 8.7)



- In this example, 3 threads keep the CPU running, but what if A2 stall lasts 3 or more clock cycles?

תשובה: במקרה כזה יחלחלו STALLS לתוך תרשים d

Fine-grained Multi-Threading

- CPU stalls can be due to a cache miss, but also to a true data dependence, or to a branch: dynamic ILP techniques do not always guarantee that a pipeline stall is avoided.
- With fine-grained multithreading in a pipelined Architecture, if:
  - the pipeline has *k* stages,
  - there are at least *k* threads to be executed,
  - and the CPU can execute a thread switch at each clock cycle
- then there can never be more than a single instruction per thread in the pipeline at any instant, so there cannot be hazards due to dependencies, and the pipeline never stalls (… another assumption is required …).

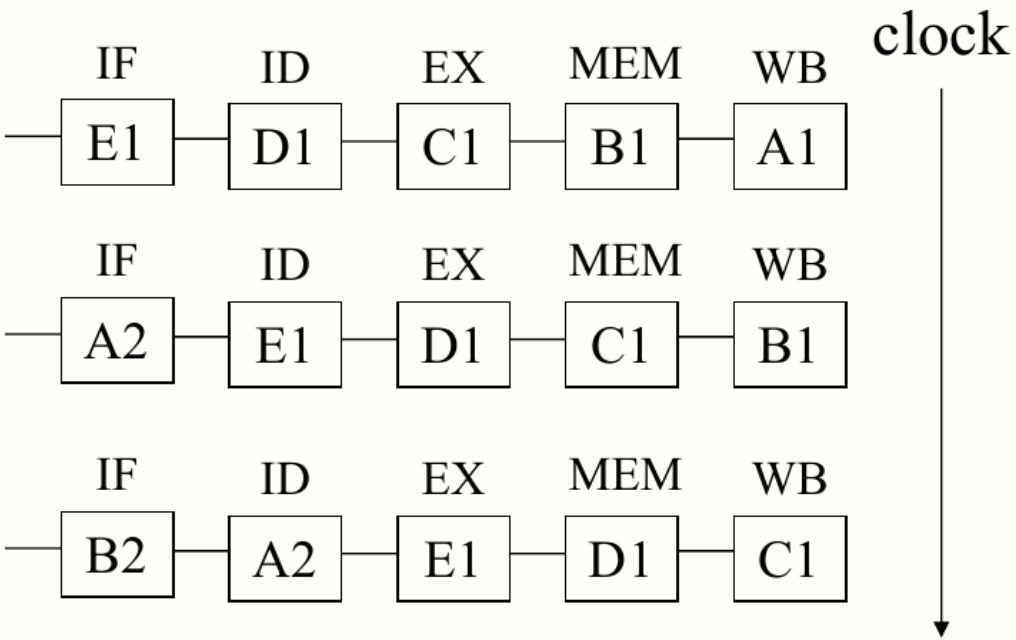
שיש מספיק רגיסטרים...

Fine-grained Multi-Threading

- Fine-grained multithreading in a CPU with a 5-stage pipeline: there are never two instructions of the same thread concurrently active in the pipeline. If instructions can be executed out of order, then it is possible to keep the CPU fully busy even in case of a cache miss.

5 threads in execution:

A1	A2	A3	A4	A5	A6	...
B1	B2	B3	B4	B5	B6	...
C1	C2	C3	C4	C5	C6	...
D1	D2	D3	D4	D5	D6	...
E1	E2	E3	E4	E5	E6	...



Fine-Grained Multi-threading

- Idea: Hardware has multiple thread contexts. Each cycle, fetch engine fetches from a different thread.
  - By the time the fetched branch/instruction resolves, no instruction is fetched from the same thread
  - Branch/instruction resolution latency overlapped with execution of other threads' instructions

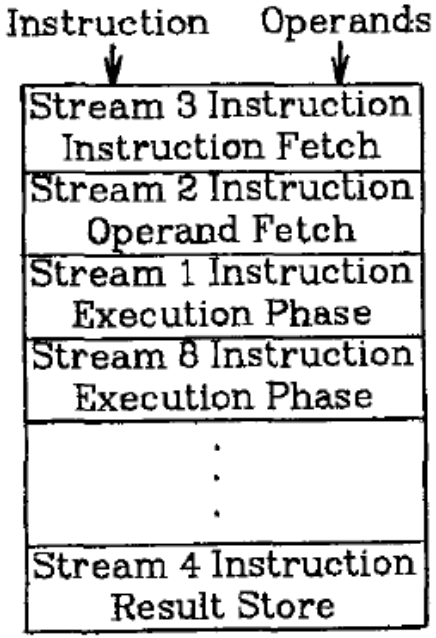
+ No logic needed for handling control and data dependences within a thread

-- Single thread performance suffers (פעם ב- N מחזורים)

-- Extra logic for keeping thread contexts

-- Does not overlap latency if not enough threads to cover the whole pipeline

(אם אין מספיק תהליכונים, יפעלו מסמן במקום)



Fine-grained Multi-threading (II)

- Idea: Switch to another thread every cycle such that no two instructions from a thread are in the pipeline concurrently

- **Tolerates** the control and data dependency **latencies** by overlapping the latency with useful work from other threads
- Improves pipeline utilization by taking advantage of multiple threads

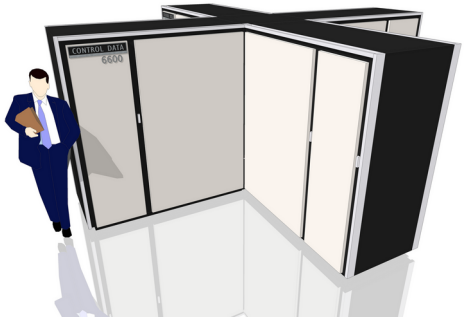
- Thornton, "Parallel Operation in the Control Data 6600," AFIPS 1964.
- Smith, "A pipelined, shared resource MIMD computer," ICPP 1978.

בשני השקפים הבאים מובאים אזכורים משני מאמרים אלה.

Fine-grained Multi-threading: History

- **CDC 6600**'s peripheral processing unit is fine-grained multi-threaded
  - Thornton, "Parallel Operation in the Control Data 6600," AFIPS 1964.
  - Processor executes a different I/O thread every cycle
  - An operation from the same thread is executed every 10 cycles
- **Denelcor HEP** (Heterogeneous Element Processor)
  - Smith, "A pipelined, shared resource MIMD computer," ICPP 1978.
  - 120 threads/processor
  - available queue vs. unavailable (waiting) queue for threads
  - each thread can have only 1 instruction in the processor pipeline; each thread independent
  - to each thread, processor looks like a non-pipelined machine
  - system throughput vs. single thread performance trade-off

PARALLEL OPERATION IN THE CONTROL DATA 6600



James E. Thornton
Control Data Corporation
Minneapolis, Minnesota

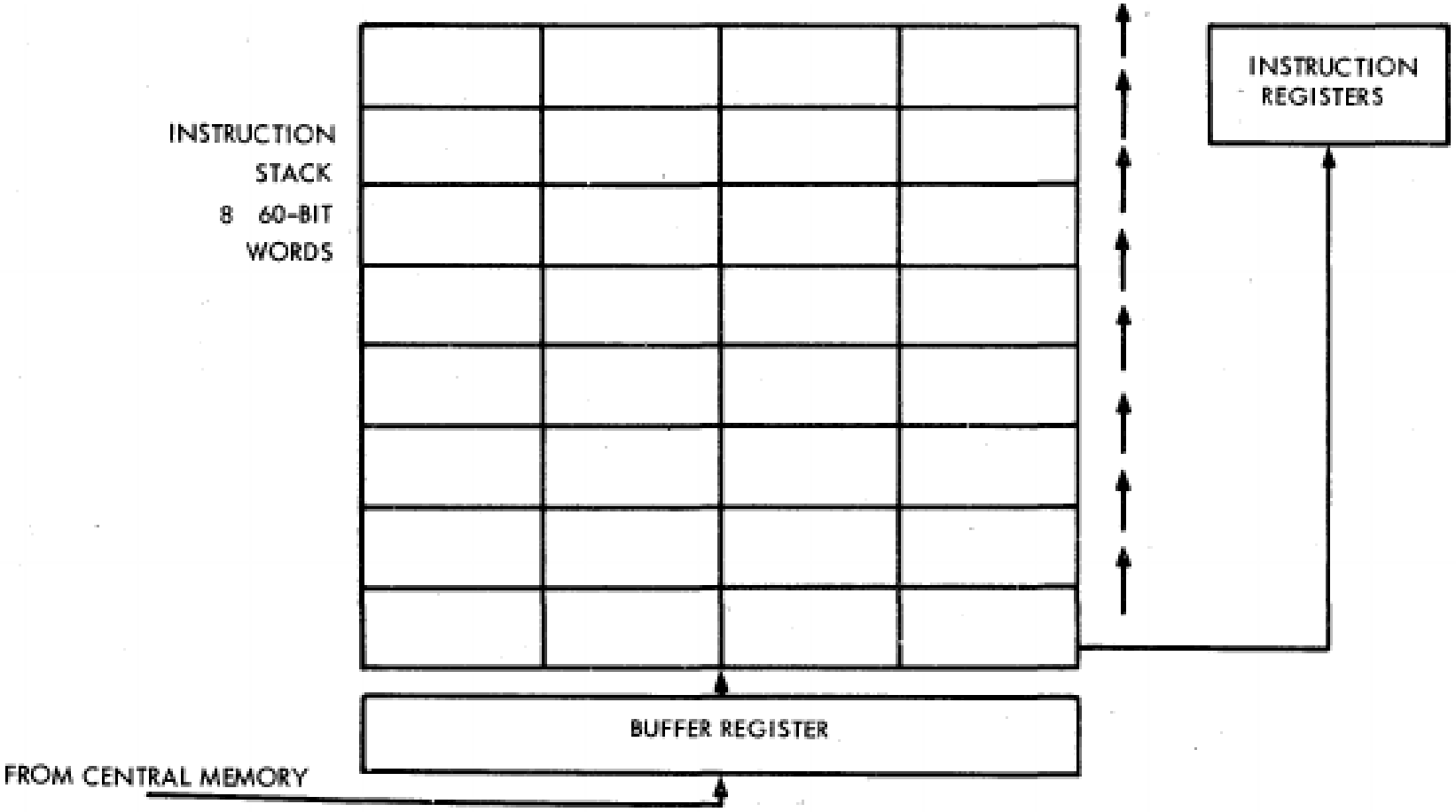
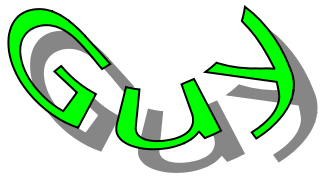


Figure 5. 6600 Instruction Stack Operation.



First commercial machine to use hardware threading in main CPU

- 120 threads per processor
- 10 MHz clock rate
- Up to 8 processors
- precursor to Tera MTA (Multithreaded Architecture)

Wikipedia: The Heterogeneous Element Processor (HEP) was introduced by Denelcor, Inc. in 1982 as the **world's first commercial MIMD computer**. The HEP's architect was Burton Smith. The machine was designed to solve fluid dynamics problems for the Ballistic Research Laboratory.[1] A HEP system, as the name implies, was pieced together from many heterogeneous components -- processors, data memory modules, and I/O modules. The components were connected via a switched network.

A PIPELINED, SHARED RESOUCE MIMD COMPUTER

Burton J. Smith
Denelcor, Inc.
Denver, Colorado 80205

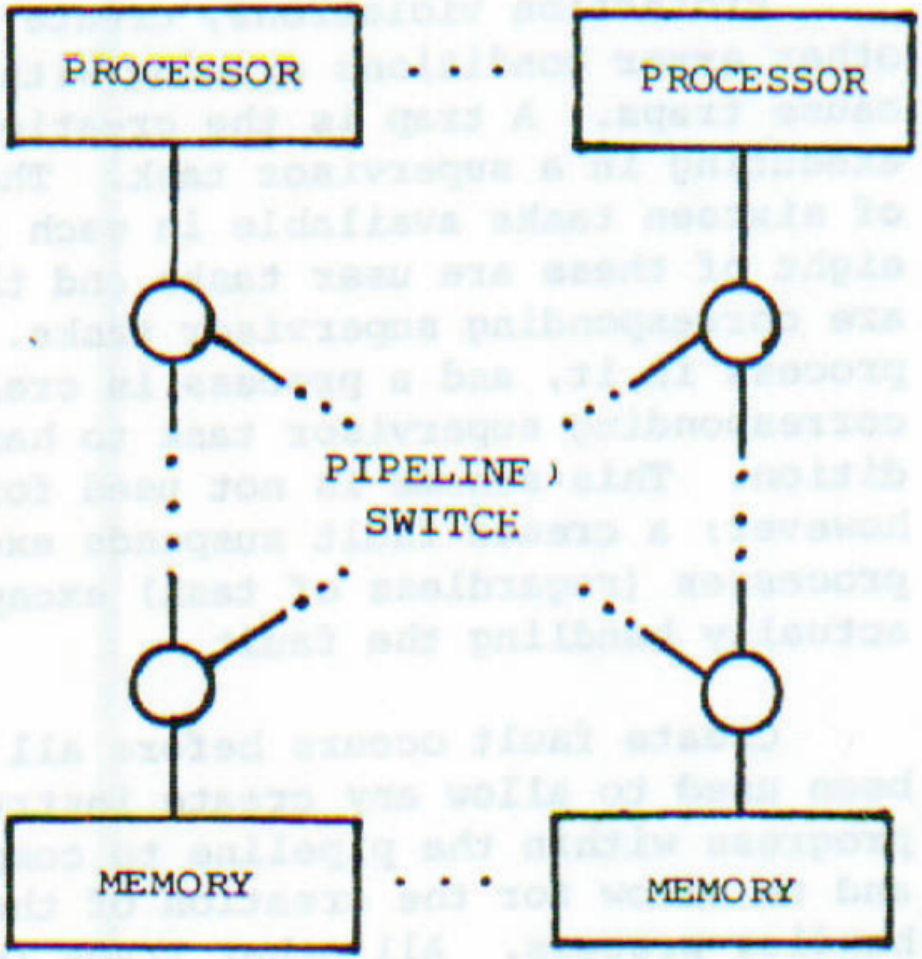
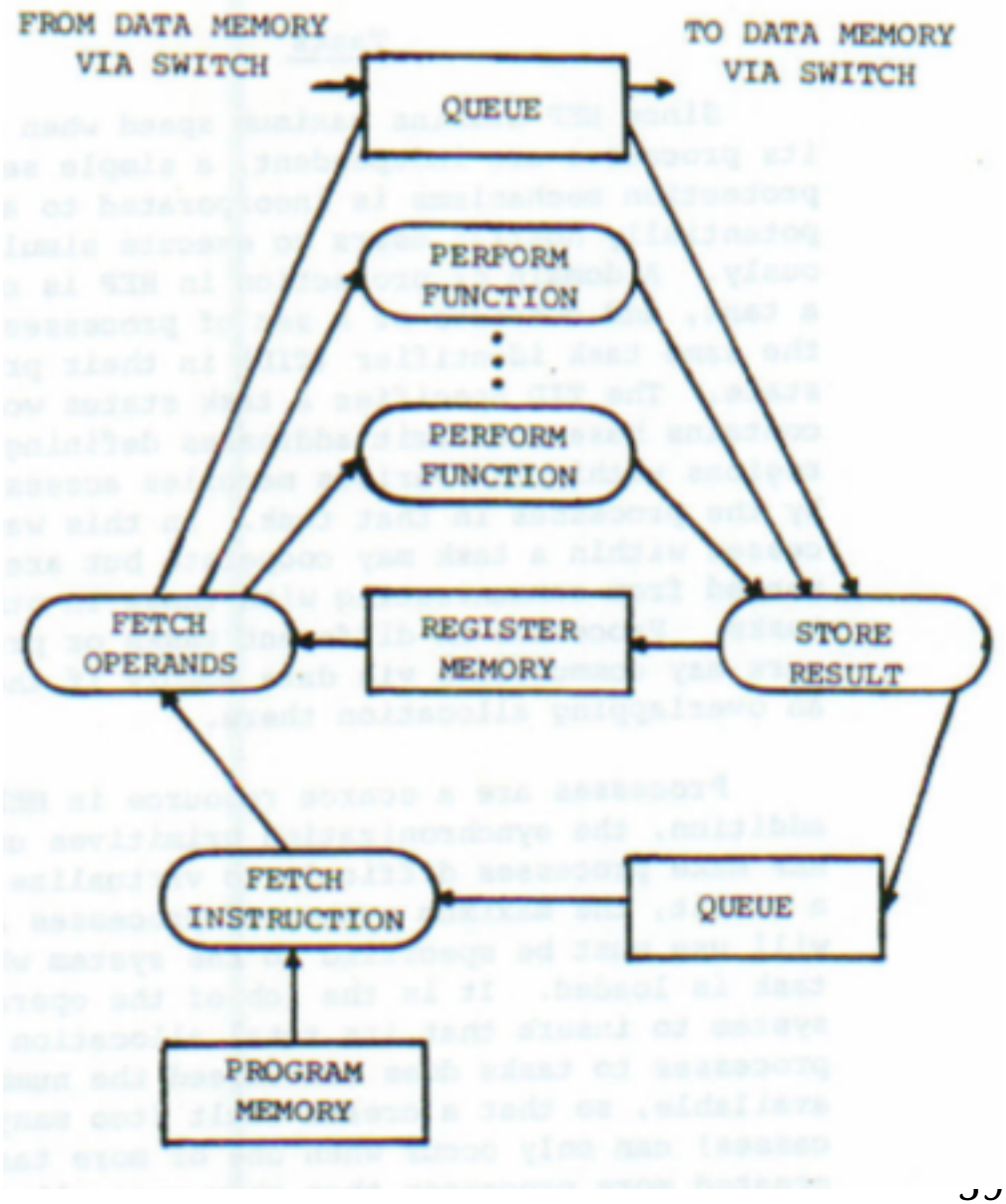


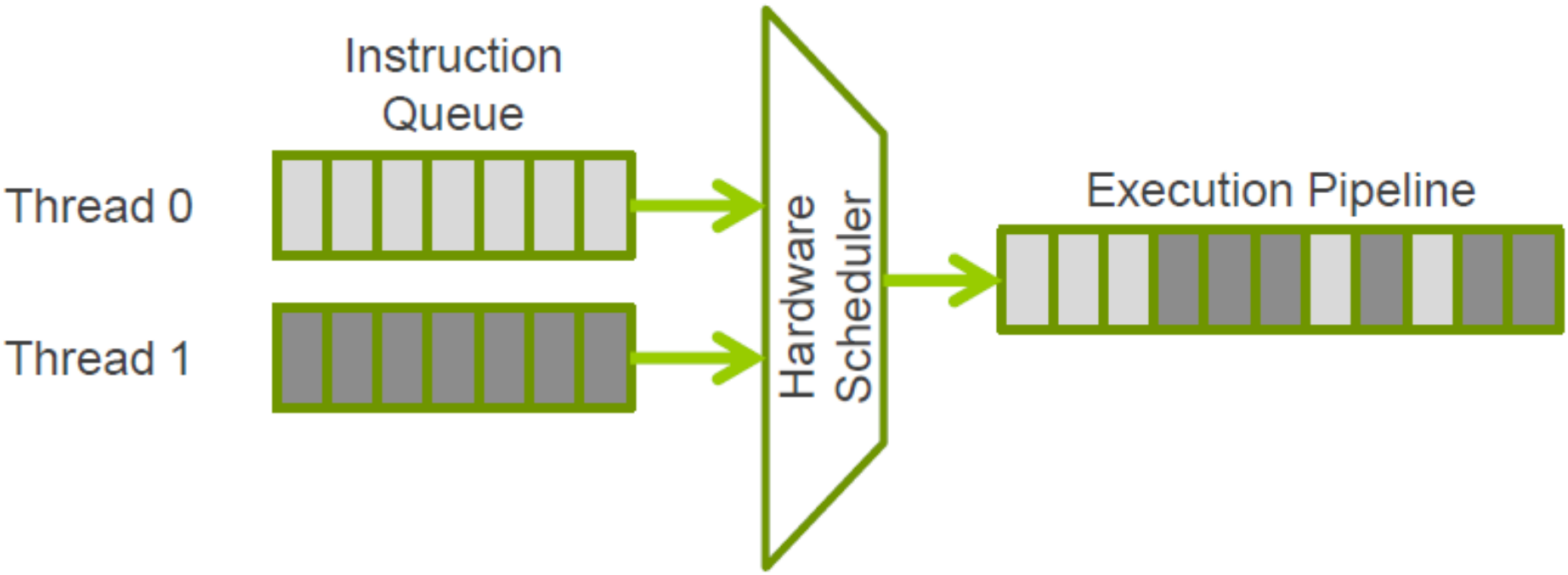
Figure 2. Overall System Layout

Fine-grained Multi-threading in HEP

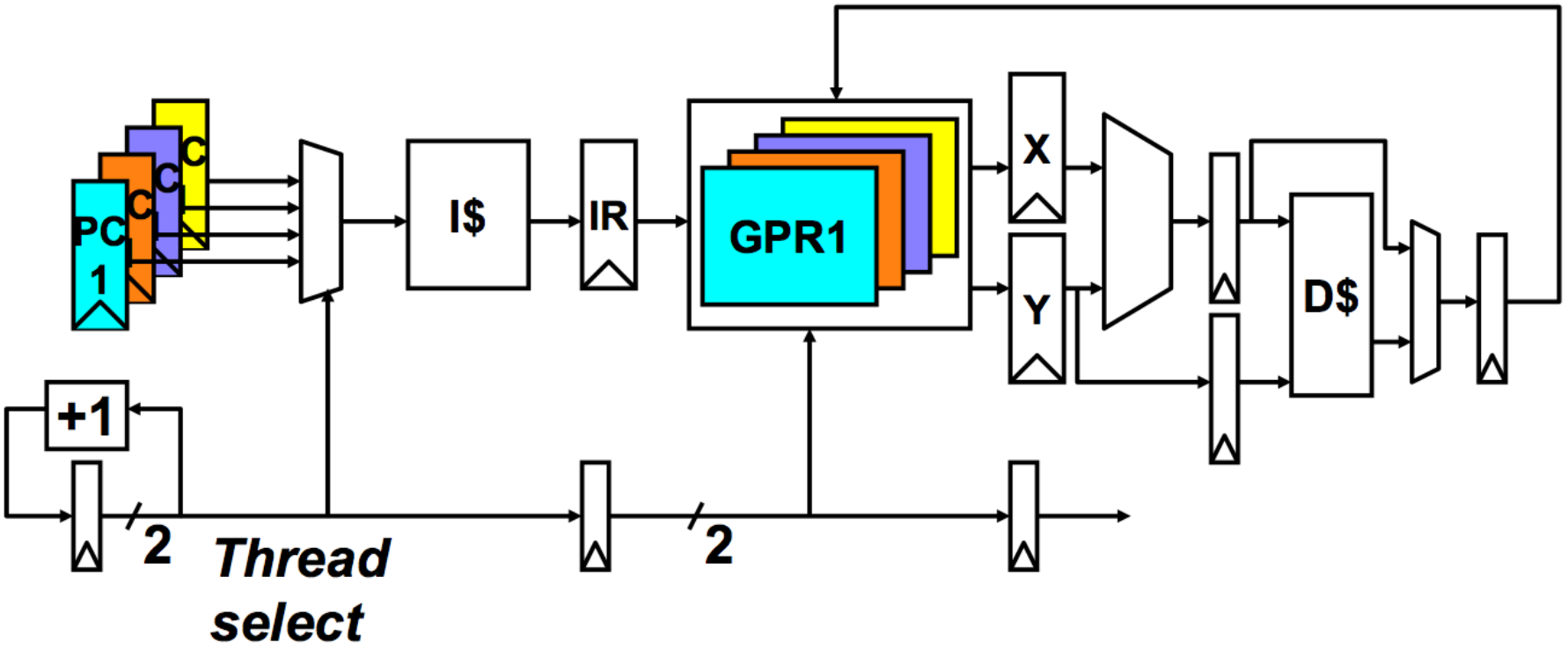
- Cycle time: 100ns
- 8 stages → 800 ns to complete an instruction
  - assuming no memory access
- No control and data dependency checking



Multi-threaded Pipeline Example - MIPS

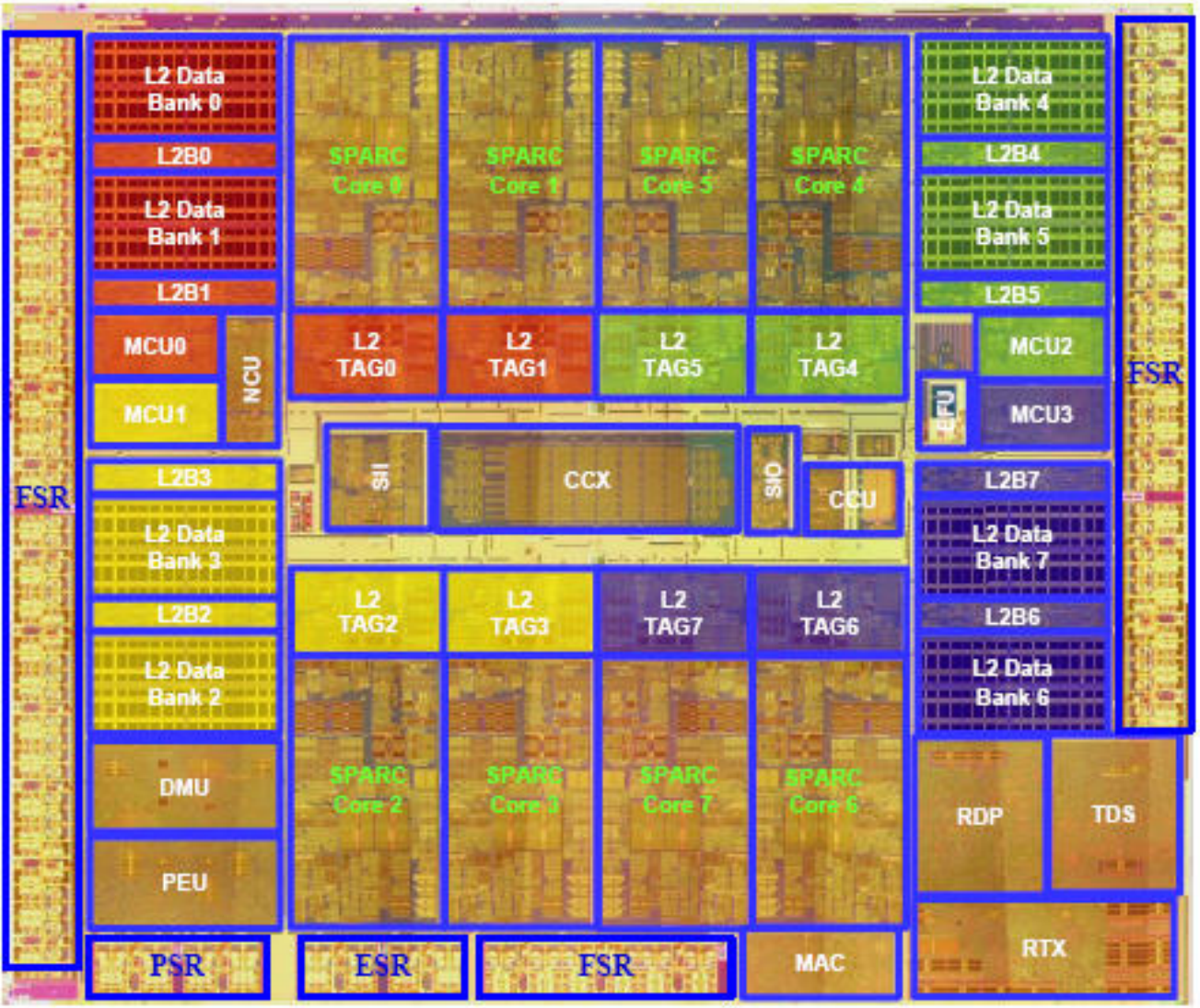


Multi-threaded Pipeline Example



Slide credit: Joel Emer

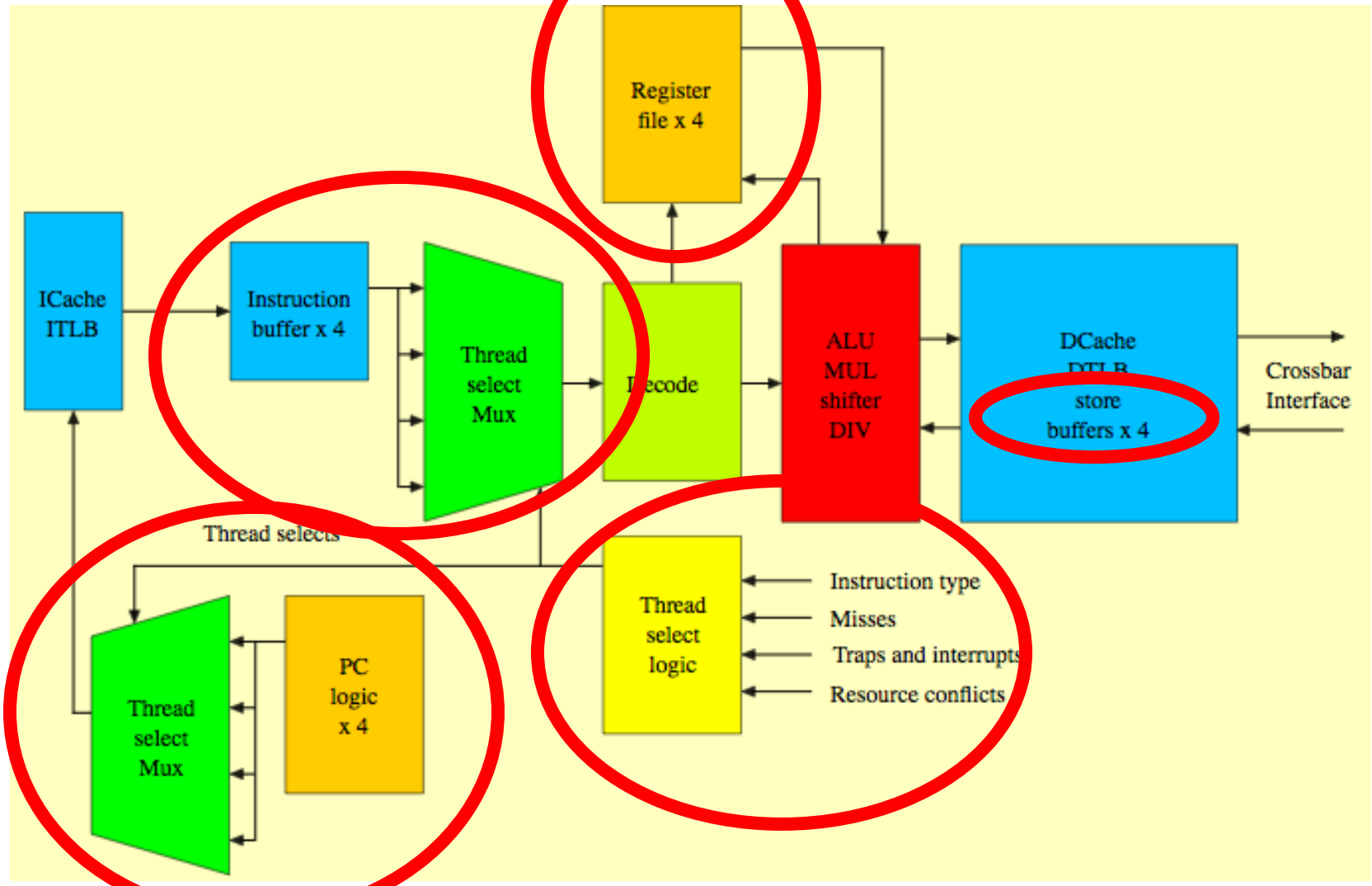
Niagara2 Chip Overview



- 8 Sparc cores, 8 threads each
- Shared 4MB L2, 8-banks, 16-way associative
- Four dual-channel FBDIMM memory controllers
- Two 10/1 Gb Enet ports
- One PCI-Express x8 1.0A port
- 342 mm^2 die size in 65 nm
- 711 signal I/O, 1831 total

אנימציה

Sun Niagara Multithreaded Pipeline



Kongetira et al., "Niagara: A 32-Way Multithreaded Sparc Processor," IEEE Micro 2005.

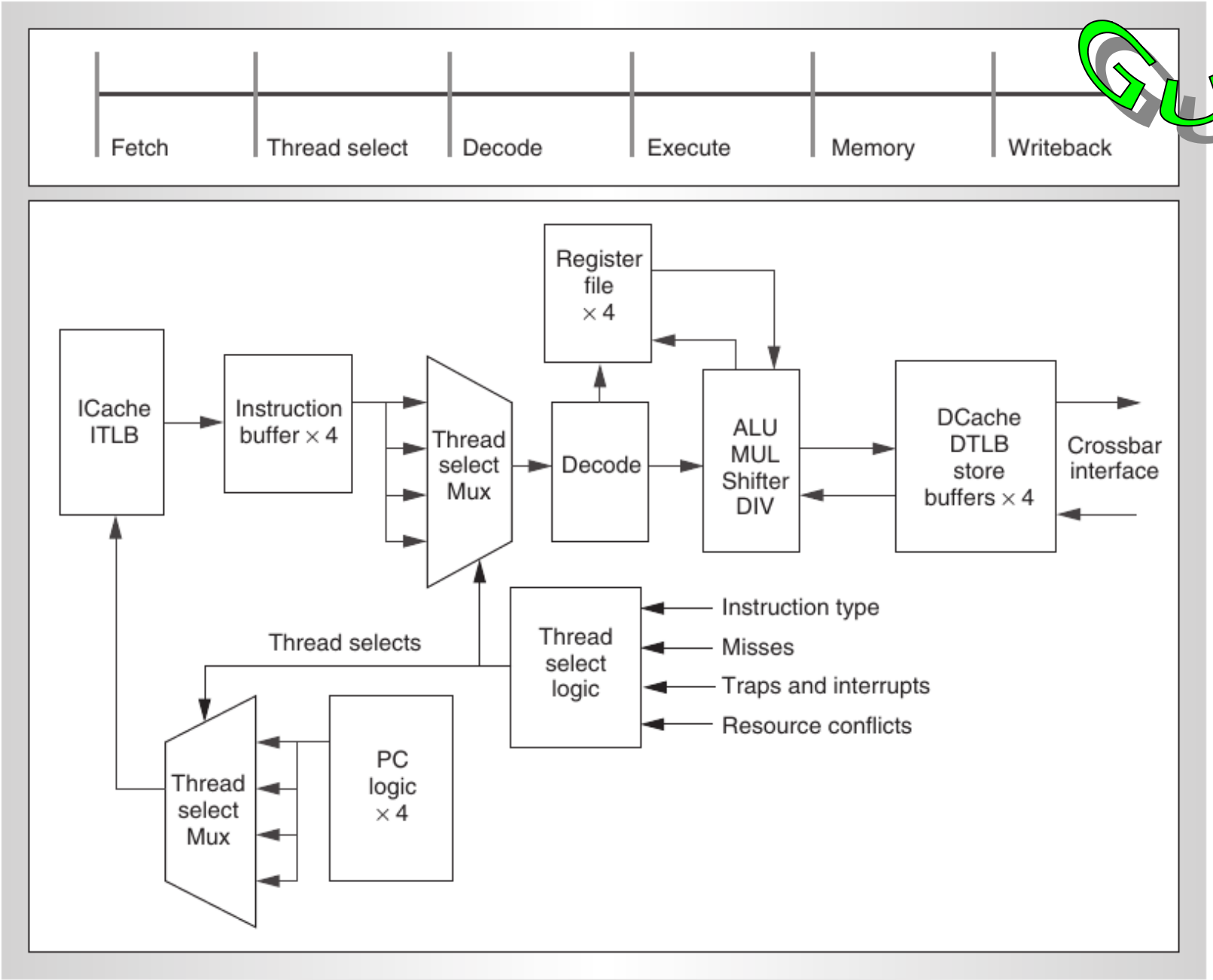


Figure 3. Sparc pipeline block diagram. Four threads share a six-stage single-issue pipeline with local instruction and data caches. Communication with the rest of the machine occurs through the crossbar interface.

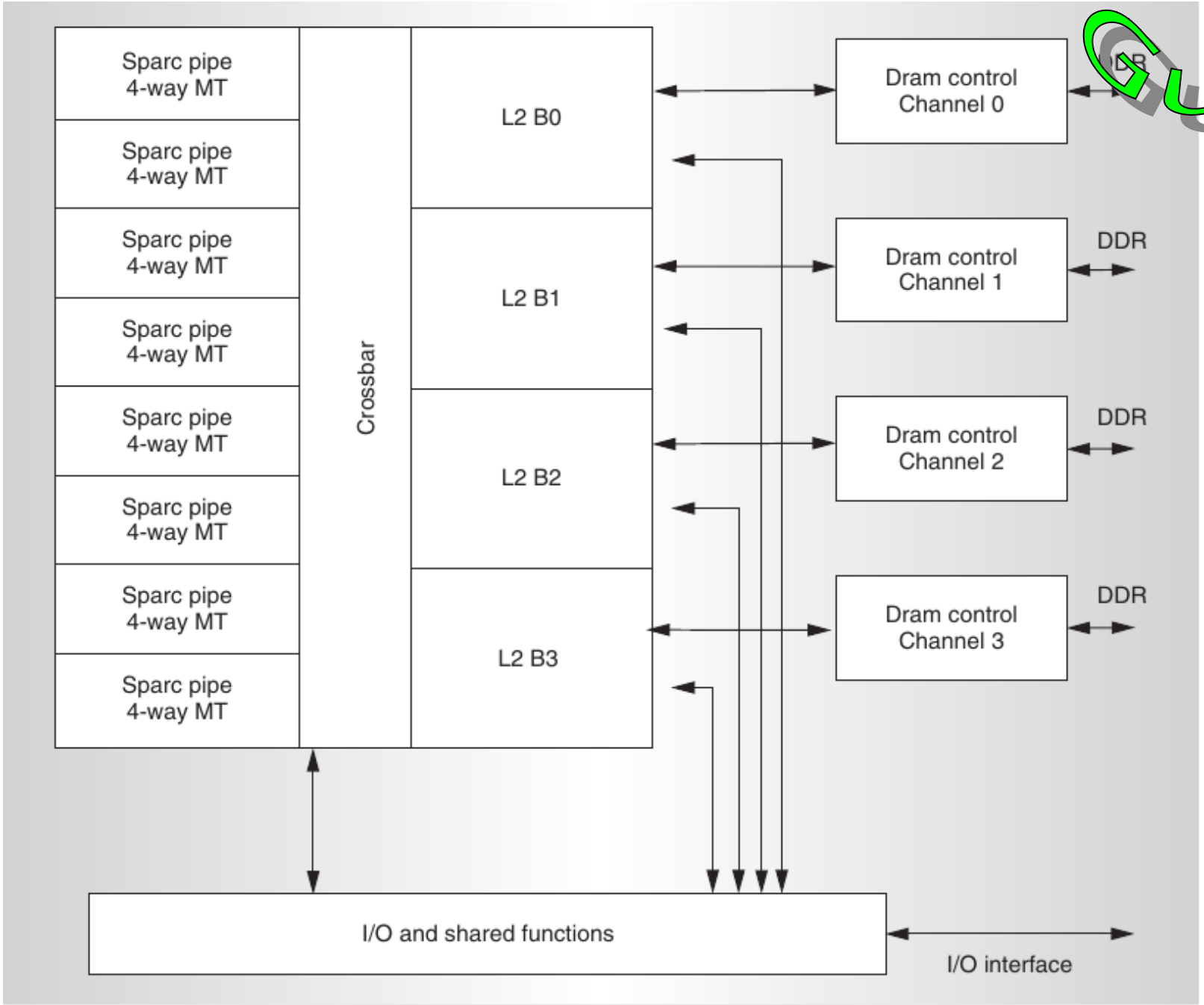


Figure 2. Niagara block diagram.

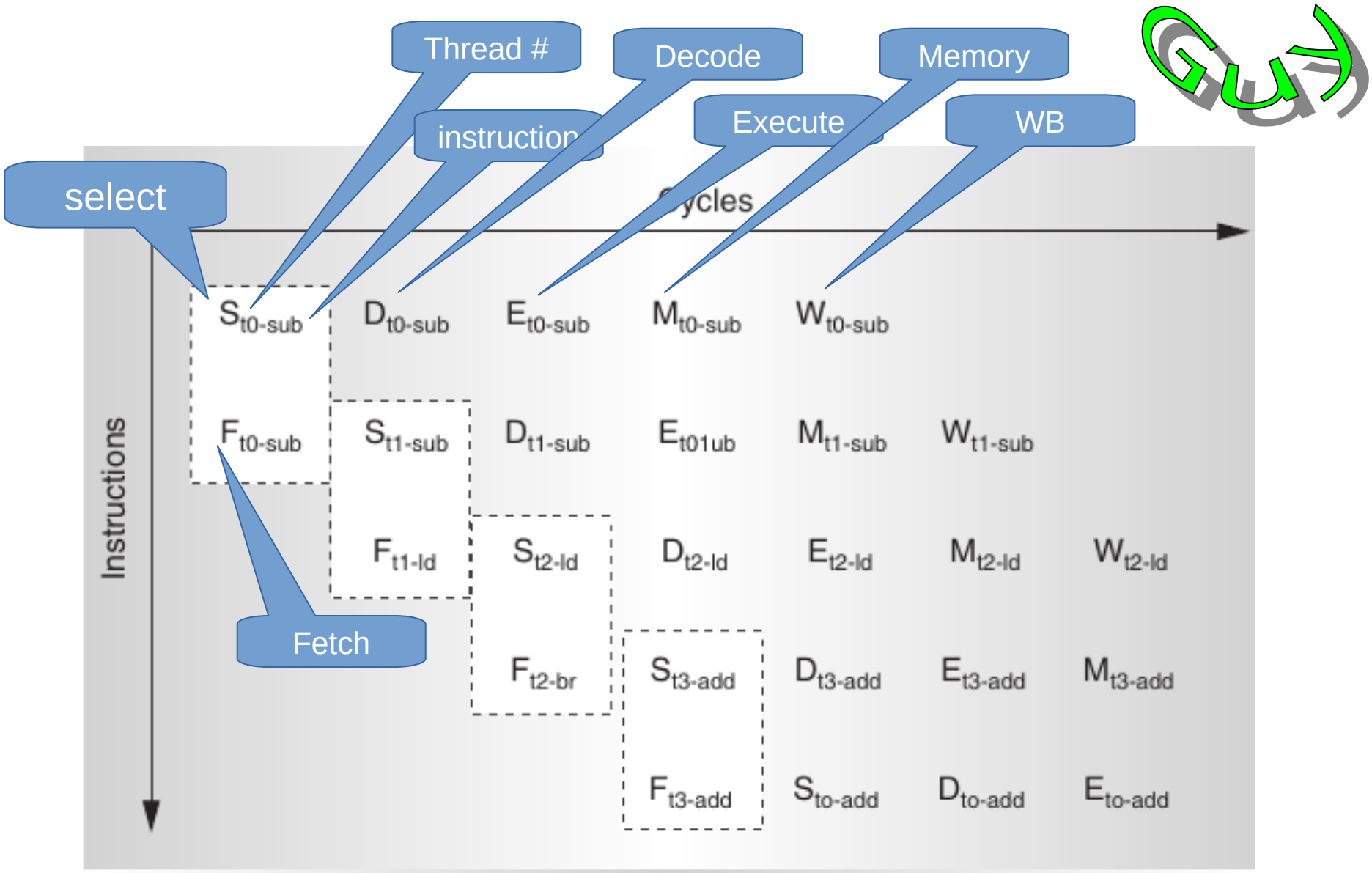


Figure 4. Thread selection: all threads available.

Fine-grained Multi-threading

- Advantages

  + **No need for dependency checking** between instructions

    (only one instruction in pipeline from a single thread)

  + **No need for branch prediction** logic

  + Otherwise-bubble cycles used for executing useful instructions from different threads

  + Improved system throughput, latency tolerance, utilization

- Disadvantages

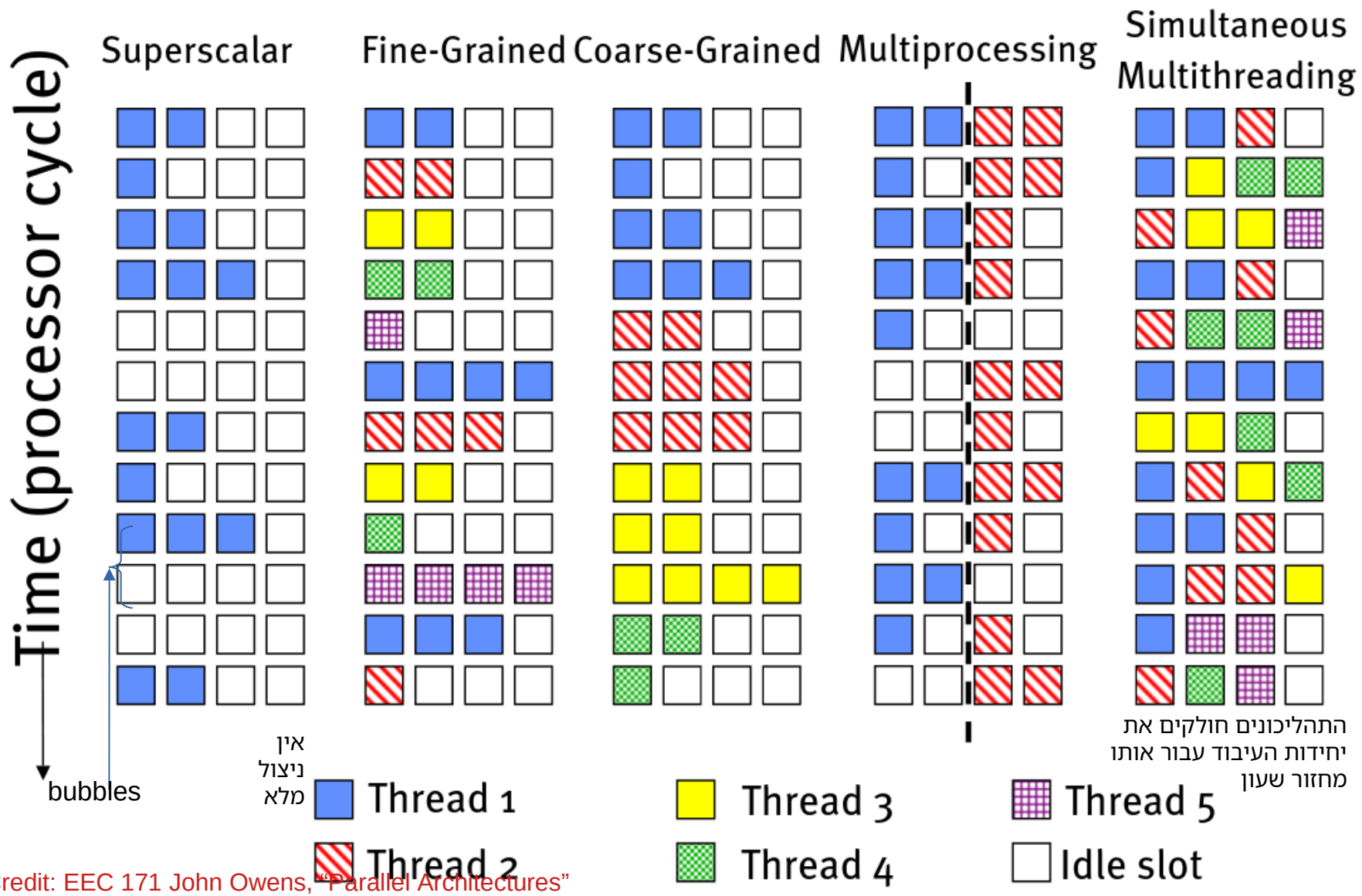
  - **Extra hardware complexity**: multiple hardware contexts (PCs, register files, …), thread selection logic

  - **Reduced single thread performance** (one instruction fetched every N cycles from the same thread)

  - **Resource contention** between threads in caches and memory

  - Some dependency checking logic *between* threads remains (**load/store**)

Multithreaded Categories



Credit: EEC 171 John Owens, "Parallel Architectures"
https://www.nvidia.com/content/cudazone/cudau/courses/ucdavis/lectures/tlp2.pdf

H&P CAQA, 7th Edition

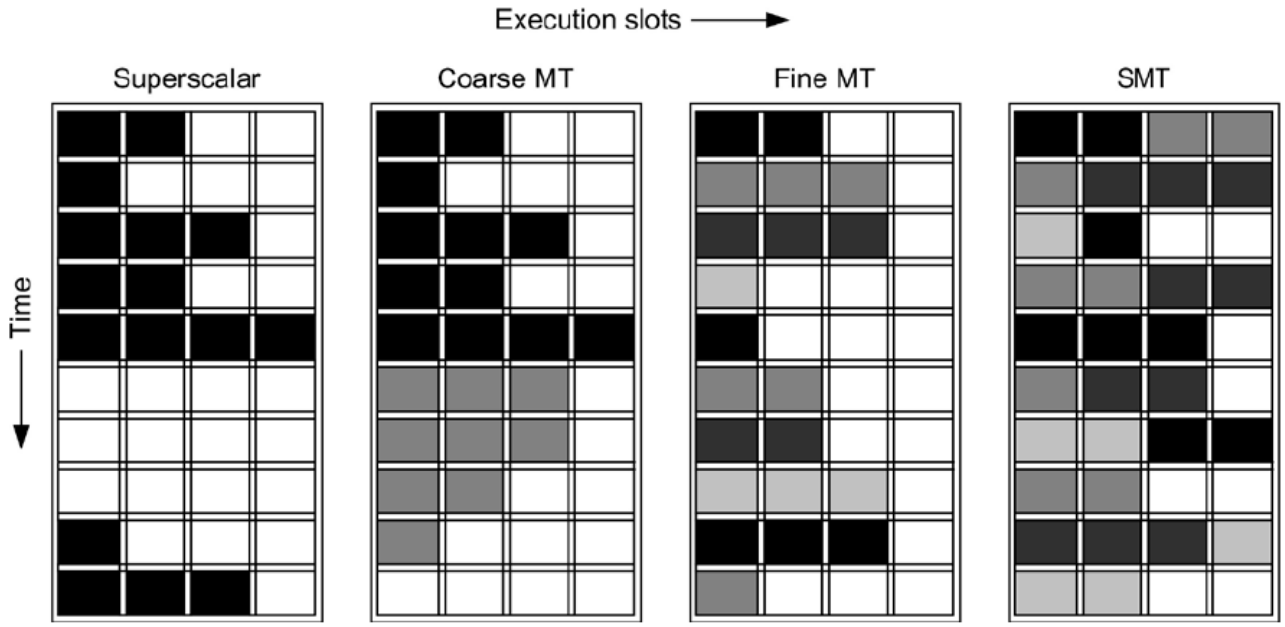
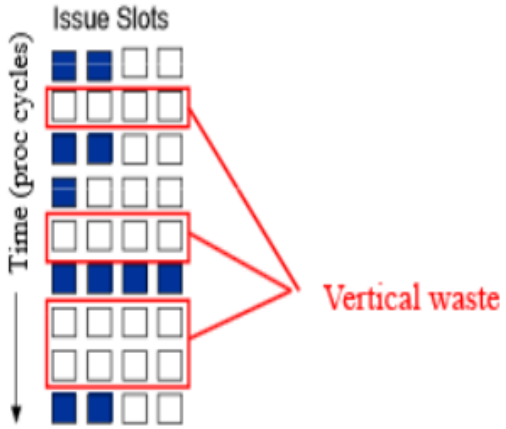
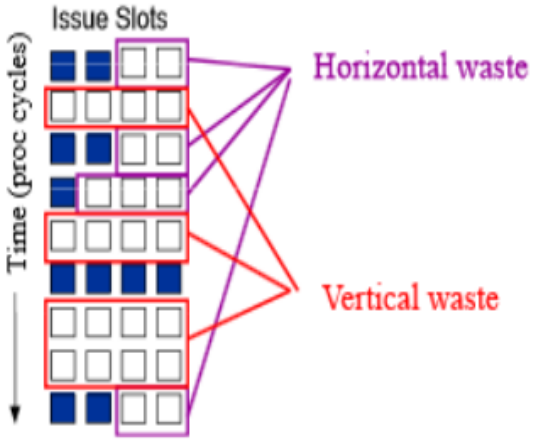


Figure 3.31 How four different approaches use the functional unit execution slots of a superscalar processor. The horizontal dimension represents the instruction execution capability in each clock cycle. The vertical dimension represents a sequence of clock cycles. An empty (white) box indicates that the corresponding execution slot is unused in that clock cycle. The shades of gray and black correspond to four different threads in the multithreading processors. Black is also used to indicate the occupied issue slots in the case of the superscalar without multithreading support. The Sun T1 and T2 (aka Niagara) processors are fine-grained, multithreaded processors, while the Intel Core i7 and IBM Power7 processors use SMT. The T2 has 8 threads, the Power7 has 4, and the Intel i7 has 2. In all existing SMTs, instructions issue from only one thread at a time. The difference in SMT is that the subsequent decision to execute an instruction is decoupled and could execute the operations coming from several different instructions in the same clock cycle.

Superscalar Execution



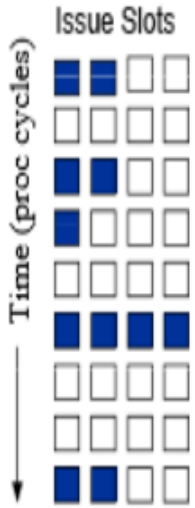
Superscalar Execution



Credit for this slide and the next two slides:

ANKIK JOSHI and DHARMESH TANK, “A FINE-GRAIN MULTITHREADING SUPERSCALAR ARCHITECTURE"

Superscalar Execution

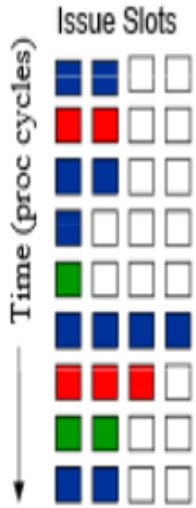


Superscalar Execution with Fine-Grain Multithreading

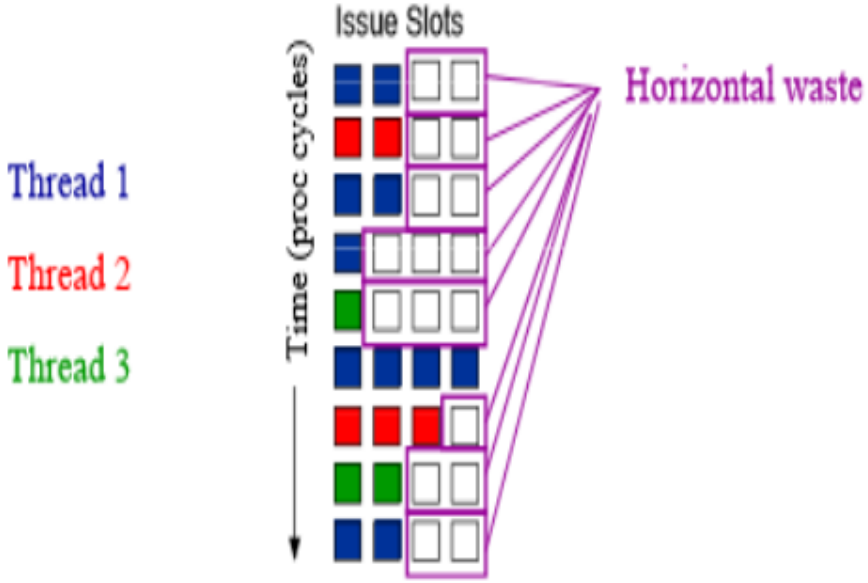
Thread 1

Thread 2

Thread 3



Superscalar Execution with Fine-Grain Multithreading



Simultaneous Multithreading

אנימציה

How to Handle Control Dependences

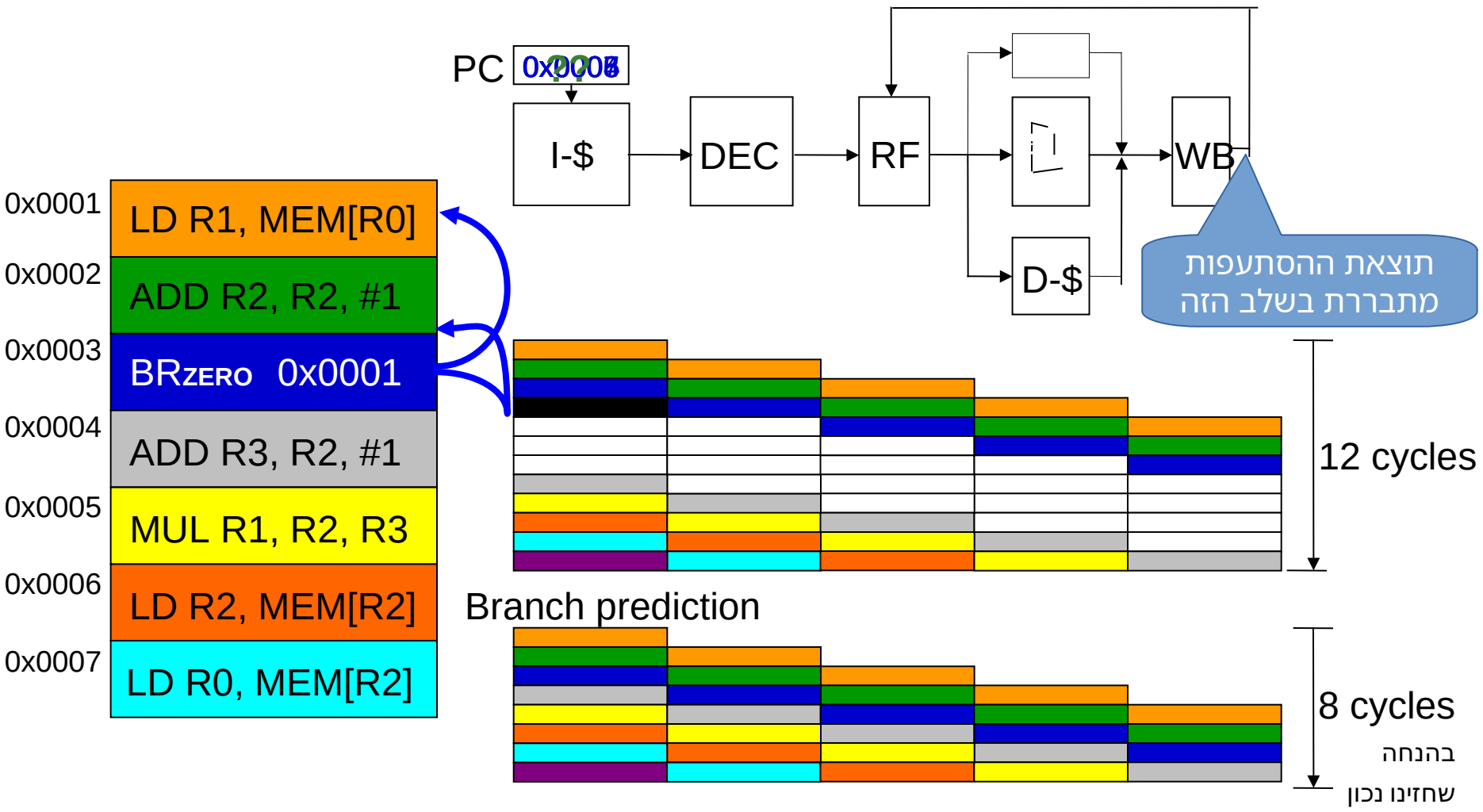
- Critical to keep the pipeline full with correct sequence of dynamic instructions.

- 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)

Branch Prediction

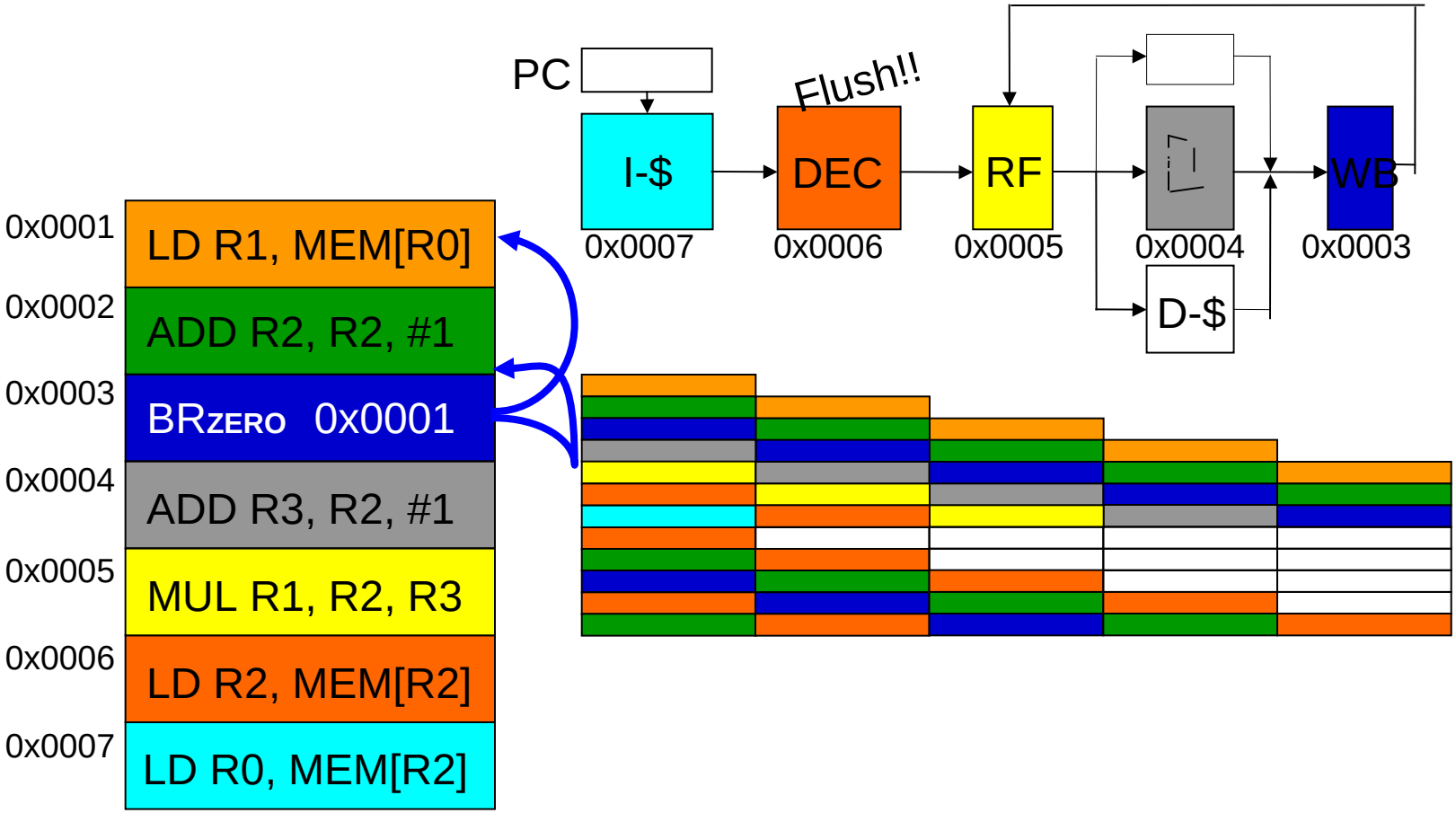
Branch Prediction: Guess the Next Instruction to Fetch

אנימציה



Misprediction Penalty

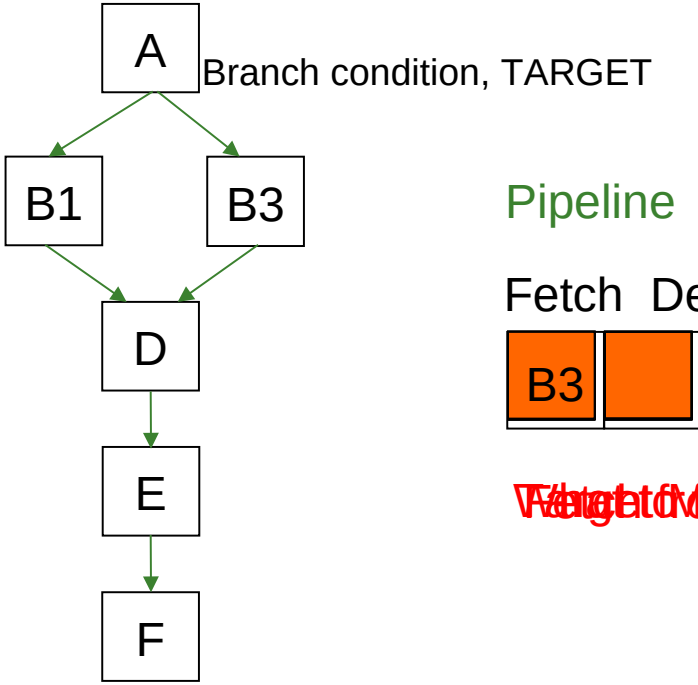
אנימציה



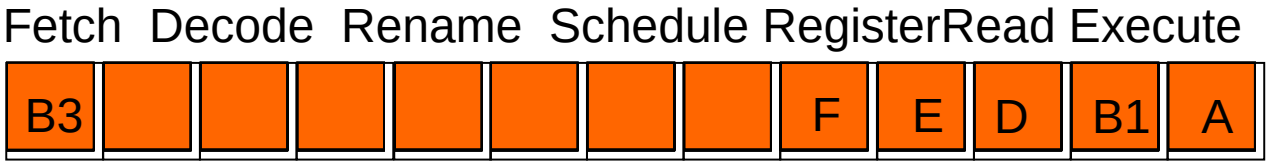
Branch Prediction

אנימציה

- Processors are pipelined to increase concurrency
- How do we keep the pipeline full in the presence of branches?
  - Guess the next instruction when a branch is fetched
  - Requires guessing the direction and target of a branch



Pipeline



Wrong Prediction Detected! Flush the pipeline

Verify the Prediction

בתרשים נראה פייפליין ארוך שמצריך 13 שלבים עד
אשר מתברר כיוון ההסתעפות!

דוגמאות למעבדים בעלי פייפליין עמוק

Basic Pentium III Processor Misprediction Pipeline

1	2	3	4	5	6	7	8	9	10
Fetch	Fetch	Decode	Decode	Decode	Rename	ROB Rd	Rdy/Sch	Dispatch	Exec

Basic Pentium 4 Processor Misprediction Pipeline

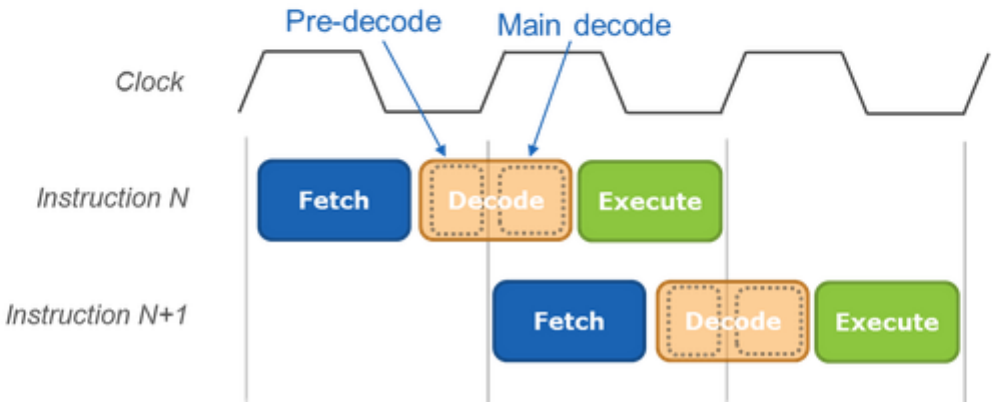
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC Nxt IP		TC Fetch		Drive	Alloc	Rename		Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flgs	Br Ck	Drive

לעומת זאת...

ARM Cortex®-M0+ Pipeline

The ARM® Cortex®-M0+ core has a two-stage pipeline (Cortex-M0, M3 and M4 have three stages). This two-stage pipeline decreases the core response time and power consumption.

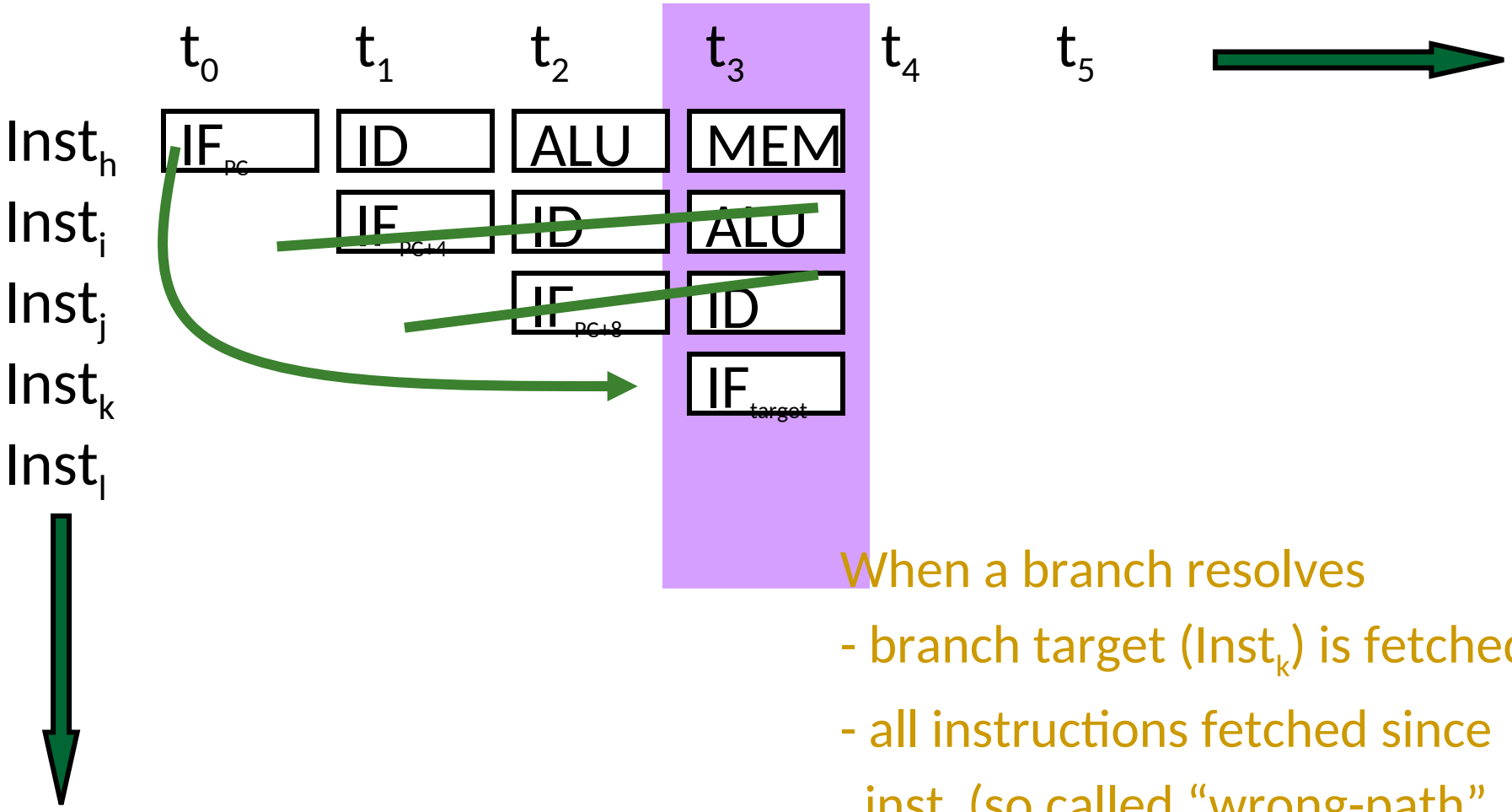
- Stage 1: Fetch & Pre-Decode
- Stage 2: Main Decode & Execute



Credit: https://microchipdeveloper.com/32arm:m0-pipeline

Branch Prediction: Always PC+4

אנימציה



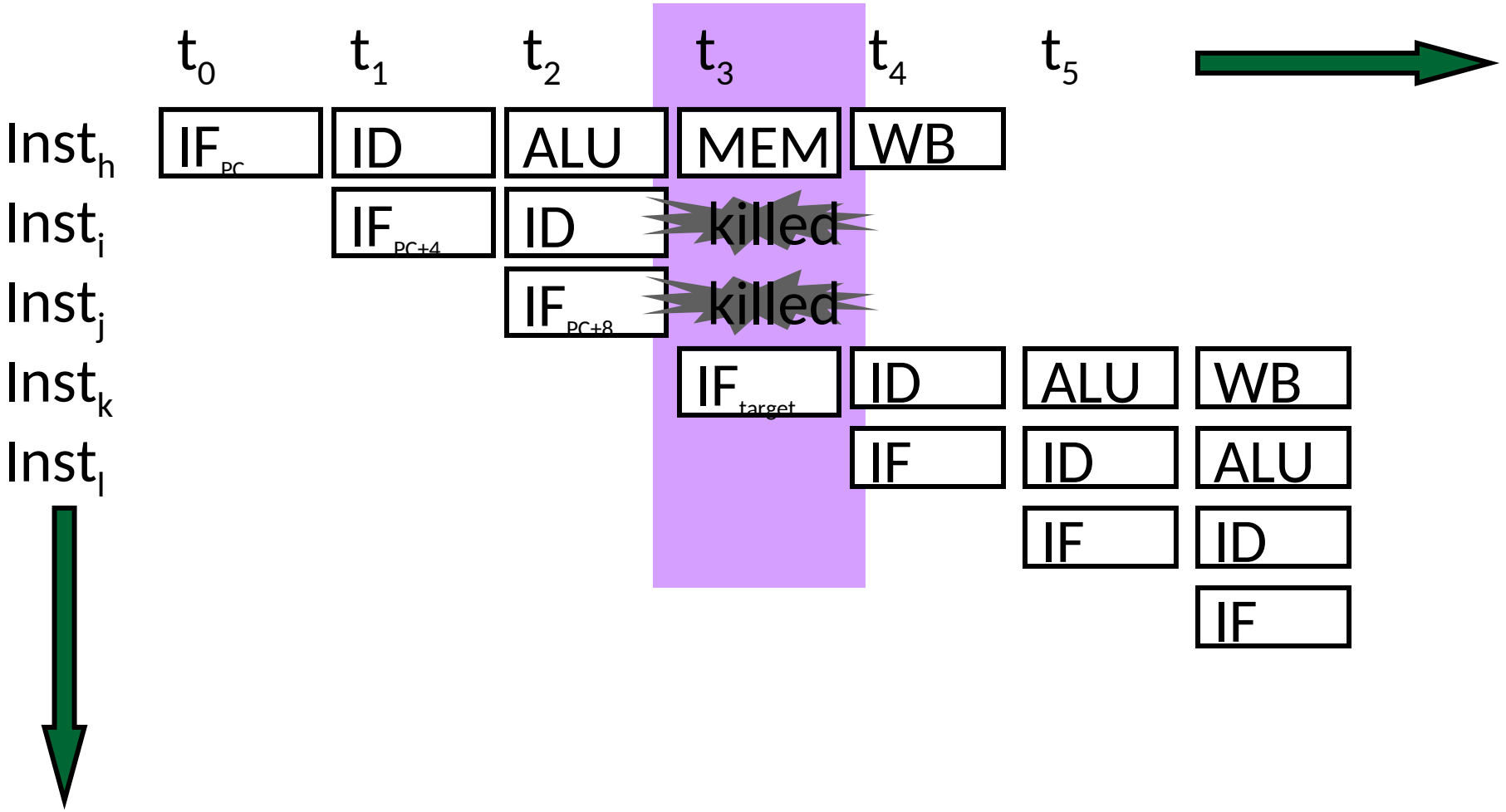
When a branch resolves

- branch target ($Inst_k$) is fetched
- all instructions fetched since $inst_h$ (so called "wrong-path" instructions) must be flushed

$Inst_h$ is a branch

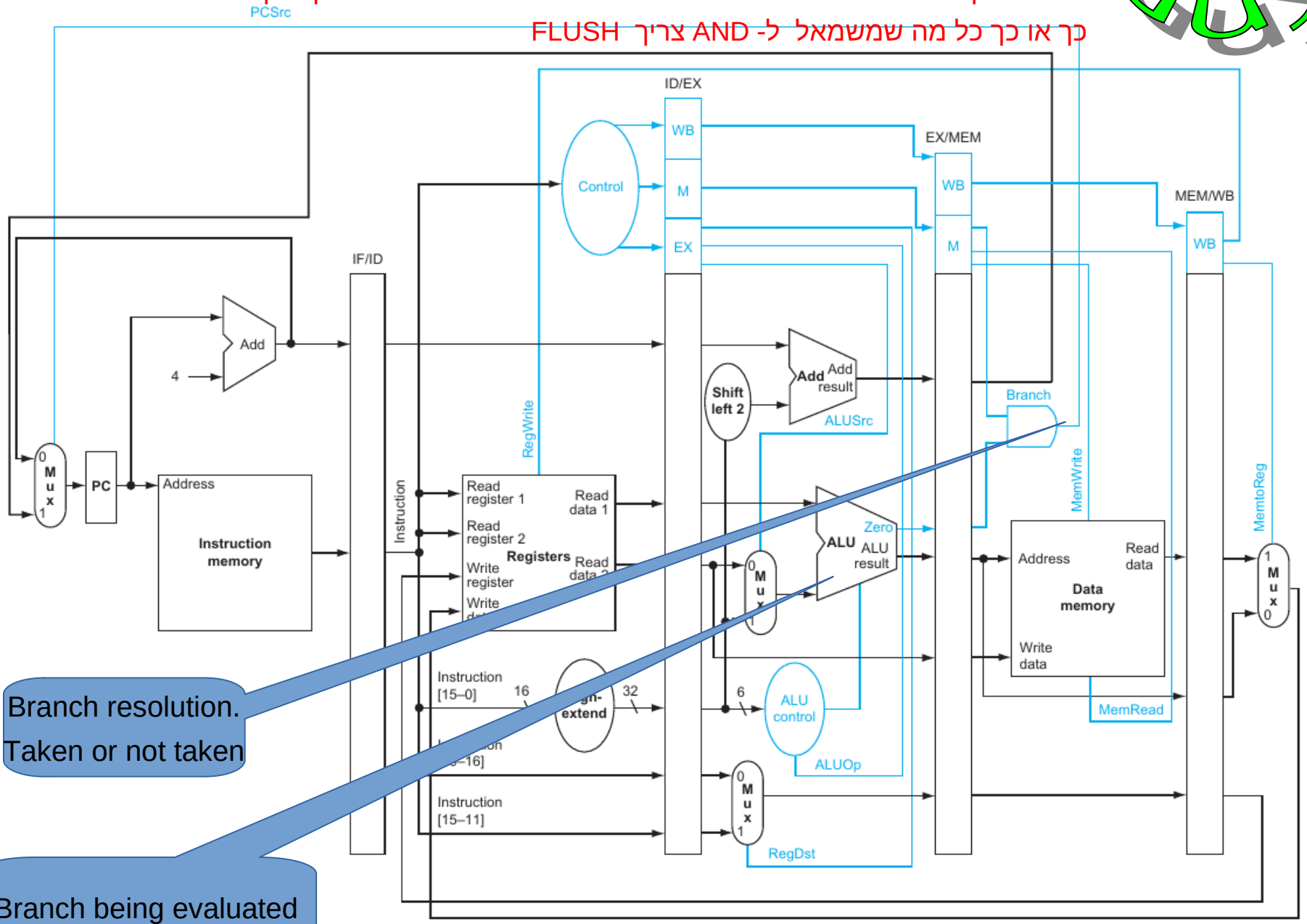
Pipeline Flush on a Misprediction

אנימציה



$Inst_h$ is a branch

הכתובת מחושבת בשלב ה ALU אך שער ה-AND בשלב ה memory קובע אם לעשות branch .
אם הלוגיקה היתה מוזזת דרגה אחת או יותר שמאלה אפשר היה לחסוך זמן.
כך או כך כל מה שמשמאל ל- AND צריך FLUSH



Performance Analysis

- correct guess ⇒ no penalty, 80%+0.2*0.3=86% of the time
- incorrect guess ⇒ 3 bubbles, 14% of the time
- Assume
  - no data dependency related stalls – אין אפקטים אחרים של תלות
  - 20% control flow instructions
  - **Always taken**. 30% correct guesses and 70% are wrong guesses
  - CPI = [1 + (0.20*0.7) * 3] =

    = [1 + 0.14 * 3] = 1.42

probability of a wrong guess

penalty for a wrong guess

המחיר של הטעות בהנתן החלטה TAKEN שלא היתה נכונה

אם הפייפליין עמוק יותר, תרומה זו אף תגדל

לשיפור הביצועים צריך להקטין את **שני** הגורמים

Can we reduce either of the two penalty terms?

Performance Analysis

- correct guess ⇒ no penalty, 80%+0.2*0.3=86% of the time
- incorrect guess ⇒ 2 bubbles, 14% of the time
- Assume
  - no data dependency related stalls – אין אפקטים אחרים של תלות
  - 20% control flow instructions
  - **Always taken**. Of them 70% are wrong guesses
  - CPI = [1 + (0.20*0.7) * 2] =

    = [1 + 0.14 * 2] = 1.28

עתה, המחיר של הטעות בהנתן החלטה TAKEN שלא היתה נכונה

probability of a wrong guess

penalty for a wrong guess

אם הפייפליין עמוק יותר, תרומה זו אף תגדל

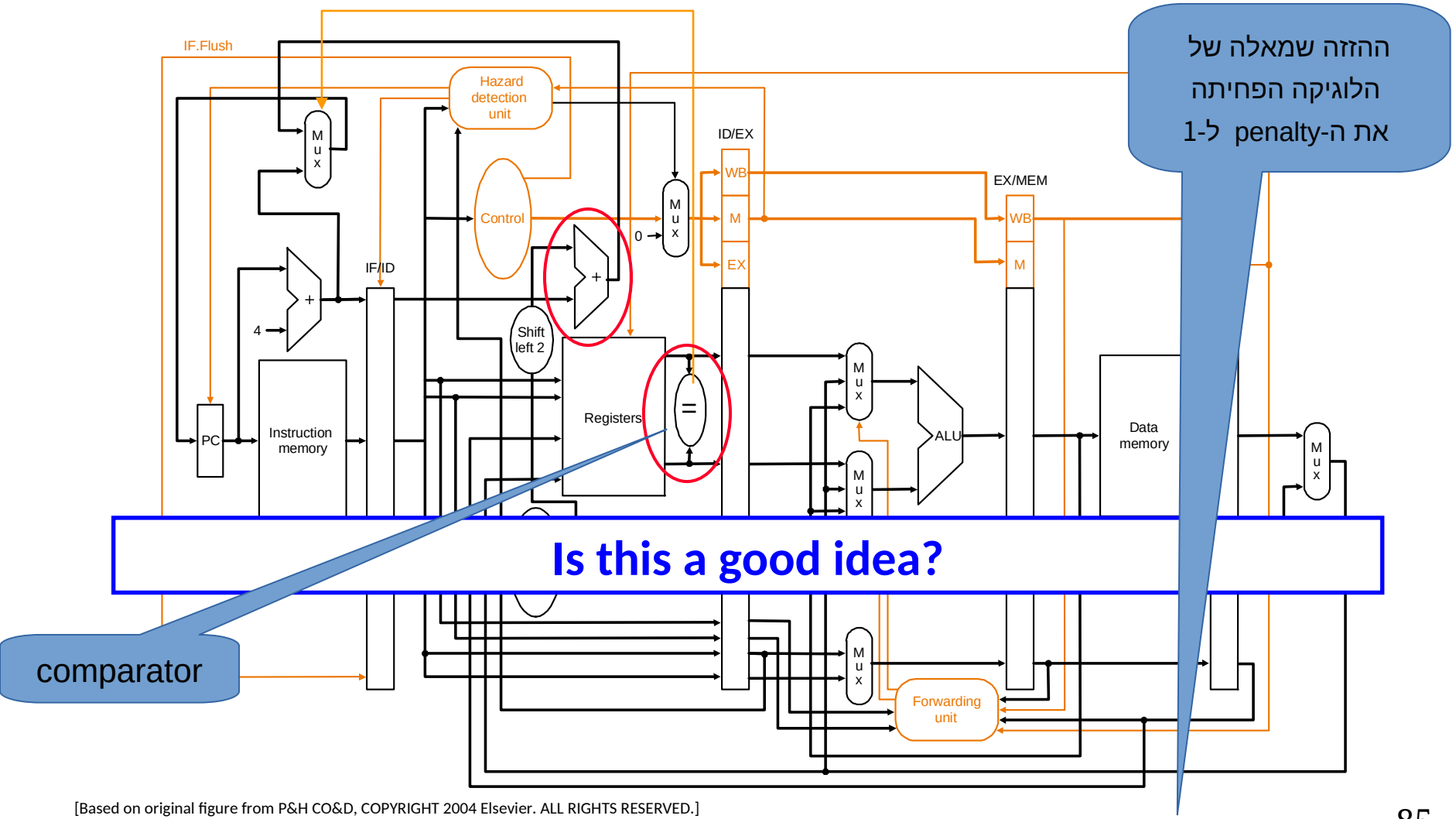
לשיפור הביצועים צריך להקטין את **שני** הגורמים

Can we reduce either of the two penalty terms?

אנימציה

Reducing Branch Misprediction Penalty

- Resolve branch condition and target address early

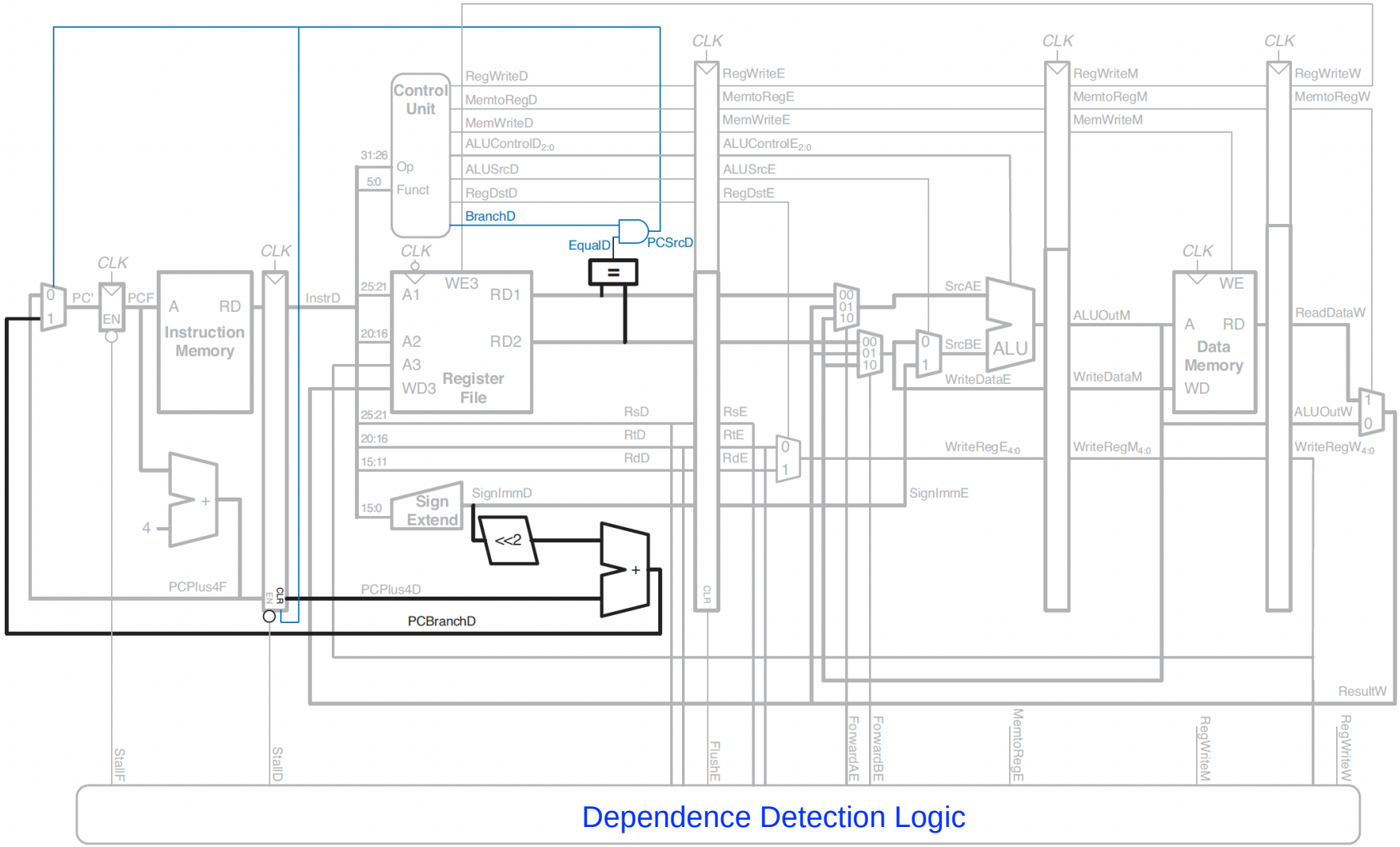


[Based on original figure from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

CPI = [1 + (0.2*0.7) * 1] = 1.14

Recall: Pipeline with Early Branch Resolution

Idea: Calculate branch target and condition in the Decode Stage



Credit: Lecture 14, DDCA Spring 2022 https://www.youtube.com/watch?v=XaW_O9nKPe0&list=PL5Q2soXY2Zi97Ya5DEUpMpO2bbAoaG7c6&index=15

תרגיל לדוגמה

Source: HP CO&D Section 4.8

נתון קטע הקוד הבא. בהנחה שאנחנו מזיזים את הבדיקה של ההסתעפות מקום אחד שמאלה, כמו בתרשים הקודם, וההסתעפות אכן מתרחשת, תארו את המתרחש.

```
36:   sub   x10, x4, x8
40:   beq   x1,  x3, 16   // PC-relative branch
                          // to 40+16*2=72
44:   and   x12, x2, x5
48:   orr   x13, x2, x6
52:   add   x14, x4, x2
56:   sub   x15, x6, x7
      ...
72:   ld    x4, 50(x7)
```

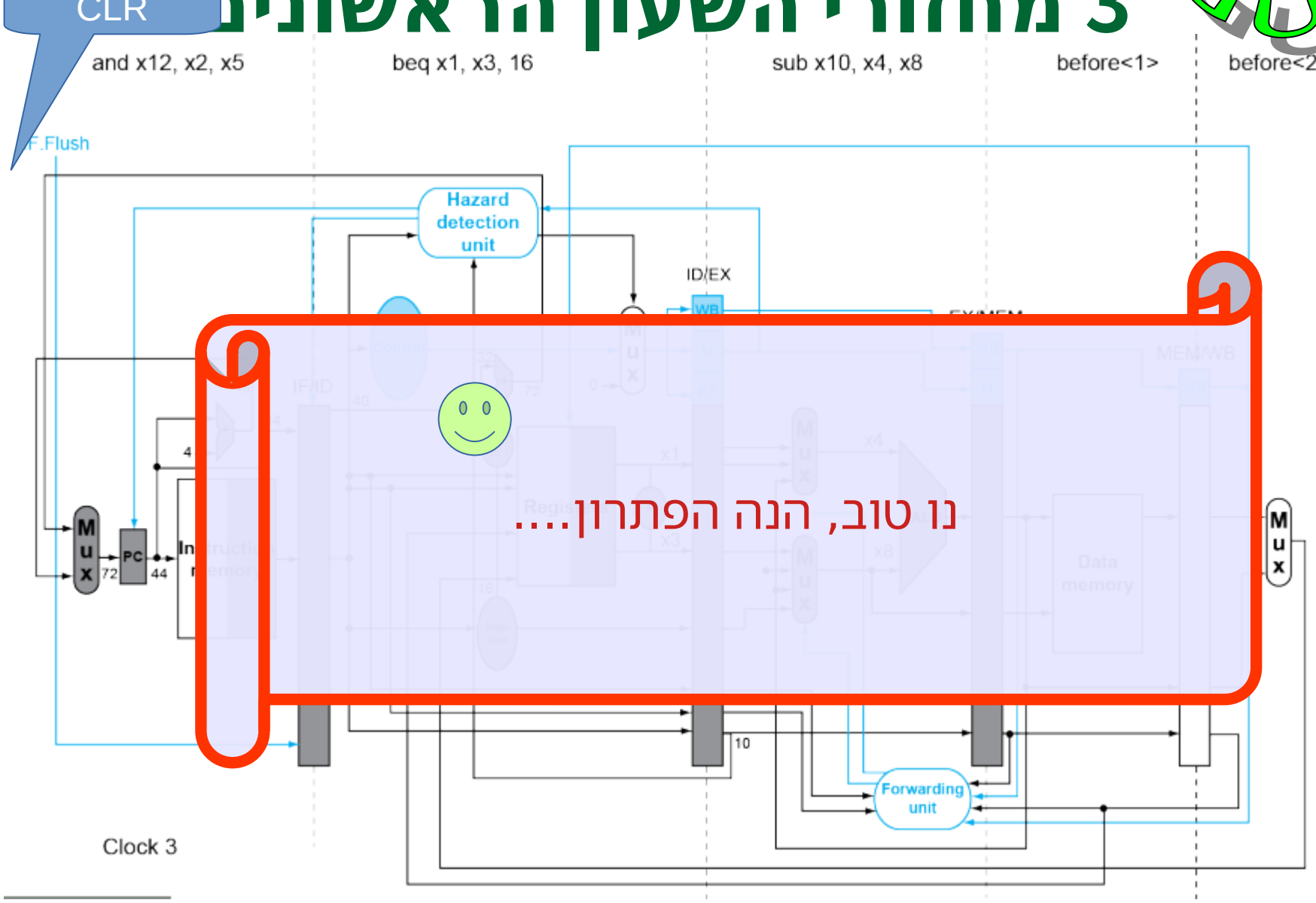
(PC+4) + 7 words =

(PC+4) + 7*4bytes

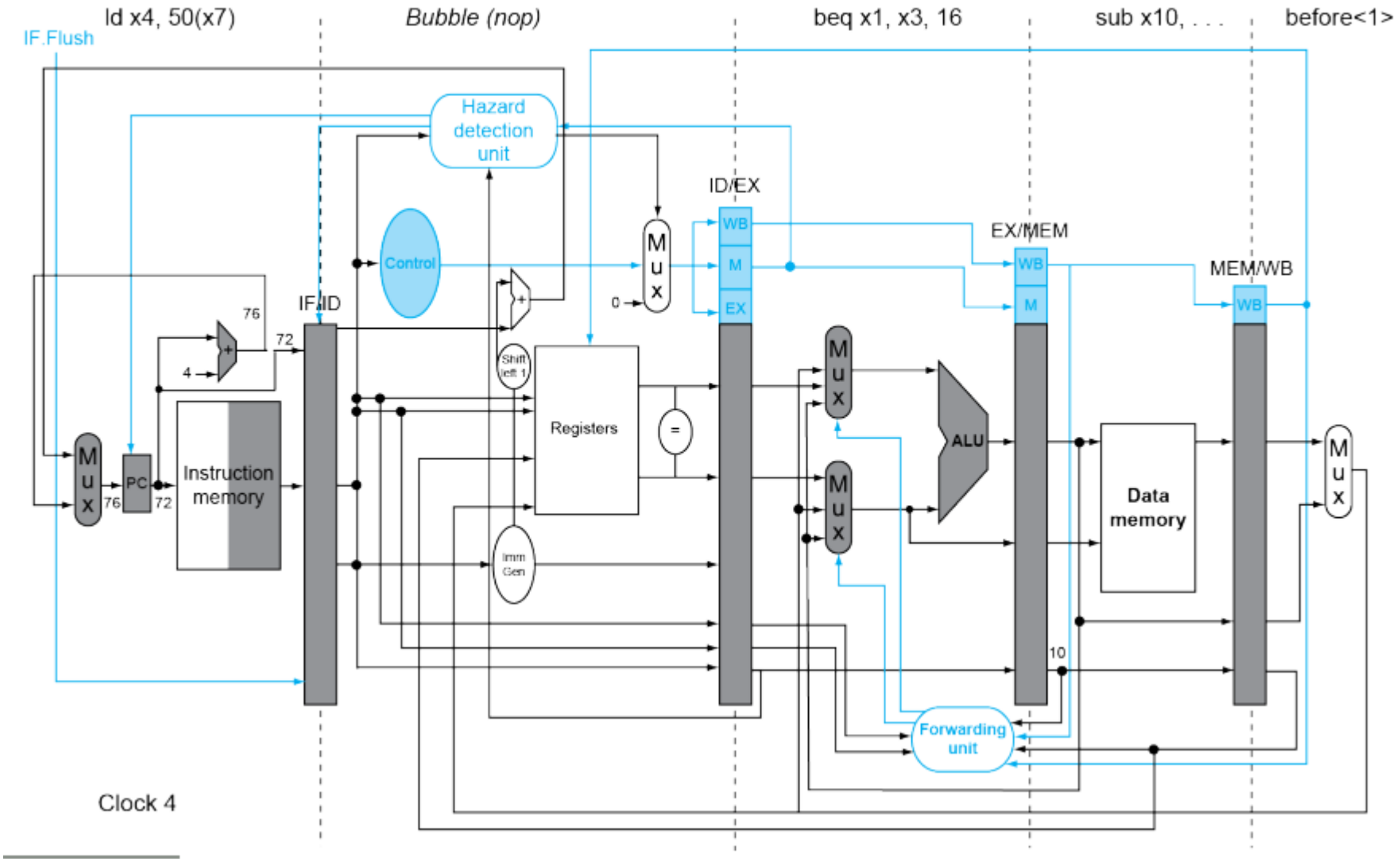
משאיר זאת כתרגיל עבורכם

The processor fetches instruction 44 but it will have to be flushed...

3 מחזורי השעון הראשונים



ההמשך (מחזור 4)



Branch Prediction (Enhanced)

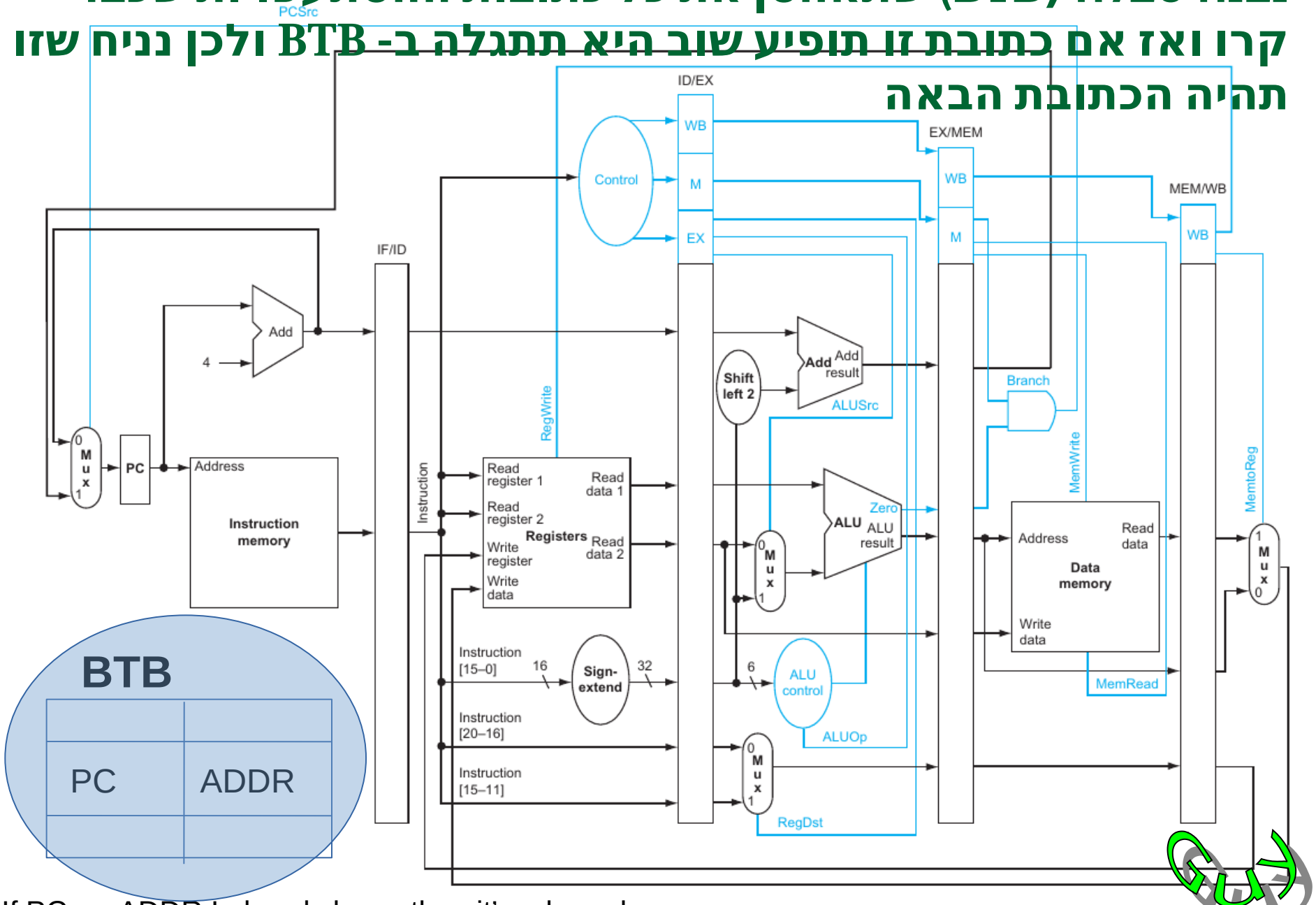
- Idea: Predict the next fetch address (to be used in the next cycle)

  לחזות את הכתובת על-סמך ידע קודם

- Requires three things to be predicted at fetch stage:
  - Whether the fetched instruction is a branch
  - (Conditional) branch direction
  - Branch target address (if taken)

- Observation: Target address remains the same for a conditional direct branch across dynamic instances
  - Idea: Store the target address from previous instance and access it with the PC
  - Called Branch Target Buffer (BTB) or Branch Target Address Cache

נבנה טבלה (BTB) שתאחסן את כל כתובות ההסתעפויות שכבר קרו ואז אם כתובת זו תופיע שוב היא תתגלה ב- BTB ולכן נניח שזו תהיה הכתובת הבאה



If PC == ADDR I already know then it's a branch

Credit: H&P CAQA 7th Edition (RISC-V)

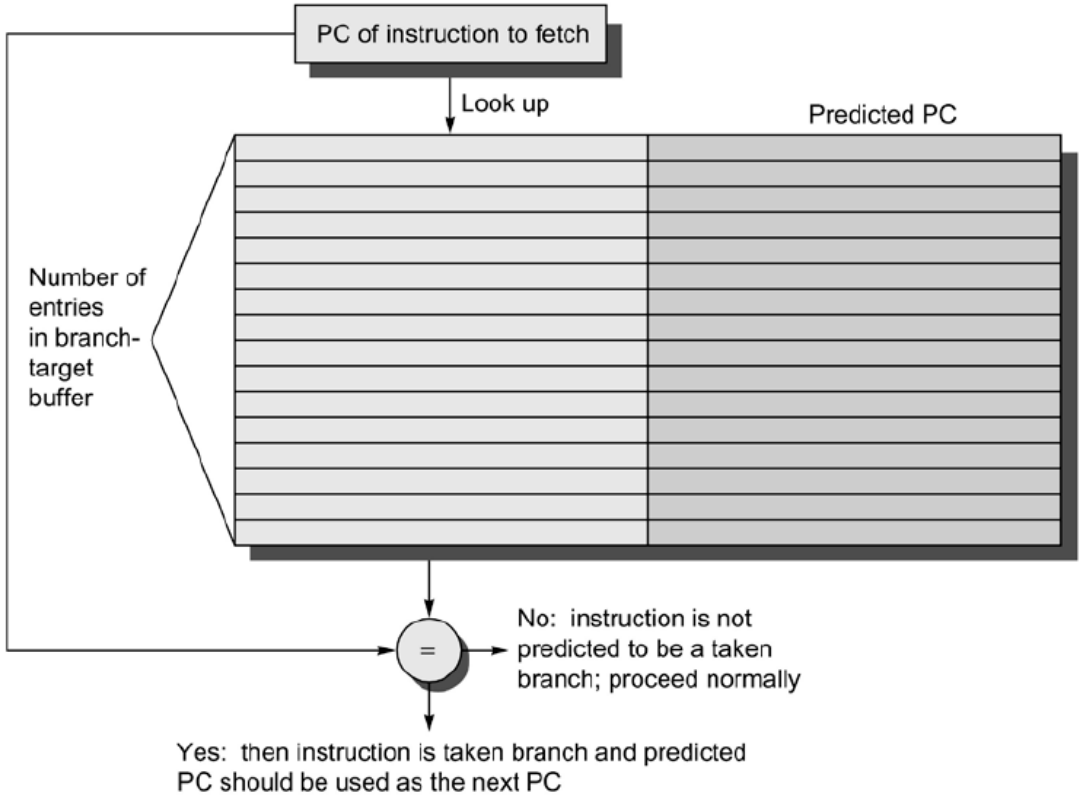
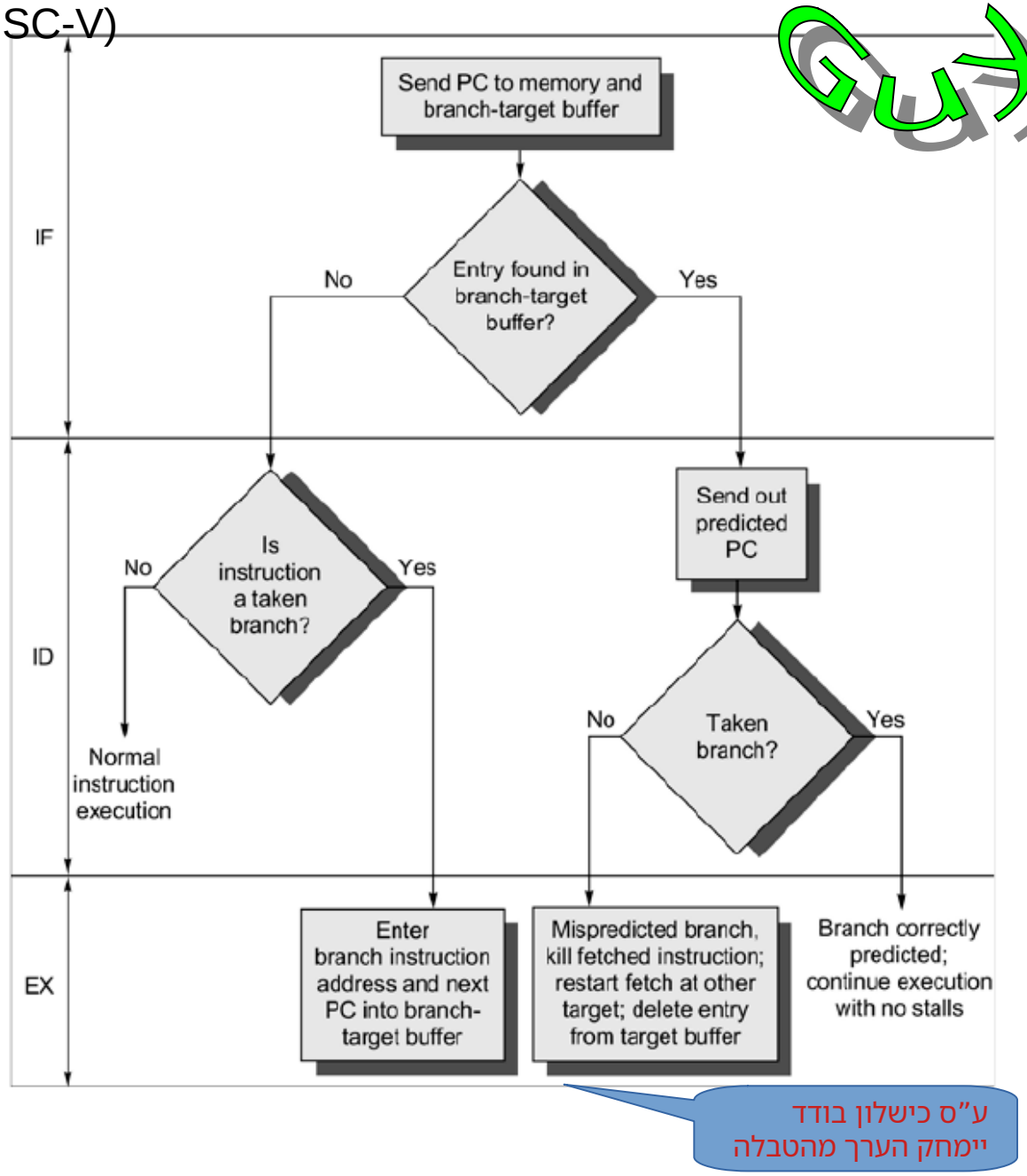


Figure 3.25 A branch-target buffer. The PC of the instruction being fetched is matched against a set of instruction addresses stored in the first column; these represent the addresses of known branches. If the PC matches one of these entries, then the instruction being fetched is a taken branch, and the second field, predicted PC, contains the prediction for the next PC after the branch. Fetching begins immediately at that address. The third field, which is optional, may be used for extra prediction state bits.

Credit: H&P CAQA 7th Edition (RISC-V)



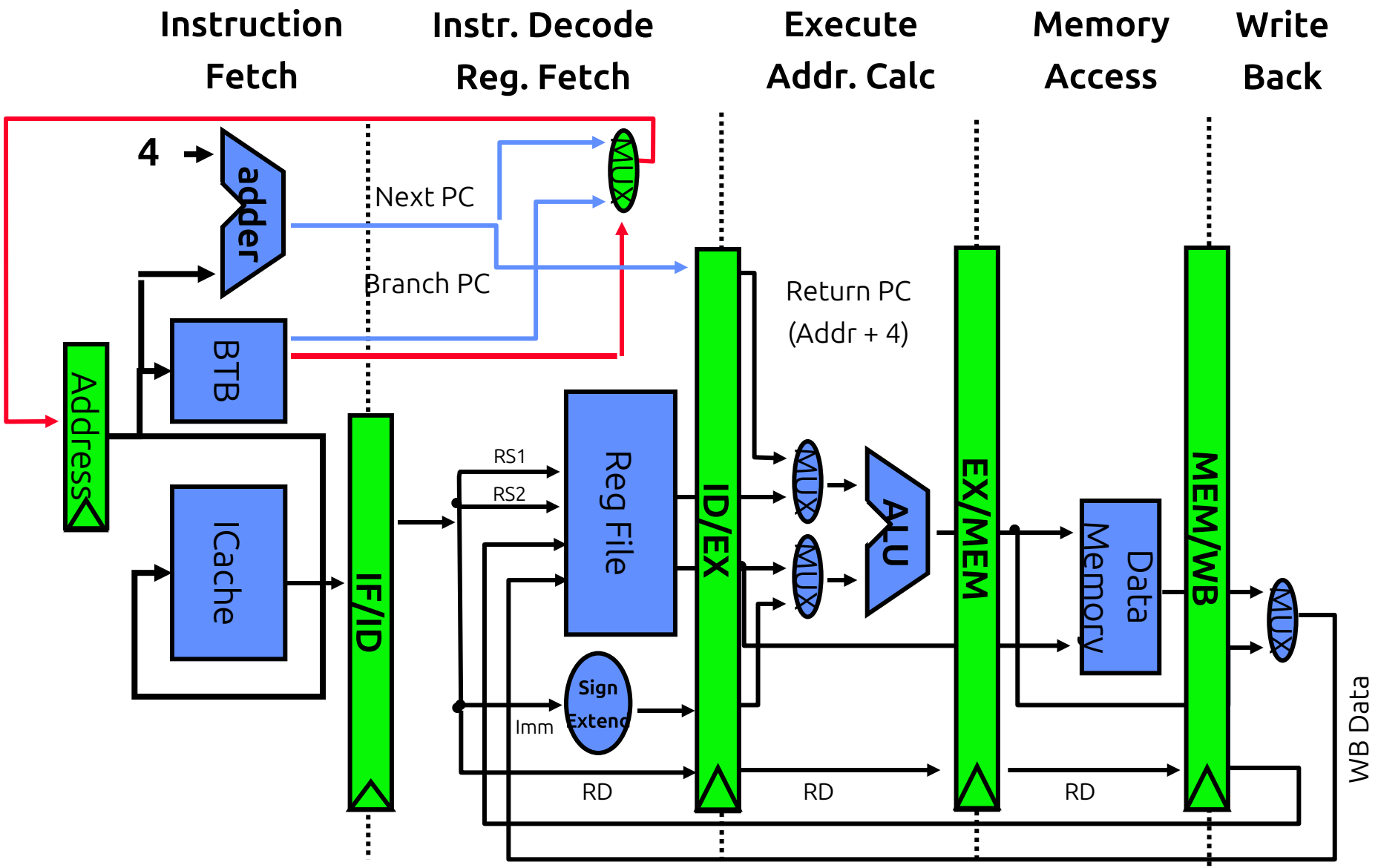
ע"ס כישלון בודד
יימחק הערך מהטבלה

Figure 3.26 The steps involved in handling an instruction with a branch-target buffer.

Credit: H&P CAQA 7th Edition (RISC-V)

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

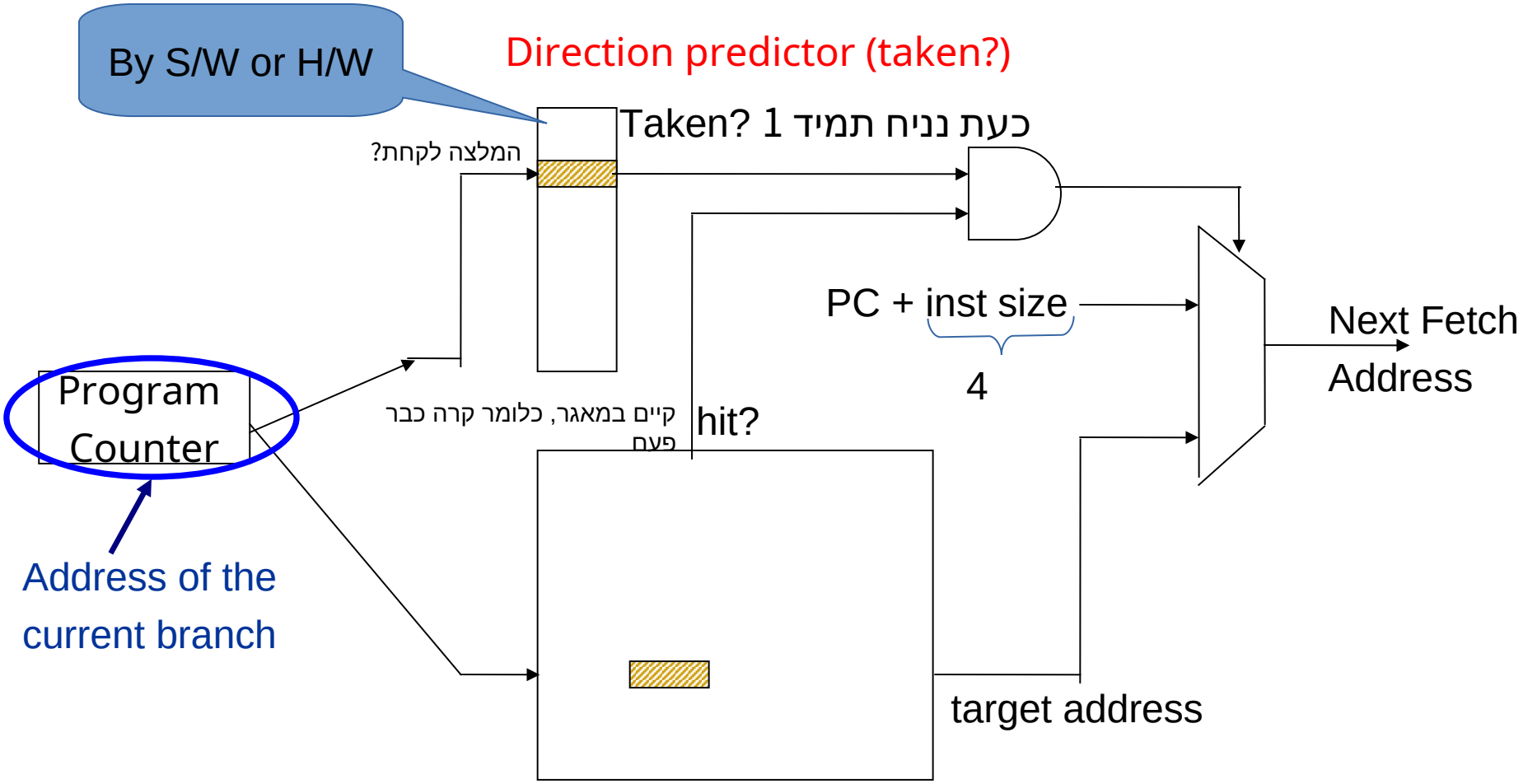
Figure 3.27 Penalties for all possible combinations of whether the branch is in the buffer and what it actually does, assuming we store only taken branches in the buffer. There is no branch penalty if everything is correctly predicted and the branch is found in the target buffer. If the branch is not correctly predicted, the penalty is equal to 1 clock cycle to update the buffer with the correct information (during which an instruction cannot be fetched) and 1 clock cycle, if needed, to restart fetching the next correct instruction for the branch. If the branch is not found and taken, a 2-cycle penalty is encountered, during which time the buffer is updated.



- BTB target & direction predictions
- Note BTB smaller (faster) than ICache, shortens path

Credit:
Christopher Carothers
Computer Architecture
CS4250

Fetch Stage with BTB and Direction Prediction

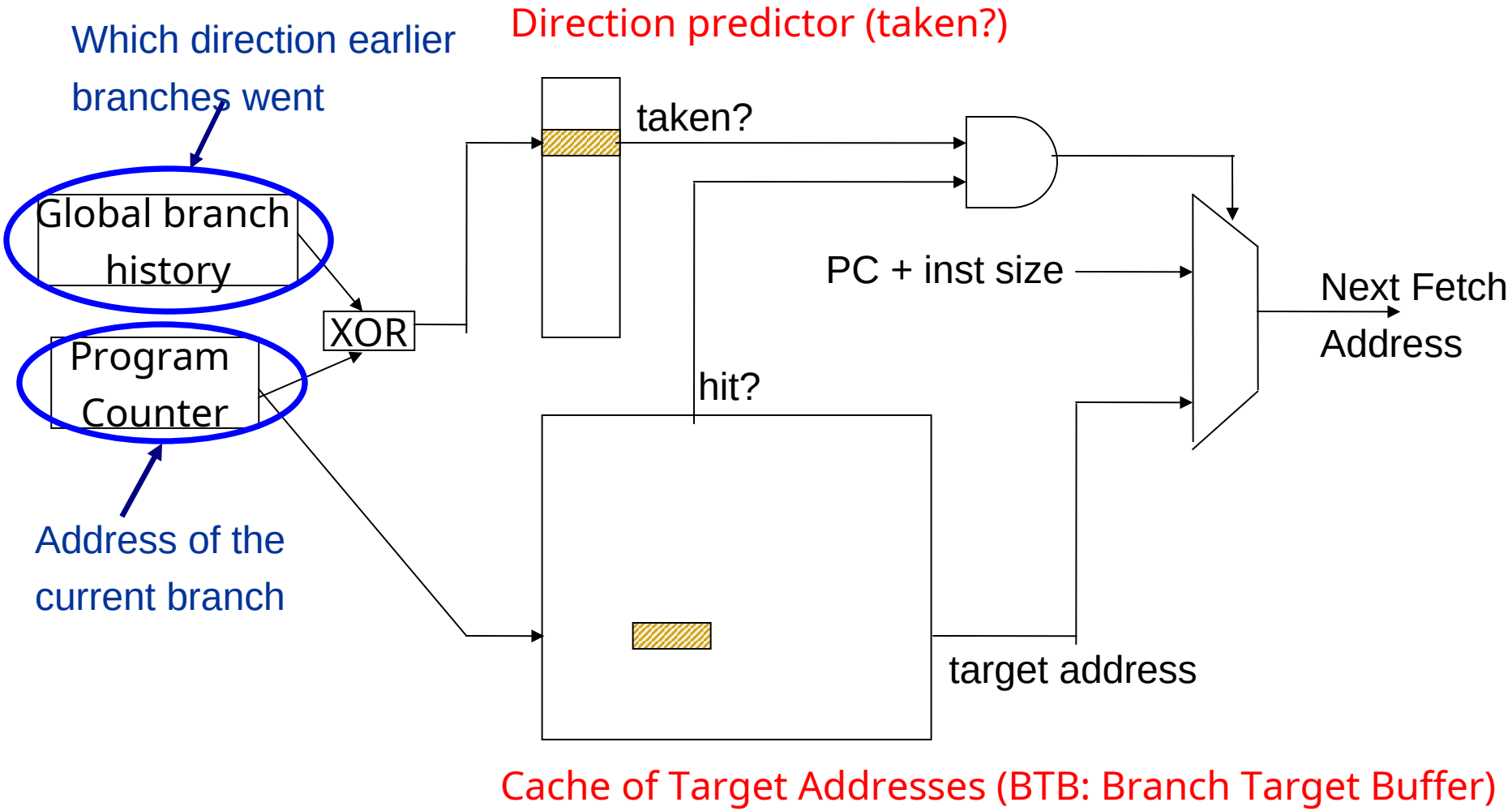


אם נניח בשלב זה taken=1 תמיד
שכיחות הסתעפות

Always taken CPI = [1 + (0.20*0.3) * 2] = 1.12 (70% of branches taken)

שימו לב, ההנחה כאן הפוכה להנחה בדוגמה המספרית הקודמת. 70% מההסתעפויות נלקחות, אך אני מניח "תמיד נלקח" ולכן אטעה ב-30% מהמקרים

More Sophisticated Branch Direction Prediction



סימוכין: המאמר של McFarling

Three Things to Be Predicted

- Requires three things to be predicted at fetch stage:
  1. Whether the fetched instruction is a branch
  2. (Conditional) branch direction
  3. Branch target address (if taken)

על המעבד להיות מסוגל לענות על 3 השאלות:

1) האם לפנינו הוראת הסתעפות?

2) אם כן, האם היא נלקחת?

3) אם היא נלקחת, אז לאיזו כתובת להסתעף?

- Third (3.) can be accomplished using a BTB
  - Remember target address computed last time branch was executed
- First (1.) can be accomplished using a BTB
  - If BTB provides a target address for the program counter, then it must be a branch
  - Or, we can store "branch metadata" bits in instruction cache/memory → partially decoded instruction stored in I-cache
- *Second (2.):* ***How do we predict the direction?***

Simple Branch Direction Prediction Schemes

- Compile time prediction (static)
  - Always not taken
  - Always taken
  - BTFN (Backward taken, forward not taken)
  - Profile based (likely direction)

- Run time prediction (dynamic)
  - Last time prediction (single-bit)

More Sophisticated Direction Prediction

- Compile time prediction (static)
  - Always not taken
  - Always taken
  - BTFN (Backward taken, forward not taken)
  - Profile based (likely direction)
  - Program analysis based (likely direction)

השקפים הבאים

- Run time prediction (dynamic)
  - Last time prediction (single-bit)
  - Two-bit counter based prediction
  - Two-level prediction (global vs. local)
  - Hybrid

לאחר מכן נדון בכל אלה -

Static Branch Prediction (I)

- Always not-taken
  - Simple to implement: no need for BTB, no direction prediction
  - Low accuracy: ~30-40% (for conditional branches)
  - Remember: Compiler can layout code such that the likely path is the "not-taken" path → more effective prediction
- Always taken
  - No direction prediction
  - Better accuracy: ~60-70% (for conditional branches)
    - Backward branches (i.e. loop branches) are usually taken
    - Backward branch: target address lower than branch PC
- Backward taken, forward not taken (BTFN)
  - Predict backward (loop) branches as taken, others not-taken

Static Branch Prediction (II)

- Profile-based גישה יותר מתוחכמת
  - Idea: Compiler determines likely direction for each branch using a profile run. Encodes that direction as a ***hint bit*** in the branch instruction format.

קודם לכן, ההחלטה T או N הייתה גורפת לכל ההסתעפויות

+ Per branch prediction (more accurate than schemes in previous slide) → accurate if profile is representative!

-- Requires ***hint bits*** in the branch instruction format

-- Accuracy depends on dynamic branch behavior:

TTTTTTTTTTNNNNNNNNNN → 50% accuracy
TNTNTNTNTNTNTNTNTNTN → 50% accuracy

-- Accuracy depends on the representativeness of profile input set

Itanium (IA-64): The Intel Itanium architecture (IA-64), based on the Explicitly Parallel Instruction Computing (EPIC) paradigm, heavily relied on compiler cooperation for performance. It had explicit branch hints in its instruction set. The compiler was responsible for providing these hints to the hardware, guiding the branch predictor. This was a core part of its design philosophy to expose more hardware details to the compiler.

x86 (with nuances): The x86 architecture has a more complex history with explicit branch hints.

Early Intel Processors (e.g., NetBurst architecture like Pentium 4): Some older Intel architectures did indeed respect specific instruction prefixes (0x2E for "Branch Not Taken" and 0x3E for "Branch Taken") as branch hints.

Later Intel Processors (e.g., **Core 2 to Broadwell/Skylake**): For a long time, these hints were largely considered "no-ops" by compilers and processors, as dynamic branch predictors became highly sophisticated and often out-performed static hints. The processors would primarily rely on their internal dynamic predictors and static heuristics (like "backward taken, forward not taken").

Newer Intel Processors (e.g., **Redwood Cove P-cores in Meteor Lake** and later): Interestingly, Intel has reintroduced the utility of these branch hint prefixes (0x3E for "Branch Taken" hint, specifically) for certain scenarios. These hints are now used by the processor when its internal branch predictor has no stored information about a particular branch. This can be particularly useful for infrequently executed branches that are almost always taken, preventing an initial misprediction. Compilers like GCC and LLVM/Clang have added support for emitting these hints based on profile-guided optimization.

שקף ממלא מקום – פה תבוא הדגמה בעתיד...

Static Branch Prediction (III)

- Program-based (or, program analysis based)
  - Idea: Use **heuristics** based on program analysis to determine statically-predicted direction

במקום פרופיילינג הקומפיילר עושה היוריסטיקה

  - Example opcode heuristic: Predict BLEZ as NT (*negative integers used as error values in many programs*)

מאחר ורוב הסיכויים הם שזה לא יקרה...

  - Example loop heuristic: Predict a branch guarding a loop execution as taken (i.e., execute the loop)

באופן דומה, בלולאות אנחנו יודעים לחזות את הכיוון ברוב הפעמים.

דוגמה נוספת: בבעיות חיפוש אנחנו עושים השוואות בין פוינטרים. לרוב הן תהינה NT ורק כשנגיע לתוצאה יהיה T

+ Does not require profiling יתרון

-- Heuristics might be not representative or good חסרון

-- Requires compiler analysis and ISA support (ditto - כנ"ל for other static methods)

- Ball and Larus, "Branch prediction for free," PLDI 1993.
  - 20% misprediction rate

Static Branch Prediction (IV)

- Programmer-based
  - Idea: Programmer provides the statically-predicted direction
  - Via *pragmas* in the programming language that qualify a branch as likely-taken versus likely-not-taken

לדוגמה ראו המאמר:

Compiler-assisted Source-to-Source Skeletonization of Application Models for System Simulation

https://www.osti.gov/servlets/purl/1513073

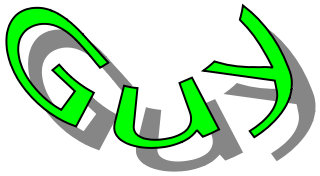
+ Does not require profiling or program analysis

+ Programmer may know some branches and their program better than other analysis techniques

-- Requires programming language, compiler, ISA support

-- Burdens the programmer?

Pragmas



- Idea: Keywords that enable a programmer to convey hints to lower levels of the transformation hierarchy

גיא: להראות דוגמאות

- if (likely(x)) { … }
- if (unlikely(error)) { … }

דוגמה 1 - מופיעה בשקף הבא.

דוגמה 2

- Many other hints and optimizations can be enabled with pragmas
  - E.g., whether a loop can be parallelized
  - **#pragma omp parallel**
  - **Description**
    - The omp parallel directive explicitly instructs the compiler to parallelize the chosen segment of code.

In gcc: **`-fguess-branch-probability`** is enabled by default at -O1 and higher.

```
int f(int i) {
    switch(i) {
    case 1:
[[fallthrough]];
    [[likely]] case 2:
return 1;
    }
    return 2;
}
```

```
int f(int n) {
    if (n > 5) [[unlikely]] {
        g(0);
        return n * 2 + 1;
    }
    return 3;
}
```

C++20

Document number: P0479R5
Date: 2018-03-16
Audience: Core Working Group, SG14
Reply-to: Clay Trychta <clay.trychta@gmail.com>

Proposed wording for likely and unlikely attributes (Revision 5)

Wording

10.6.N Likelihood attributes [dcl.attr.likelihood]

The *attribute-tokens* **likely** and **unlikely** may be applied to labels or statements. The *attribute-tokens* **likely** and **unlikely** sh shall not appear in an *attribute-specifier-seq* that contains the *attribute-token* **unlikely**.

[*Note:* The use of the **likely** attribute is intended to allow implementations to optimize for the case where paths of execution includir or label. The use of the **unlikely** attribute is intended to allow implementations to optimize for the case where paths of execution inc statement or label. A path of execution includes a label if and only if it contains a jump to that label. Excessive usage of either of these

Acknowledgements

Thank you to Jens Maurer, Chandler Carruth, Andrew Pardoe, Davis Herring, John McFarlane, Bartosz Bielecki, Carl Cook, Matt Dziubinski

מתוך התיעוד של הקומפיילר gcc

3.11 Options That Control Optimization

These options control various sorts of optimizations.

Without any optimization option, the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Statement statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the re

Turning on optimization flags makes the compiler attempt to improve the performance and/or code size at the expense of compilation time and possibly t

The compiler performs optimization based on the knowledge it has of the program. Compiling multiple files at once to a single output file mode allows th of them.

Not all optimizations are controlled directly by a flag. Only optimizations that have a flag are listed in this section.

Most optimizations are completely disabled at `-O0` or if an `-O` level is not set on the command line, even if individual optimization flags are specified. Simila

Depending on the target and how GCC was configured, a slightly different set of optimizations may be enabled at each `-O` level than those listed here. You optimizations that are enabled at each level. See Overall Options, for examples.

`-O`
`-O1`

Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function.

With `-O`, the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.

`-O` turns on the following optimization flags:

```
-fauto-inc-dec
-fbranch-count-reg
-fcombine-stack-adjustments
-fcompare-elim
-fcprop-registers
-fdce
-fdefer-pop
-fdelayed-branch
-fdse
-fforward-propagate
-fguess-branch-probability
-fif-conversion
-fif-conversion2
```

https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Static Branch Prediction

- All previous techniques can be combined
  - Profile based - פרופיילינג
  - Program based – התכנית מניחה
  - Programmer based – היוריסטיקה של המתכנת
- How would you do that? ניתן לאחד את 3 השיטות
- What is the common disadvantage of all three techniques?
  - Cannot adapt to dynamic changes in branch behavior
    - This can be mitigated by a **dynamic** compiler, but not at a fine granularity (and a dynamic compiler has its overheads…)
    - What is a Dynamic Compiler?
      - Remember Transmeta? Code Morphing Software?
      - Java JIT (just in time) compiler, Microsoft CLR (common lang. runtime)

* הדגמות

Note to self:

Demos folders:

~/science/Teaching/CPU/lectures/09/code

*** Profiling:**

/home/telzur/science/Teaching/CPU/lectures/09/code/profiling

*** loop unrolling(**):**

/home/telzur/science/Teaching/CPU/lectures/09/code/loop_unrolling

*** Predicate:**

~/science/Teaching/CPU/lectures/09/code/predicate_exe

*** likely/unlikely:**

~/science/Teaching/CPU/lectures/09/code/branch_prediction2

* ההדגמות הללו אינן מושלמות... אבל מהוות חומר למחשבה ולאתגר

הדגמה לפרופיילינג

Note to self:

Folder:

~/science/Teaching/CPU/lectures/09/code/profiling/May2025

סטטוס ההדגמה: בפיתוח, לא סופי.
תלמידים: אפשר להתעלם ההדגמה

הדגמה לפרופיילינג

Note to self:

Folder: ~/science/Teaching/CPU/lectures/09/code/profiling/May2025

See the two folders: A) Instrumented, B) Optimized

```
echo 0 | sudo tee /proc/sys/kernel/perf_event_paranoid
rm *.gcda
gcc -O2 -g -fprofile-generate -o test_profile1.o -c test_profile1.c -lm
gcc -g -fprofile-generate -o test_profile1 test_profile1.o -lm
./test_profile1
gcc -g -O3 -fprofile-use -o test_profile1.o -c test_profile1.c -lm
gcc -g -O2 -fprofile-generate -o test_profile1.o -c test_profile1.c -lm
gcc -g -fprofile-generate -o my_program_instrumented test_profile1.o -lm
./my_program_instrumented # Run with your typical workload
gcc -g -O3 -fprofile-use -o test_profile1.o -c test_profile1.c -lm
# For linking:
gcc -g -fprofile-use -o my_program_optimized test_profile1.o -lm
./my_program_optimized
perf record -e branches,branch-misses ./my_program_optimized
sudo perf test
sudo perf list
sudo perf record -e branches,branch-misses -- ./my_program_optimized
# select the "cpu_core/branch-misses/" and then click a (for annotate)
sudo perf report
see:
./instrumented
./optimized
two profilings: the instrumented contains many more braches and mis-predicted branches
```

Demo #1:

Profiling

```
#include<math.h>
#include<stdlib.h>
#include<stdio.h>
int NMAX=100;
   float func1() {
   int i;
   float x = 0;
   for (i=0;i<NMAX;i++) {
      x+=sin((float)i/(float)NMAX); }
   return x+10;    }
   float func2() {
   int i;
   float x =0;
   for (i=0;i<NMAX;i++) {
      x+=sin((float)i/(float)NMAX); }
   return x;     }
int main() {
   float a,b,c,d,e,f,g,h;
   float x;
   int N = 100;
   a = func1();
   b = func2();
   for (int i=0;i<N;i++) {
         x = (float)rand()/(float)RAND_MAX;
   if (a==b)
    { c = func1(); printf("computing c\n"); }
   else if (a>b+20+x)
    { d = func1(); printf("computing d\n"); }
   else if (a>b+17+x)
            { e = func1(); printf("computing e\n"); }
         else if (a>b+9.5+x) // should somewhat be here
            { f = func1(); printf("computing f\n"); }
         else if (a>b+9+x) // should somewhat be here
            { g = func1(); printf("computing g\n"); }
         else
      { h = func1(); printf("computing h\n"); }
   }
   return 0;
}
}
```

Using gcov

```
$ info gcov   # gcov - coverage testing tool
$ cat ./run_gcov.sh

#!/bin/sh
rm *.gcda
rm *.c.gcov
gcc -fprofile-arcs -ftest-coverage -o test_profile1 ./test_profile1.c -lm
./test_profile1
gcov -b ./test_profile1.c
```

the output

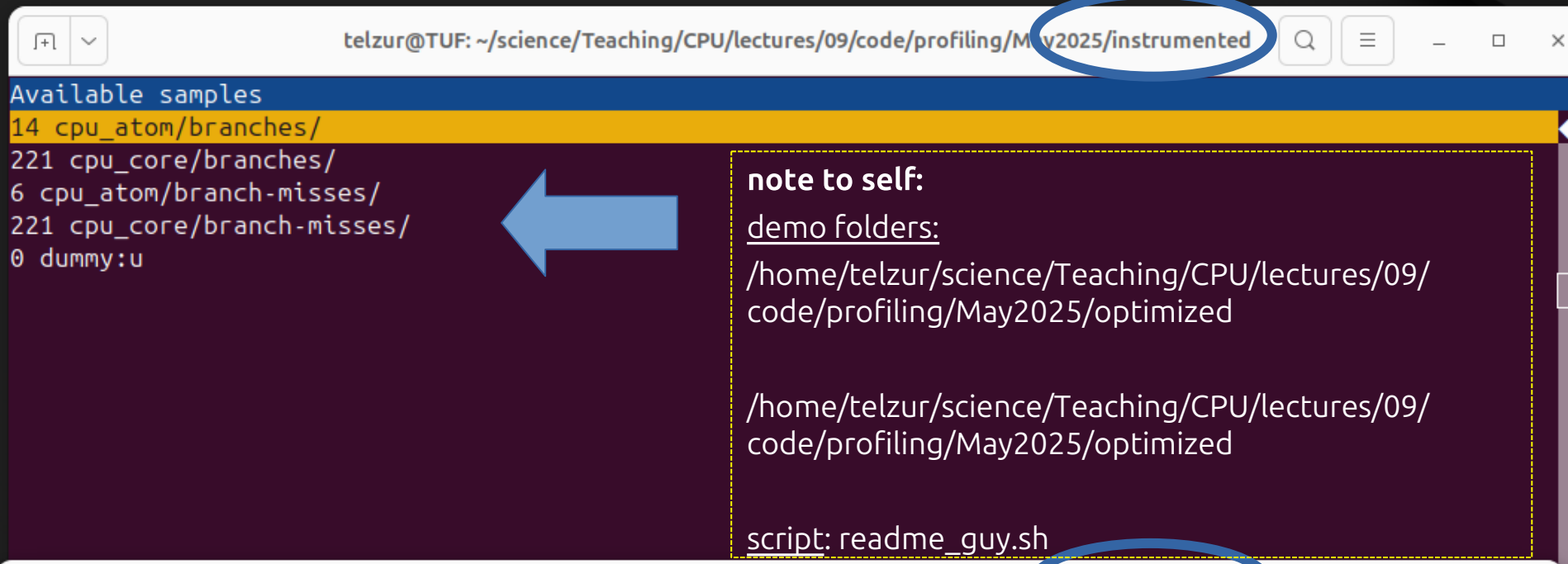
```
computing f
computing g
computing g
computing f
computing f
computing g
computing f
computing g
computing g
computing g
computing g
computing g
computing f
computing g
computing f
computing g
computing g
computing g
File 'test_profile1.c'
Lines executed:85.71% of 28
Branches executed:100.00% of 16
Taken at least once:75.00% of 16
Calls executed:46.67% of 15
Creating 'test_profile1.c.gcov'

Lines executed:85.71% of 28
```

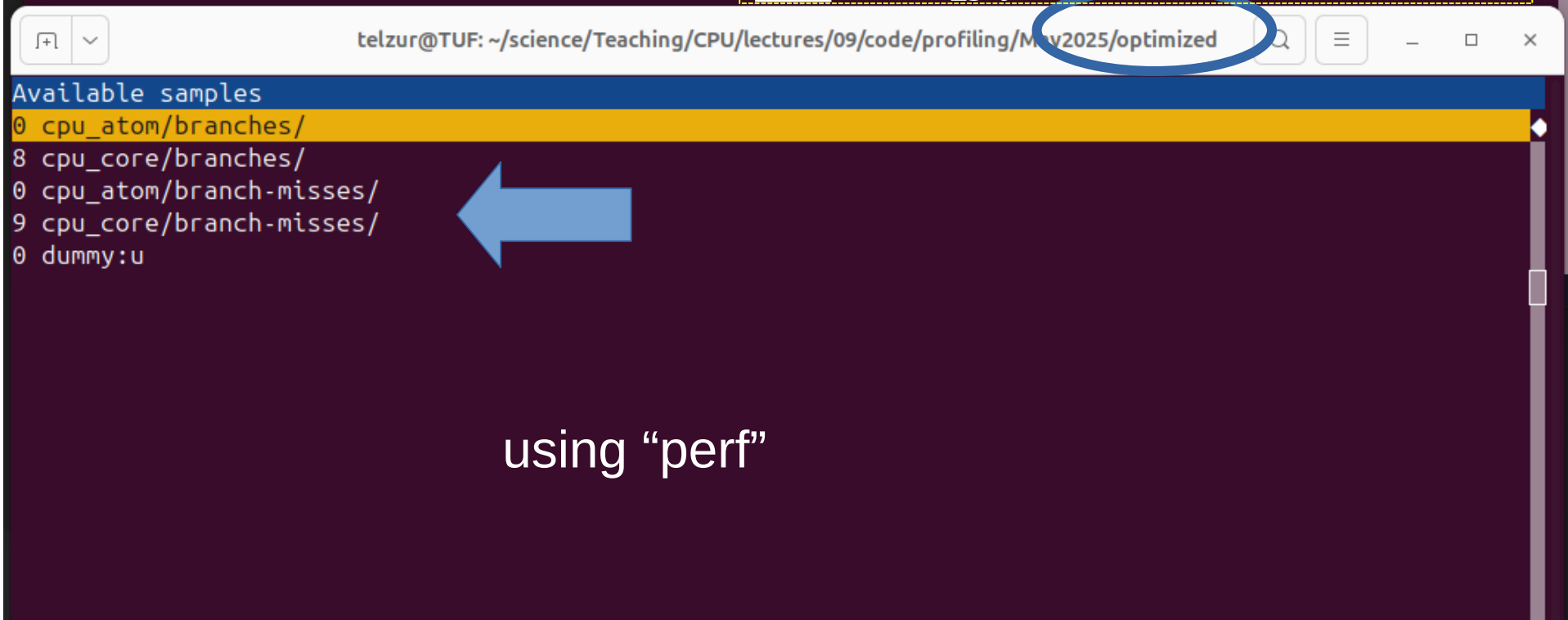
output of:

test_profile1.c.gcov

100 iterations in total (43 + 57)

```
        1:   26:    a = func1();
call    0 returned 100%
        1:   27:    b = func2();
call    0 returned 100%
      101:   28:    for (int i=0;i<N;i++) {
branch  0 taken 99%
branch  1 taken 1% (fallthrough)
      100:   29:            x = (float)rand()/(float)RAND_MAX;
call    0 returned 100%
      100:   30:        if (a==b)
branch  0 taken 0% (fallthrough)
branch  1 taken 100%
    #####:   31:            { c = func1(); printf("computing c\n"); }
call    0 never executed
call    1 never executed
      100:   32:        else if (a>b+20+x)
branch  0 taken 0% (fallthrough)
branch  1 taken 100%
    #####:   33:            { d = func1(); printf("computing d\n"); }
call    0 never executed
call    1 never executed
      100:   34:        else if (a>b+17+x)
branch  0 taken 0% (fallthrough)
branch  1 taken 100%
    #####:   35:                { e = func1(); printf("computing e\n"); }
call    0 never executed
call    1 never executed
      100:   36:            else if (a>b+9.5+x) // should somewhat be here
branch  0 taken 43% (fallthrough)
branch  1 taken 57%
       43:   37:                { f = func1(); printf("computing f\n"); }
call    0 returned 100%
call    1 returned 100%
       57:   38:            else if (a>b+9+x) // should somewhat be here
branch  0 taken 100% (fallthrough)
branch  1 taken 0%
       57:   39:                { g = func1(); printf("computing g\n"); }
call    0 returned 100%
call    1 returned 100%
        -:   40:            else
    #####:   41:            { h = func1(); printf("computing h\n"); }
```


telzur@TUF: ~/science/Teaching/CPU/lectures/09/code/profiling/May2025/instrumented
Available samples
14 cpu_atom/branches/
221 cpu_core/branches/
6 cpu_atom/branch-misses/
221 cpu_core/branch-misses/
0 dummy:u
note to self:
demo folders:
/home/telzur/science/Teaching/CPU/lectures/09/code/profiling/May2025/optimized
/home/telzur/science/Teaching/CPU/lectures/09/code/profiling/May2025/optimized
script: readme_guy.sh


telzur@TUF: ~/science/Teaching/CPU/lectures/09/code/profiling/May2025/optimized
Available samples
0 cpu_atom/branches/
8 cpu_core/branches/
0 cpu_atom/branch-misses/
9 cpu_core/branch-misses/
0 dummy:u

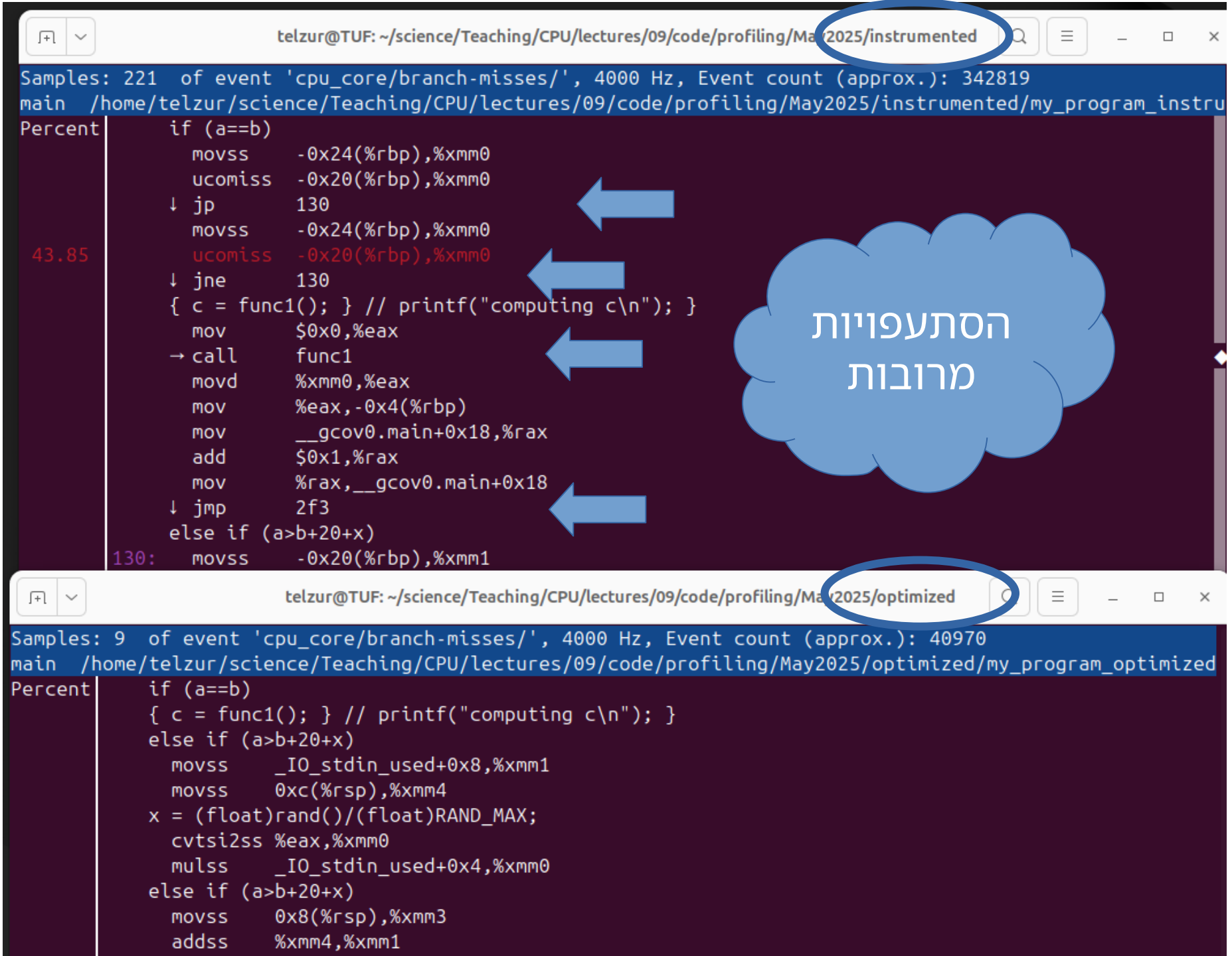
using “perf”

telzur@TUF: ~/science/Teaching/CPU/lectures/09/code/profiling/May2025/instrumented

```
Samples: 221  of event 'cpu_core/branch-misses/', Event count (approx.): 342819
Overhead  Command          Shared Object             Symbol
  58.57%  my_program_inst  libm.so.6                [.] __sin_fma
  27.90%  my_program_inst  libc.so.6                [.] __random
   8.84%  my_program_inst  my_program_instrumented  [.] func1
   3.27%  my_program_inst  my_program_instrumented  [.] main
   0.49%  my_program_inst  [kernel.kallsyms]        [k] __rb_erase_color
   0.47%  my_program_inst  [kernel.kallsyms]        [k] __pkru_allows_pkey
   0.45%  my_program_inst  [kernel.kallsyms]        [k] irqentry_exit_to_user_mode
   0.00%  my_program_inst  [kernel.kallsyms]        [k] __intel_pmu_enable_all.isra.0
   0.00%  my_program_inst  [kernel.kallsyms]        [k] perf_ctx_enable
```

telzur@TUF: ~/science/Teaching/CPU/lectures/09/code/profiling/May2025/optimized

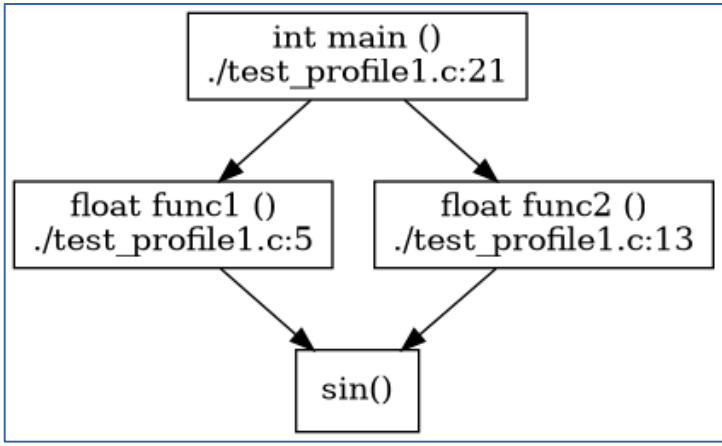
```
Samples: 9  of event 'cpu_core/branch-misses/', Event count (approx.): 40970
Overhead  Command          Shared Object             Symbol
  58.90%  my_program_opti  libc.so.6                [.] __random
  24.09%  my_program_opti  libc.so.6                [.] __random_r
  17.01%  my_program_opti  my_program_optimized     [.] main
   0.00%  perf-exec        [kernel.kallsyms]        [k] perf_ctx_enable
```


telzur@TUF: ~/science/Teaching/CPU/lectures/09/code/profiling/May2025/instrumented
Samples: 221 of event 'cpu_core/branch-misses/', 4000 Hz, Event count (approx.): 342819
main /home/telzur/science/Teaching/CPU/lectures/09/code/profiling/May2025/instrumented/my_program_instru
Percent
if (a==b)
movss -0x24(%rbp),%xmm0
ucomiss -0x20(%rbp),%xmm0
↓ jp 130
movss -0x24(%rbp),%xmm0
43.85 ucomiss -0x20(%rbp),%xmm0
↓ jne 130
{ c = func1(); } // printf("computing c\n"); }
mov $0x0,%eax
→ call func1
movd %xmm0,%eax
mov %eax,-0x4(%rbp)
mov __gcov0.main+0x18,%rax
add $0x1,%rax
mov %rax,__gcov0.main+0x18
↓ jmp 2f3
else if (a>b+20+x)
130: movss -0x20(%rbp),%xmm1
הסתעפויות מרובות
telzur@TUF: ~/science/Teaching/CPU/lectures/09/code/profiling/May2025/optimized
Samples: 9 of event 'cpu_core/branch-misses/', 4000 Hz, Event count (approx.): 40970
main /home/telzur/science/Teaching/CPU/lectures/09/code/profiling/May2025/optimized/my_program_optimized
Percent
if (a==b)
{ c = func1(); } // printf("computing c\n"); }
else if (a>b+20+x)
movss _IO_stdin_used+0x8,%xmm1
movss 0xc(%rsp),%xmm4
x = (float)rand()/(float)RAND_MAX;
cvtsi2ss %eax,%xmm0
mulss _IO_stdin_used+0x4,%xmm0
else if (a>b+20+x)
movss 0x8(%rsp),%xmm3
addss %xmm4,%xmm1

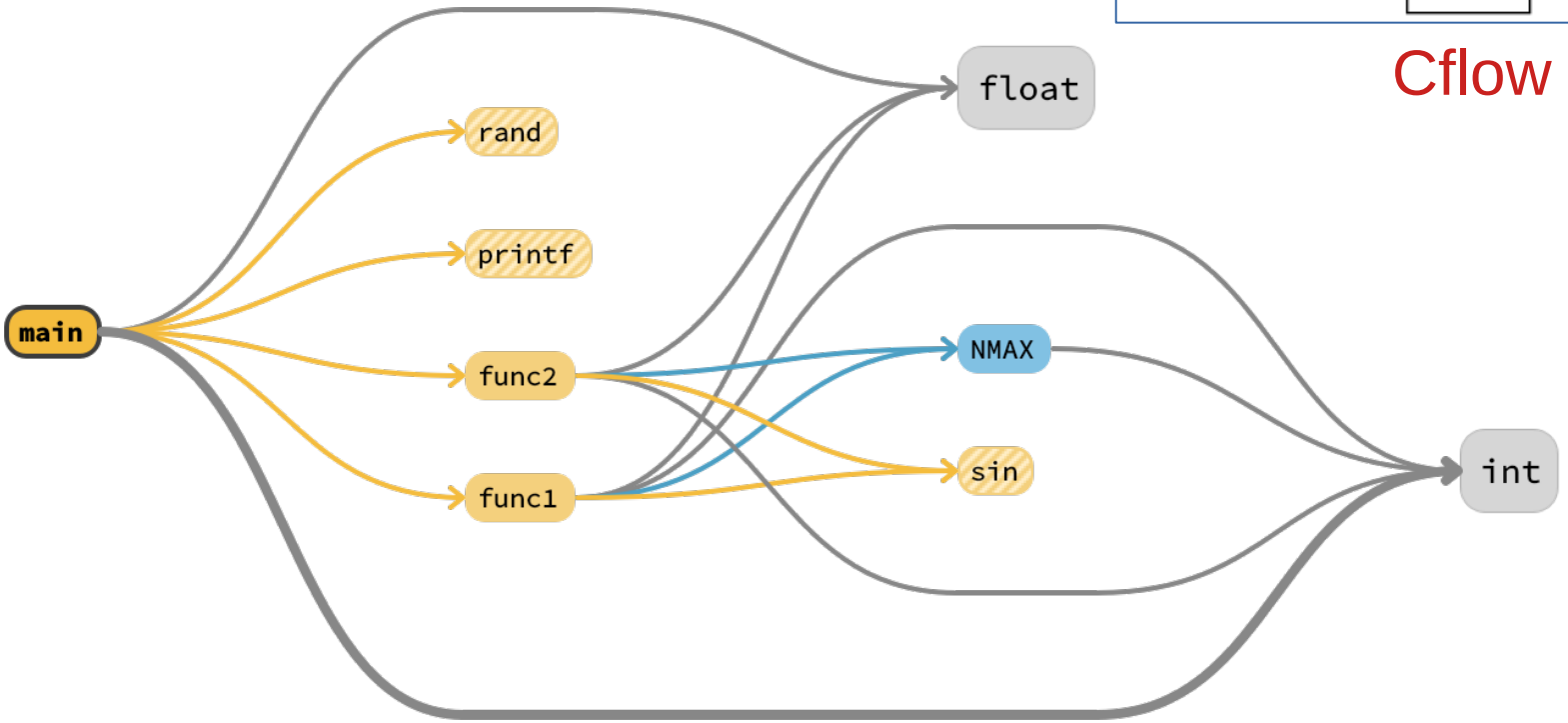
Call graph (Guy)

מיפוי של קשרים בתוך הקוד

אבל זהו **מיפוי סטטי**!



Cflow



Sourcetrail

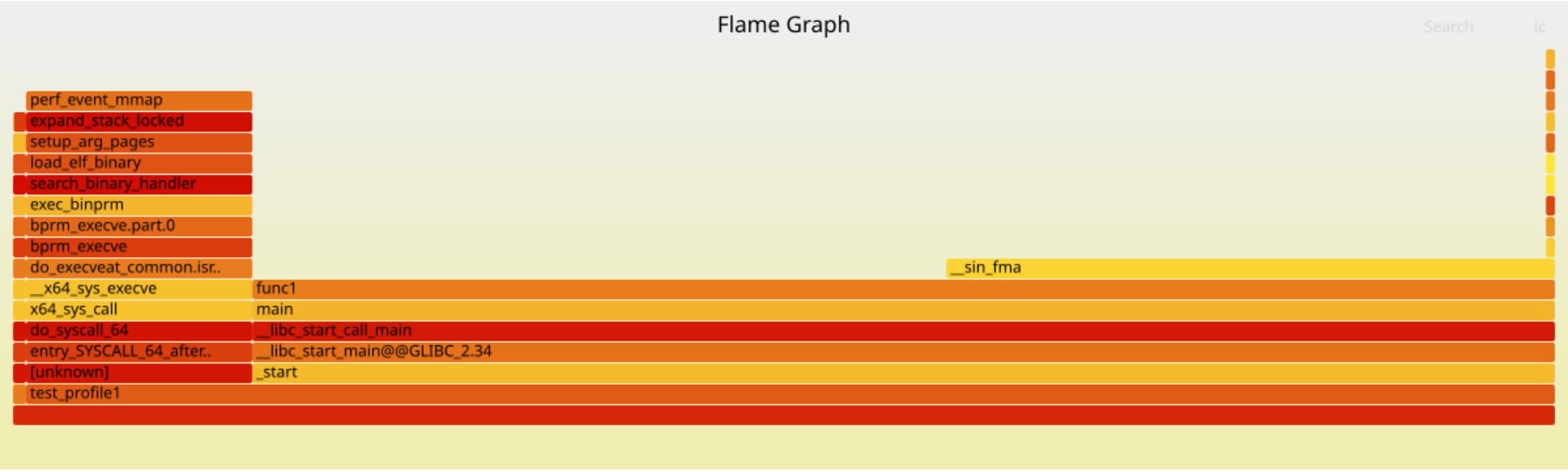
Exported from Sourcetrail ®

```
$ ~/appimages/Sourcetrail_2021_4_19_Linux_64bit.AppImage --no-sandbox
```

Perf + FlameGraph !

מיפוי של קשרים בתוך הקוד

עתה זהו **מיפוי דינאמי**!



FlameGraph is a visualization tool for profiling data (CPU, memory, etc.). It shows where time/samples are spent in your program as an inverted "flame" chart — each layer represents a function/call in the stack, wider = more time spent.

`file:///home/telzur/science/Teaching/CPU/lectures/09/code/profiling/May2025/FlameGraph/callgraph.svg`

Demo #2: Loop unrolling

```
telzur@TUF:~/science/Teaching/CPU/lectures/09/code/loop_unrolling$ cat ./code_with_a_loop.c
// compile: gcc -O0 -o code_with_a_loop ./code_with_a_loop.c -lm
// execute using: time ./code_with_a_loop
#include<math.h>
#include<stdio.h>
int main() {
  int SIZE = 40000;
  int LOOP = 1000;
  int i,j;

  float A[SIZE],B[SIZE],C[SIZE],D[SIZE];

  for (i=0;i<SIZE;i++) {
        A[i]=i+1;
        B[i]=2*i+1;
        C[i]=3*i+1;
        D[i]=4*i+1;
  }
  for (j=0;j<LOOP;j++)
  for (i=0;i<SIZE;i++) {
        A[i]=4+i;
        B[i]=2*i-cos(A[i]);
        C[i]=sin(B[i])+sqrt(B[i])/sqrt(A[i])+sin(A[i]);
            D[i]=cos(C[i])-cos(C[i])*sin(B[i])-sqrt(B[i]);
            // uncomment the next line for debugging. print the last 5 iterations:
            // if (i>=SIZE-5) printf("%d, %f, %f, %f, %f\n",i,A[i],B[i],C[i],D[i]);
  }
  return 0;
}
```

The original code

The same code with loop unrolling

```
  for (j=0;j<LOOP;j++)
  for (i=0;i<SIZE;i+=4) {
        A[i]=4+i;
        B[i]=2*i-cos(A[i]);
        C[i]=sin(B[i])+sqrt(B[i])/sqrt(A[i])+sin(A[i]);
        D[i]=cos(C[i])-cos(C[i])*sin(B[i])-sqrt(B[i]);


        A[i+1]=4+i;
        B[i+1]=2*i-cos(A[i+1]);
        C[i+1]=sin(B[i+1])+sqrt(B[i+1])/sqrt(A[i+1])+sin(A[i+1]);
        D[i+1]=cos(C[i+1])-cos(C[i+1])*sin(B[i+1])-sqrt(B[i+1]);

        A[i+2]=4+i;
        B[i+2]=2*i-cos(A[i+2]);
        C[i+2]=sin(B[i+2])+sqrt(B[i+2])/sqrt(A[i+2])+sin(A[i+2]);
        D[i+2]=cos(C[i+2])-cos(C[i+2])*sin(B[i+2])-sqrt(B[i+2]);

        A[i+3]=4+i;
        B[i+3]=2*i-cos(A[i+3]);
        C[i+3]=sin(B[i+3])+sqrt(B[i+3])/sqrt(A[i+3])+sin(A[i+3]);
        D[i+3]=cos(C[i+3])-cos(C[i+3])*sin(B[i+3])-sqrt(B[i+3]);
            // uncomment the next line for debugging. print the last 5 iterations:
            // if (i>=SIZE-5) printf("%d, %f, %f, %f, %f\n",i,A[i],B[i],C[i],D[i]);
  }
  return 0;
}
```

The results

```
telzur@TUF:~/science/Teaching/CPU/lectures/09/code/loop_unrolling$ cat ./run_me.sh
#!/bin/sh
gcc -g -O3 -o ./code_with_loop_unrolling ./code_with_loop_unrolling.c -lm
gcc -g -O3 -o ./code_with_a_loop ./code_with_a_loop.c -lm
echo "Now running a simple loop:"
time ./code_with_a_loop
echo " "
echo "Now running the same loop but unrolled:"
time ./code_with_loop_unrolling
echo " "
```

הסתייגות: הדגמת האפקט של פריסת לולאות ממוסכת על-ידי אופטימיזציות נוספות שהקומפיילר מבצע ובראשן **וקטוריזציה**

```
telzur@TUF:~/science/Teaching/CPU/lectures/09/code/loop_unrolling$ ./run_me.sh
Now running a simple loop:
1.65user 0.00system 0:01.66elapsed 100%CPU (0avgtext+0avgdata 2400maxresident)k
0inputs+0outputs (0major+237minor)pagefaults 0swaps

Now running the same loop but unrolled:
0.75user 0.00system 0:00.75elapsed 99%CPU (0avgtext+0avgdata 2400maxresident)k
0inputs+0outputs (0major+237minor)pagefaults 0swaps
```

Demo #3: Predicate execution

```
telzur@TUF:~/science/Teaching/CPU/lectures/09/code/predicate_exe$ more ./test1.c
void foo(int x, int y) {
        int  z;
        z = (y > x) ? y:x;
}
```

gcc -S ./test1.c

דוגמה זו ניתנה כבר קודם עם שימוש ב- Compiler Explorer

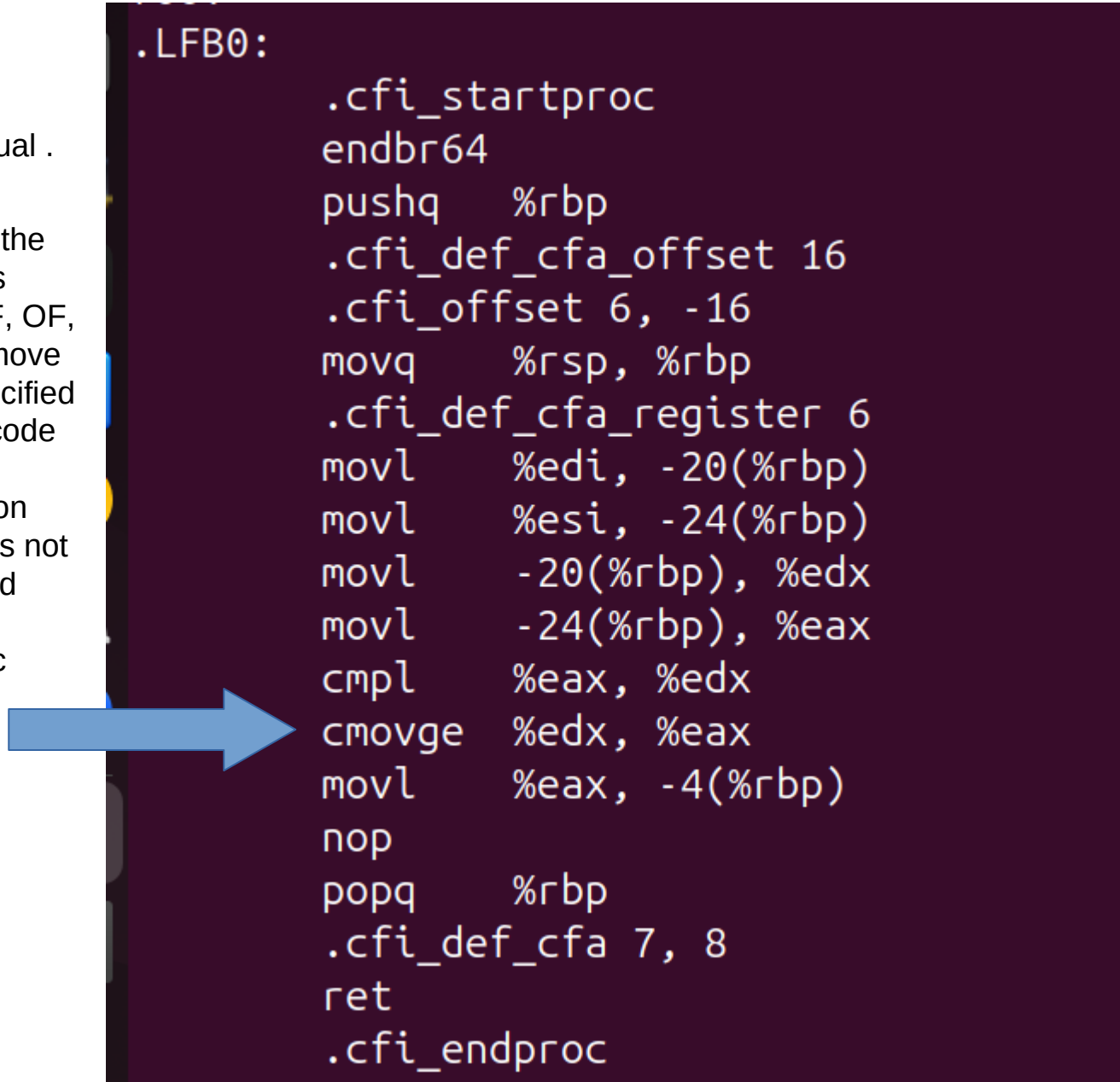
https://godbolt.org/

AMD64

CMOVGE

conditional move if greater or equal .

The CMOVcc instructions check the state of one or more of the status flags in the EFLAGS register (CF, OF, PF, SF, and ZF) and perform a move operation if the flags are in a specified state (or condition). A condition code (cc) is associated with each instruction to indicate the condition being tested for. If the condition is not satisfied, a move is not performed and execution continues with the instruction following the CMOVcc instruction.



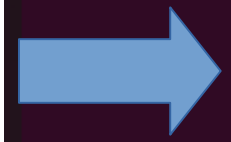
MIPS

MOVZ rd, rs, rt

This instruction means:

If the value in register rt is zero, then copy the contents of register rs into register rd. Otherwise, do nothing (i.e., rd retains its original value).

Therefore there is no branch as a result of this instruction!



```
foo:
        .frame  $fp,16,$31
        .mask   0x40000000,-4
        .fmask  0x00000000,0
        .set    noreorder
        .set    nomacro
        addiu   $sp,$sp,-16
        sw      $fp,12($sp)
        move    $fp,$sp
        sw      $4,16($fp)
        sw      $5,20($fp)
        lw      $2,20($fp)
        lw      $4,16($fp)
        lw      $3,16($fp)
        slt     $4,$4,$2
        movz    $2,$3,$4
        sw      $2,4($fp)
        nop
        move    $sp,$fp
        lw      $fp,12($sp)
        addiu   $sp,$sp,16
        jr      $31
        nop
```

RISC-V 64

```
telzur@TUF:~/science/Teaching/CPU/lectures/09/code/predicate_exe$
        .file   "test1.c"
        .option nopic
        .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .align  1
        .globl  foo
        .type   foo, @function
foo:
        addi    sp,sp,-48
        sd      ra,40(sp)
        sd      s0,32(sp)
        addi    s0,sp,48
        mv      a5,a0
        mv      a4,a1
        sw      a5,-36(s0)
        mv      a5,a4
        sw      a5,-40(s0)
        lw      a5,-40(s0)
        mv      a2,a5
        lw      a5,-36(s0)
        sext.w  a3,a5
        sext.w  a4,a2
        bge     a3,a4,.L2
        mv      a5,a2
.L2:
        sw      a5,-20(s0)
        nop
        ld      ra,40(sp)
        ld      s0,32(sp)
        addi    sp,sp,48
        jr      ra
        .size   foo, .-foo
        .ident  "GCC: () 15.2.0"
        .section        .note.GNU-stack,"",@progbits
```

Guy
Aside: compare the code size

```
telzur@TUF:~/science/Teaching/CPU/lectures/09/code/predicate_exe$ ll *.s.*
-rw-rw-r-- 1 telzur telzur 744 May 30 21:23 test1.s.mips
-rw-rw-r-- 1 telzur telzur 634 May 30 18:28 test1.s.riscv64
-rw-rw-r-- 1 telzur telzur 680 Jan 26  2023 test1.s.x86
telzur@TUF:~/science/Teaching/CPU/lectures/09/code/predicate_exe$
```

Branch hints: Likely/Unlikely

3 versions

version0: no “hints” – to be used as a reference version

```
telzur@TUF:~/science/Teaching/CPU/lectures/09/code/branch_prediction2$ cat ./version0.c
#include<stdio.h>
#define likely(x)       __builtin_expect(!!(x),1)
#define unlikely(x)     __builtin_expect(!!(x),0)
int main(int argc, char *argv[])
{
int n;
printf("Enter a number....\n");
scanf("%d",&n);
        if (n > 0) {
                printf("Positive\n");
        } else {
                printf("Zero or Negative\n");
        }
        return 0;
}
```

Reference:
https://medium.com/software-design/likely-unlikely-directives-802c09bd5232

a script to test the codes, for X86 and for MIPS

folder:

/home/telzur/science/Teaching/CPU/lectures/09/code/branch_prediction2

```
/bin/clear

cp version0.c version0_mips.c
cp version1.c version1_mips.c
cp version2.c version2_mips.c

#------------X86----------------------
echo " "
echo " X86: "
echo "======"
gcc -o version0 ./version0.c
gcc -o version1 ./version1.c
gcc -o version2 ./version2.c

gcc -S ./version0.c
gcc -S ./version1.c
gcc -S ./version2.c

diff  ./version0.s ./version1.s

#-----------MIPS----------------
echo " "
echo "MIPS: "
echo "======"
mips-linux-gnu-gcc -S ./version0_mips.c
mips-linux-gnu-gcc -S ./version1_mips.c
mips-linux-gnu-gcc -S ./version2_mips.c

diff ./version0_mips.s ./version1_mips.s
```

The results:

היפוך התנאים

היפוך התנאים

```
 X86:
======
1c1
<        .file   "version0.c"
---
>        .file   "version1.c"
41c41,44
<        jle     .L2
---
>        setg    %al
>        movzbl  %al, %eax
>        testq   %rax, %rax
>        je      .L2

MIPS:
======
1c1
<        .file   1 "version0_mips.c"
---
>        .file   1 "version1_mips.c"
69c69,71
<        blez    $2,$L2
---
>        slt     $2,$0,$2
>        andi    $2,$2,0x00ff
>        beq     $2,$0,$L2
telzur@TUF:~/science/Teaching/CPU/lectures/09/code/branch_prediction2$
```

version 1

```
telzur@TUF:~/science/Teaching/CPU/lectures/09/code/branch_prediction2$ cat ./version1.c
#include<stdio.h>
#define likely(x)       __builtin_expect(!!(x),1)
#define unlikely(x)     __builtin_expect(!!(x),0)
int main(int argc, char *argv[])
{
int n;
printf("Enter a number....\n");
scanf("%d",&n);
        if (likely(n > 0)) {
                printf("Positive\n");
        } else {
                printf("Zero or Negative\n");
        }
        return 0;
}
```

version 2

```
telzur@TUF:~/science/Teaching/CPU/lectures/09/code/branch_prediction2$ cat ./version2.c
#include<stdio.h>
#define   likely(x)     __builtin_expect(!!(x),1)
#define unlikely(x)     __builtin_expect(!!(x),0)
int main(int argc, char *argv[])
{
int n;
printf("Enter a number....\n");
scanf("%d",&n);
        if (unlikely(n > 0)) {
                printf("Positive\n");
        } else {
                printf("Zero or Negative\n");
        }
        return 0;
}
```


telzur@TUF: ~/science/Teaching/CPU/lectures/09/code/branch_prediction2
telzur@TUF:~/science/Teaching/CPU/lectures/09/code/branch_prediction2$ diff version0_mips.c version1_mips.c
9c9
<   if (n > 0) {

>   if (likely(n > 0)) {
telzur@TUF:~/science/Teaching/CPU/lectures/09/code/branch_prediction2$ diff version0_mips.s version1_mips.s
1c1
<   .file   1 "version0_mips.c"

>   .file   1 "version1_mips.c"
69c69,71
<   blez $2,$L2

>   slt $2,$0,$2
>   andi $2,$2,0x00ff
>   beq $2,$0,$L2
telzur@TUF:~/science/Teaching/CPU/lectures/09/code/branch_prediction2$

Dynamic Branch Prediction

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

    הסתגלות 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%? תשובה: 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)

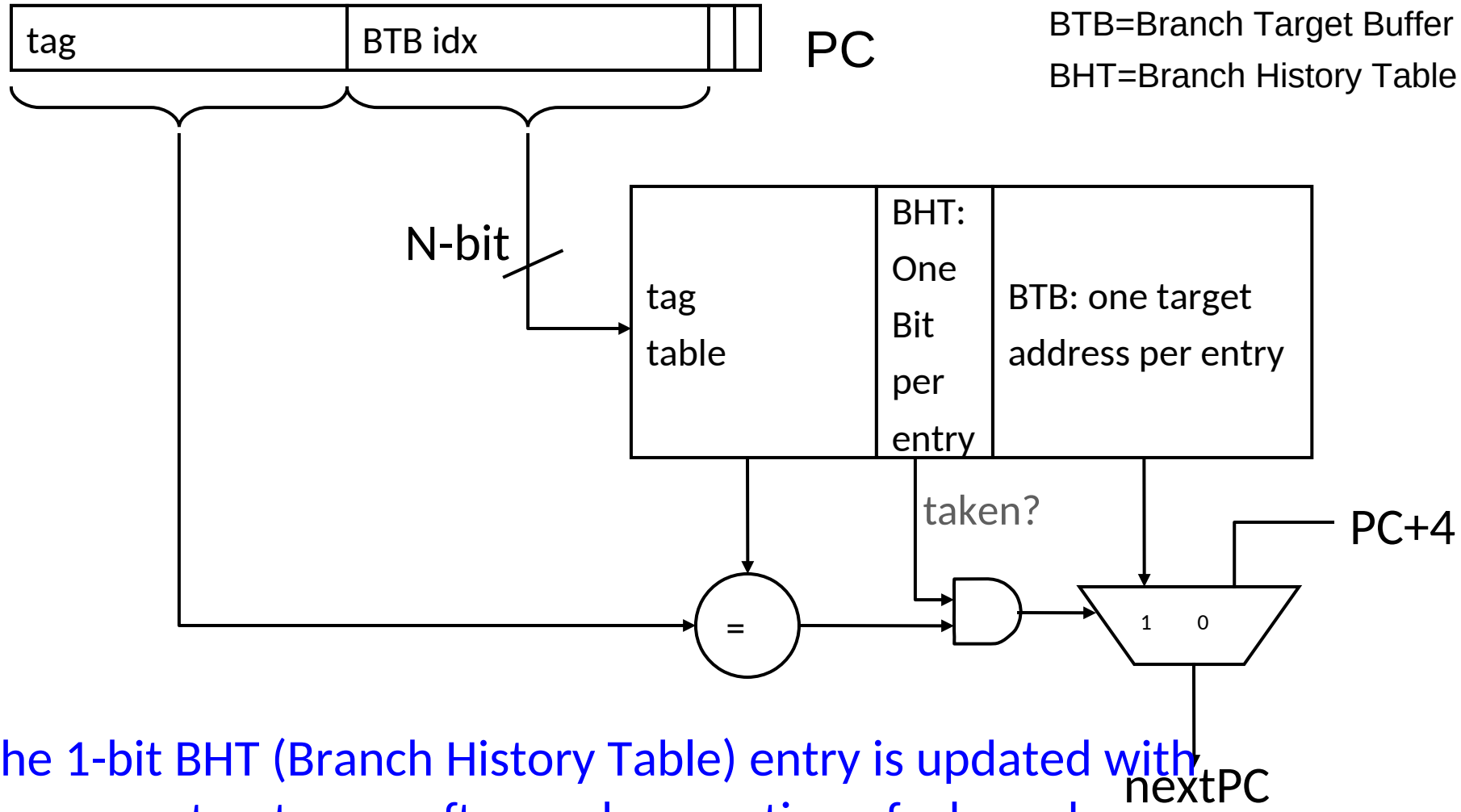
-- Loop branches for loops with small N (number of iterations)

TNTNTNTNTNTNTNTNTNTN → 0% accuracy

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

פה מניחים 20% סיכוי להסתעפות ו-15% סיכוי להחטאה

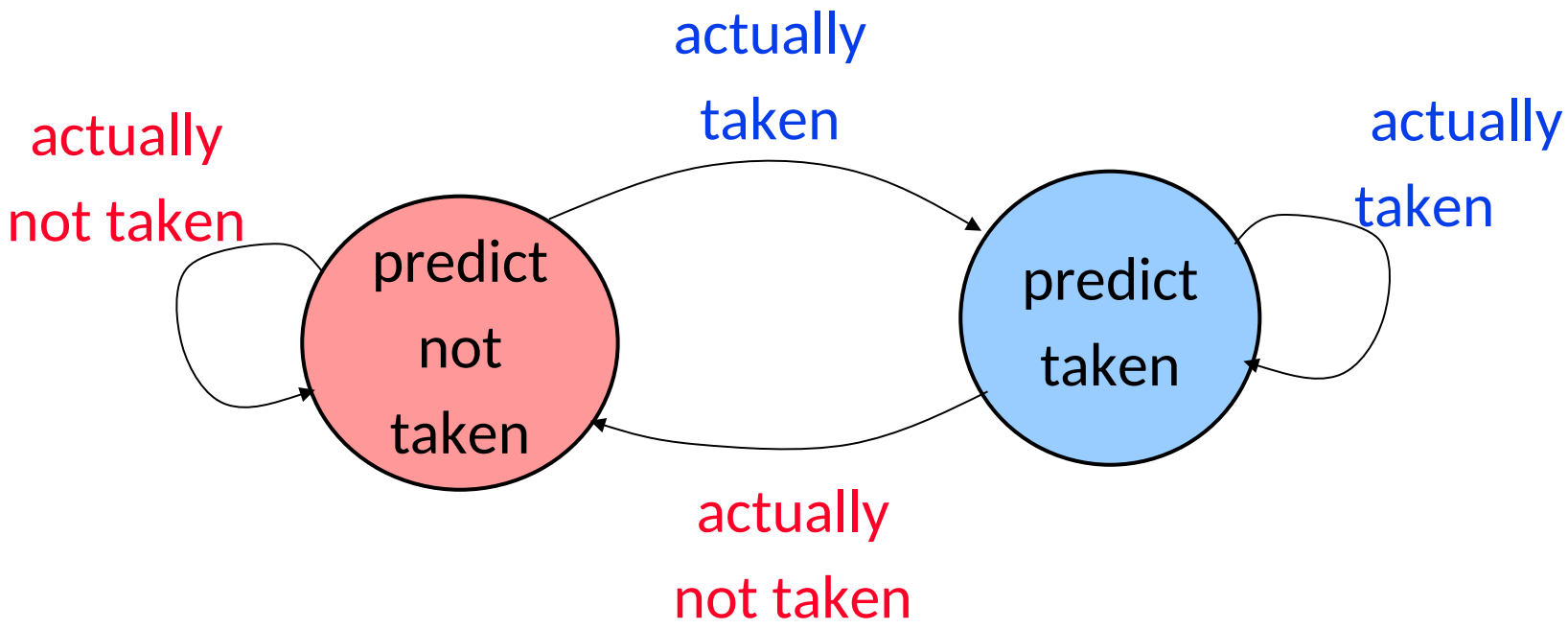
Implementing the Last-Time Predictor



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.

A STUDY OF BRANCH PREDICTION STRATEGIES

JAMES E. SMITH

Control Data Corporation
Arden Hills, Minnesota

Strategy 1: Predict that all branches will be taken.

Strategy 2: Always predict that a branch will be decided as on its last execution.

בסה"כ ניכר שיפור
באסטרטגיה השנייה לעומת
האסטרטגיה הראשונה

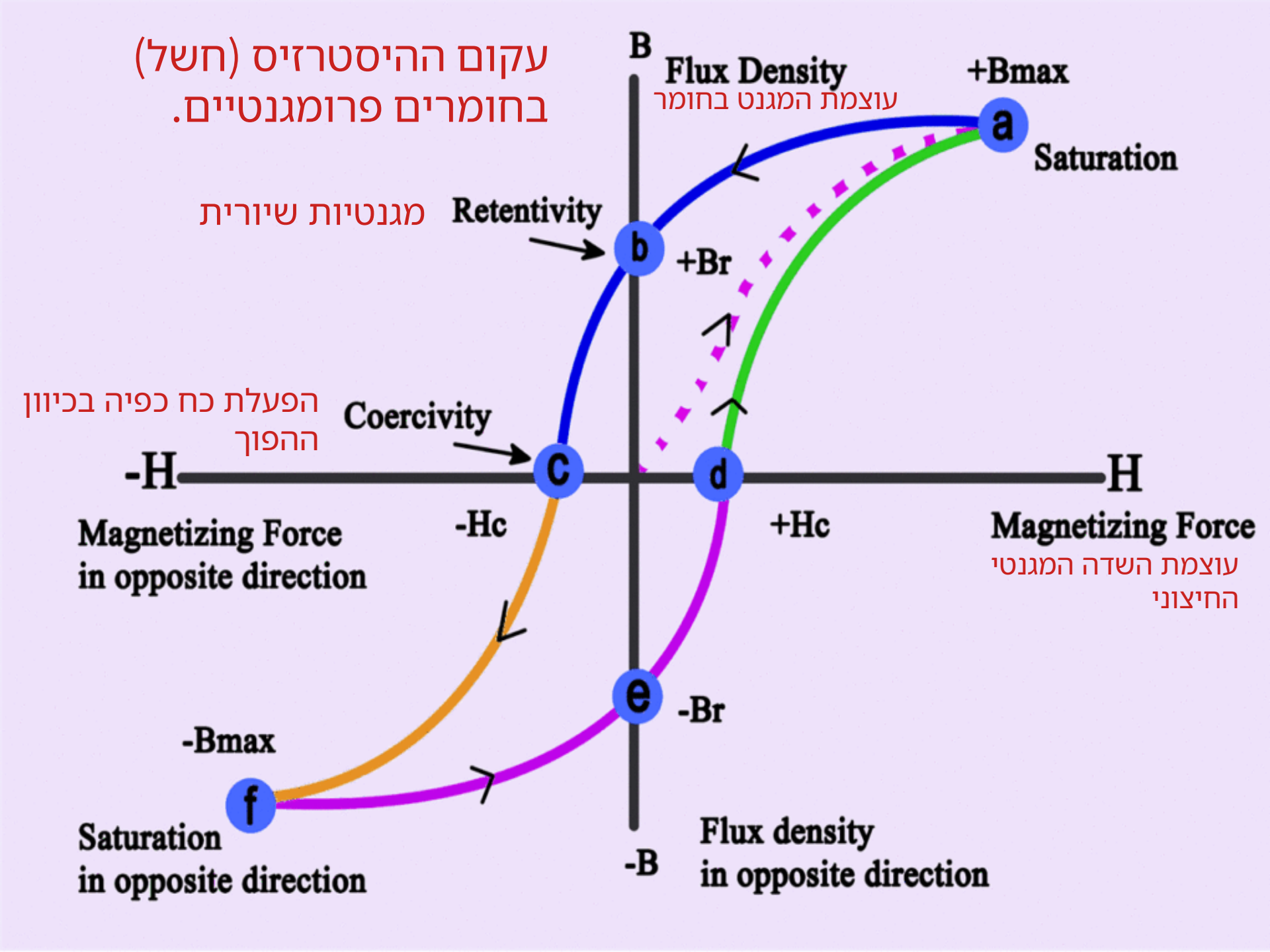
Program	Prediction Accuracy
ADVAN	99.4
GIBSON	65.4
SCI2	96.2
SINCOS	80.2
SORTST	57.4
TBLLNK	61.5

Figure 1. Accuracy of Prediction (%) for Strategy 1

Program	Prediction Accuracy
ADVAN	98.9
GIBSON	97.9
SCI2	96.0
SINCOS	76.2
SORTST	81.7
TBLLNK	91.7

Figure 2. Accuracy of Prediction (%) for Strategy 2

עקום ההיסטרזיס (חשל) בחומרים פרומגנטיים.



Two-Bit Counter Based Prediction

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

T=Taken
N=Not Taken

TNTNTNTNTNTNTNTNTNTNTN → 50% accuracy

עבור חיזוי באמצעות 2 ביטים

(assuming counter initialized to weakly taken)

צריך פעמיים N כדי לשנות את החיזוי, במקום ב- 100% מהמקרים במנגנון של ביט אחד

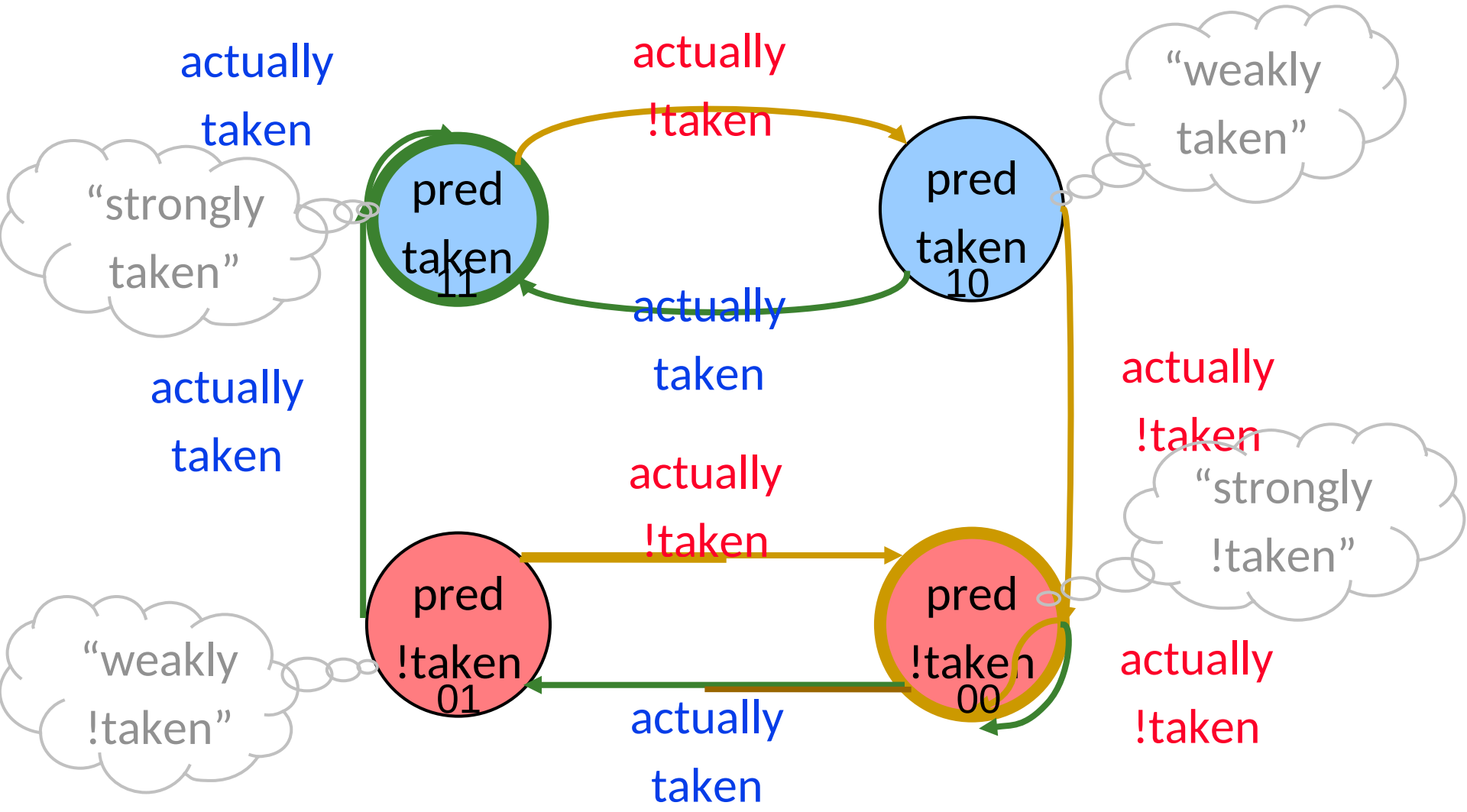
+ Better prediction accuracy

בדוגמה מלפני מספר שקפים TT...TTTNNN...NNN: נטעה ב-2 מקרים מתוך 20 --> 10%

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 7th Edition, appendix C

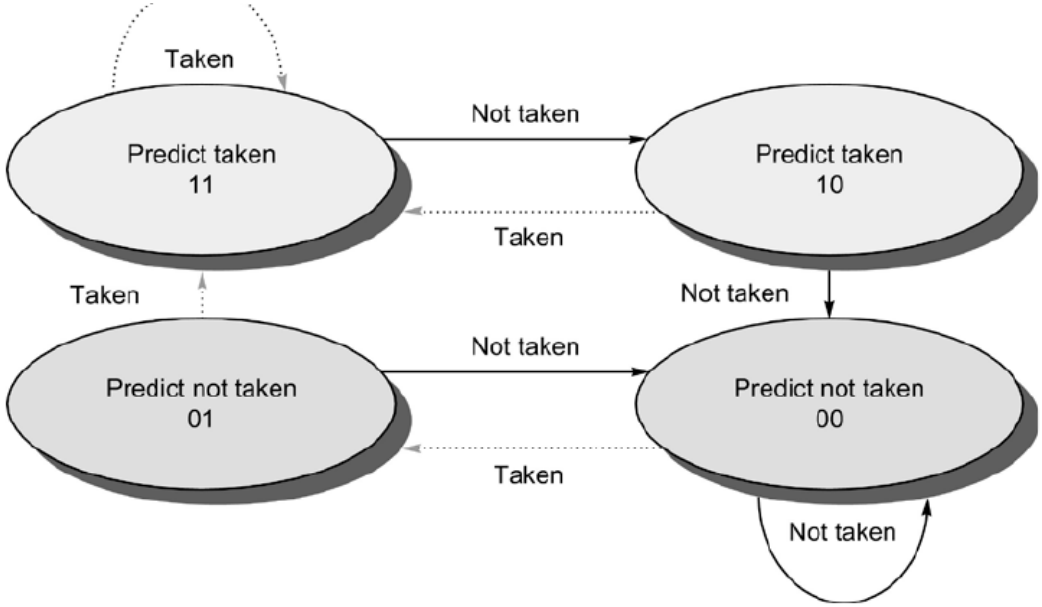
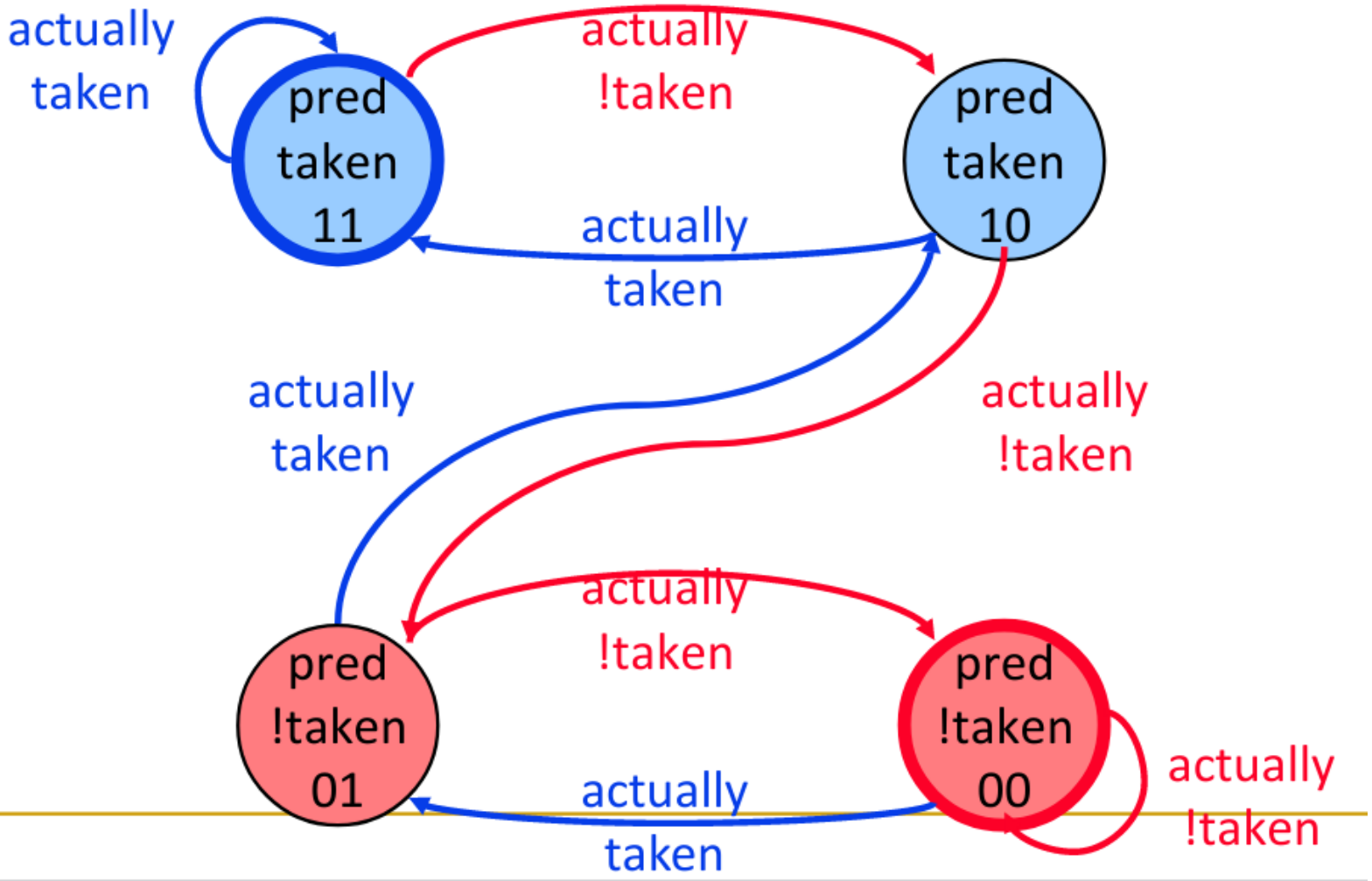


Figure C.15 The states in a 2-bit prediction scheme. By using 2 bits rather than 1, a branch that strongly favors taken or not taken—as many branches do—will be mispredicted less often than with a 1-bit predictor. The 2 bits are used to encode the four states in the system. The 2-bit scheme is actually 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

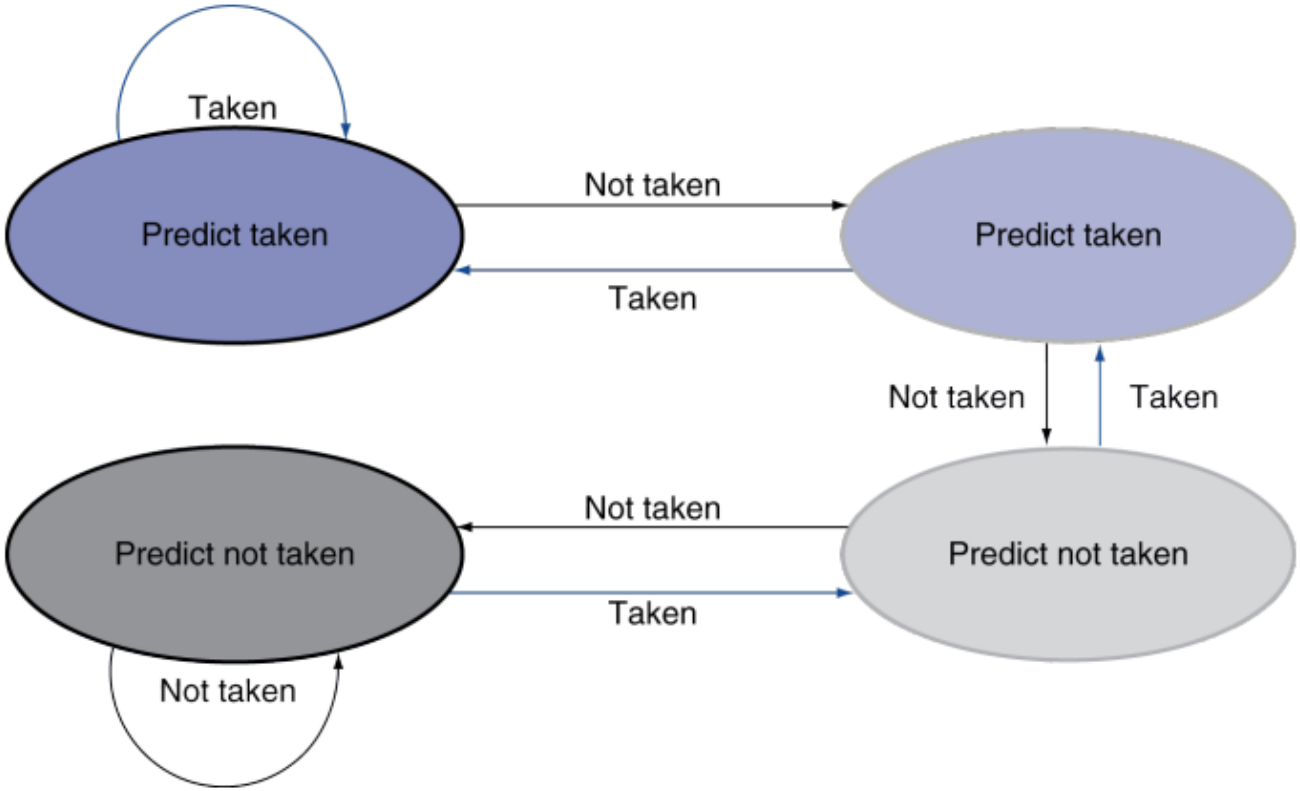
- Counter using *saturating arithmetic*
  - Arithmetic with maximum and minimum values



2-Bit Predictor

H&P.CO&D RISC-V Ed. , Chapter 4

- Only change prediction on two successive mispredictions



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

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, IPC=500/100=5
- 99% accuracy
  - 100 (correct path) + 20 (wrong path) = 120 cycles
  - 20% extra instructions fetched, IPC=500/120=4.17
- 98% accuracy
  - 100 (correct path) + 20 * 2 (wrong path) = 140 cycles
  - 40% extra instructions fetched, IPC=500/140=3.57
- 95% accuracy
  - 100 (correct path) + 20 * 5 (wrong path) = 200 cycles
  - 100% extra instructions fetched, IPC=500/200=2.5

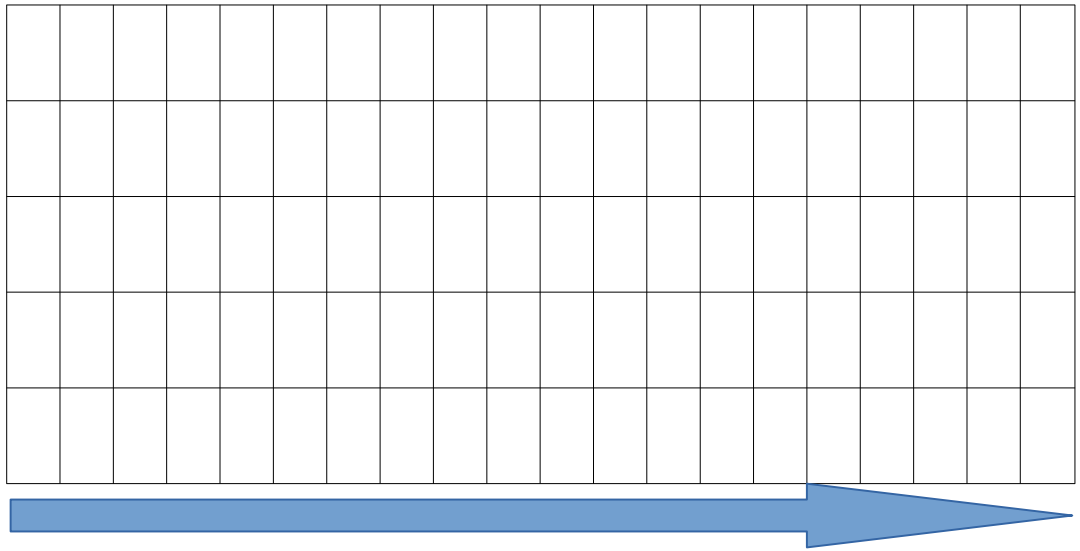
מצב אידאלי: בכל מחזור שעון מבוצעות 5 הוראות. לכן ידרשו 100=500/5 מחזורים

אם יש 100 מחזורי שעון (500 הוראות) ובאחד המחזורים היה חיזוי שגוי אז יאבדו 20 מחזורי שעון שנצטרך לחזור עליהם. סה"כ 100+20

מאבד 50% מה- throughput של החישוב

פייפליין באורך 20 דרגות ויכולת ביצוע של 5 הוראות במקביל

הסבר נוסף לשקף הקודם



עבור 100% הצלחה בניבוי:

בכל מחזור שעון מבוצעות 5 הוראות. לכן ב- 100 מחזורי שעון יבוצעו 500 הוראות.

IPC=500/100=5

עבור 95% הצלחה בניבוי:

אם יש 100 מחזורי שעון (500 הוראות) ובאחד המחזורים היה חיזוי שגוי אז יאבדו 20 מחזורי שעון שנצטרך לחזור. זה דיוק של 99%. אבל אם הדיוק הוא 95% זאת אומרת פי 5 יותר מחזורי שעון שילכו לאיבוד כלומר 100 מחזורי שעון. סהכ הפעם נשלים את החישוב ב-100 +100 מחזורי שעון, כלומר נזדקק ל-200 מחזורי שעון.

IPC=500/200=2.5

Can We Do Better?

- Last-time and 2BC predictors exploit “last-time” predictability

מתייחס רק להיסטוריה של ההסתעפות מהפעם הקודמת

- 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 (other than the outcome of the branch “last-time” it was executed)
  - → **Local** branch correlation

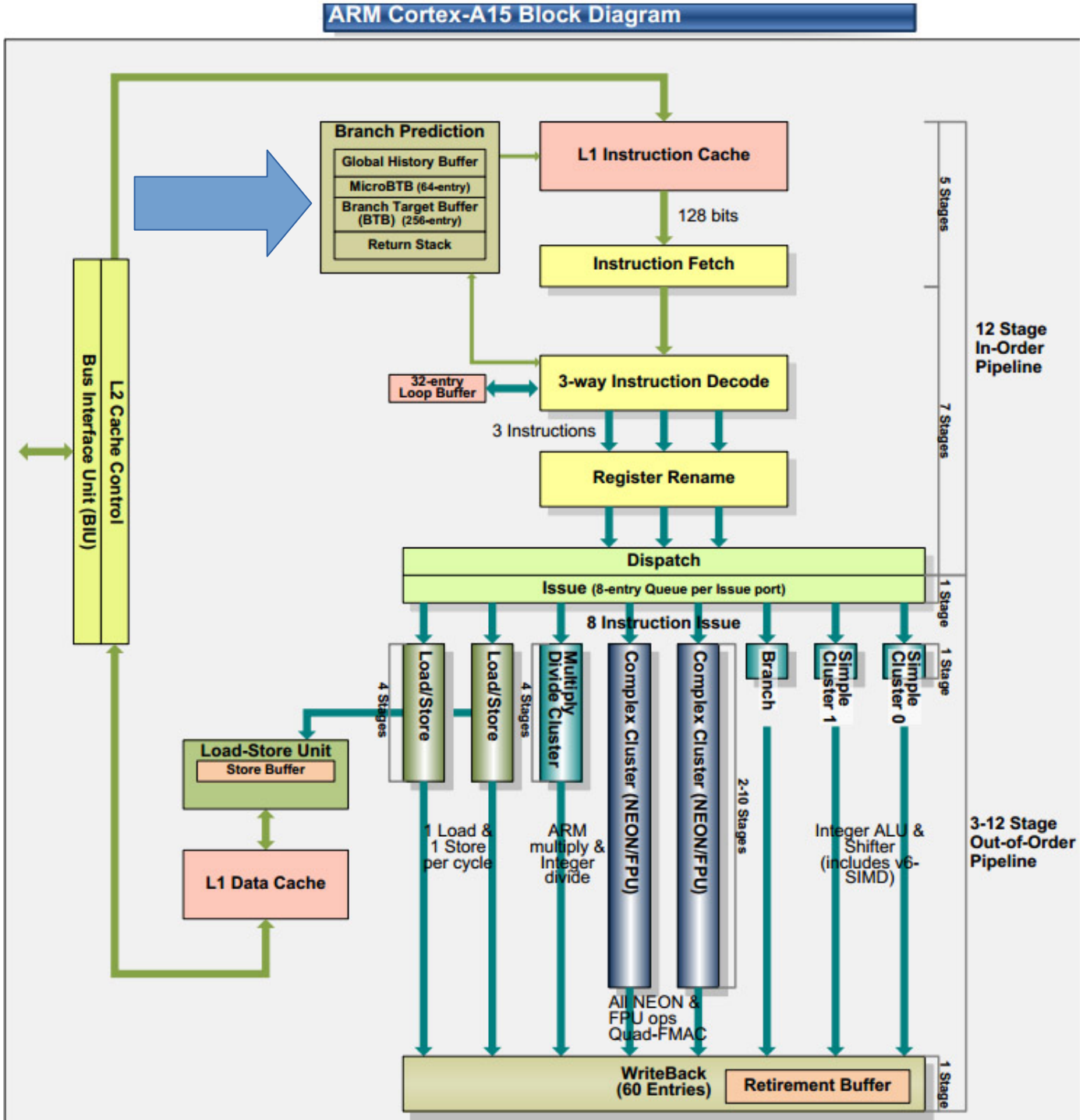
2BC=2 bit counter

What did we cover in this 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?


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

Spectre & Meltdown Side Channels Attacks

CVE-2017-5715 PUBLISHED

View JSON

Important CVE JSON 5 Information +

Assigner: Intel Corporation
Published: 2018-01-03 **Updated:** 2021-08-16

Systems with microprocessors utilizing speculative execution and indirect branch prediction may allow unauthorized disclosure of information to an attacker with local user access via a side-channel analysis.

Product Status

Learn About the Versions Section +

Vendor
Intel Corporation

Product
Microprocessors with Speculative Execution

Versions
Default Status: *unknown*

- affected at **All**

GCC Mitigation (check `man gcc`)

Branch Target Injection / CVE-2017-5715 / INTEL-SA-00088

Branch target injection takes advantage of the indirect branch predictors used by processors to direct what operations are speculatively executed after a near indirect branch instruction. By controlling how indirect branch predictors operate (“training”), an attacker can cause certain instructions to be speculatively executed and then use the effects the malicious code has on the processor’s caches to infer data values.

Ref: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/branch-target-injection.html

telzur@GL553VD ~/science/Downloads

File Edit View Search Terminal Help

-mindirect-branch=choice
Convert indirect call and jump with choice. The default is **keep**, which keeps indirect call and jump unmodified. **thunk** converts indirect call and jump to call and return thunk. **thunk-inline** converts indirect call and jump to inlined call and return thunk. **thunk-extern** converts indirect call and jump to external call and return thunk provided in a separate object file. You can control this behavior for a specific function by using the function attribute "indirect_branch".

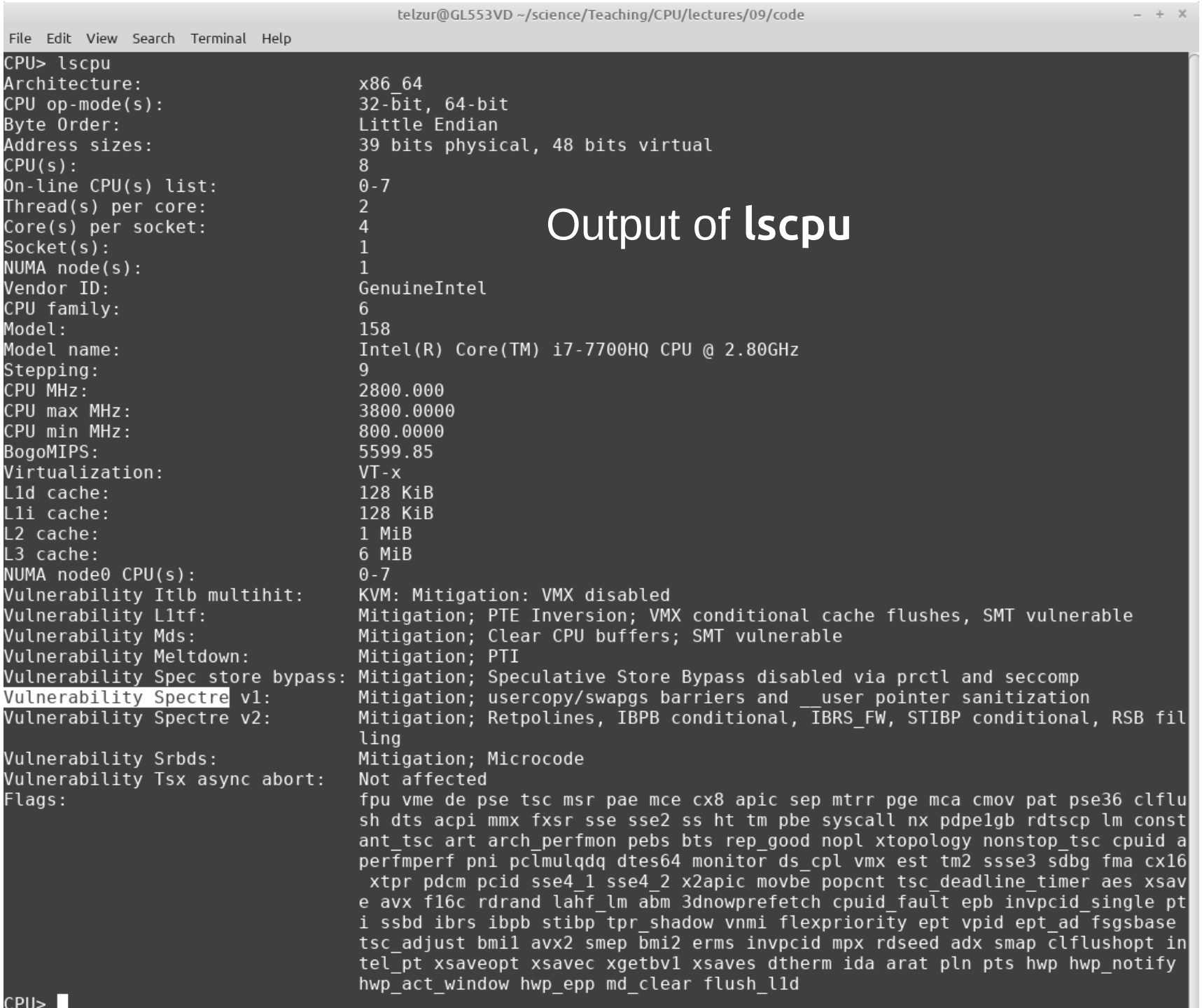
Note that **-mcmodel=large** is incompatible with **-mindirect-branch=thunk** and **-mindirect-branch=thunk-extern** since the thunk function may not be reachable in the large code model.

Note that **-mindirect-branch=thunk-extern** is compatible with **-fcf-protection=branch** since the external thunk can be made to enable control-flow check.

-mfunction-return=choice
Convert function return with choice. The default is **keep**, which keeps function return unmodified. **thunk** converts function return to call and return thunk. **thunk-inline** converts function return to inlined call and return thunk. **thunk-extern** converts function return to external call and return thunk provided in a separate object file. You can control this behavior for a specific function by using the function attribute "function_return".

Note that **-mindirect-return=thunk-extern** is compatible with **-fcf-protection=branch** since the external thunk can be made to enable control-flow check.

Note that **-mcmodel=large** is incompatible with **-mfunction-return=thunk** and **-mfunction-return=thunk-extern** since the thunk function may not be reachable in the large code model.


telzur@GL553VD ~/science/Teaching/CPU/lectures/09/code
File Edit View Search Terminal Help
CPU> lscpu
Architecture: x86_64
CPU op-mode(s): 32-bit, 64-bit
Byte Order: Little Endian
Address sizes: 39 bits physical, 48 bits virtual
CPU(s): 8
On-line CPU(s) list: 0-7
Thread(s) per core: 2
Core(s) per socket: 4
Socket(s): 1
NUMA node(s): 1
Vendor ID: GenuineIntel
CPU family: 6
Model: 158
Model name: Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
Stepping: 9
CPU MHz: 2800.000
CPU max MHz: 3800.0000
CPU min MHz: 800.0000
BogoMIPS: 5599.85
Virtualization: VT-x
L1d cache: 128 KiB
L1i cache: 128 KiB
L2 cache: 1 MiB
L3 cache: 6 MiB
NUMA node0 CPU(s): 0-7
Vulnerability Itlb multihit: KVM: Mitigation: VMX disabled
Vulnerability L1tf: Mitigation; PTE Inversion; VMX conditional cache flushes, SMT vulnerable
Vulnerability Mds: Mitigation; Clear CPU buffers; SMT vulnerable
Vulnerability Meltdown: Mitigation; PTI
Vulnerability Spec store bypass: Mitigation; Speculative Store Bypass disabled via prctl and seccomp
Vulnerability Spectre v1: Mitigation; usercopy/swapgs barriers and __user pointer sanitization
Vulnerability Spectre v2: Mitigation; Retpolines, IBPB conditional, IBRS_FW, STIBP conditional, RSB filling
Vulnerability Srbds: Mitigation; Microcode
Vulnerability Tsx async abort: Not affected
Flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb invpcid_single pti ssbd ibrs ibpb stibp tpr_shadow vnmi flexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid mpx rdseed adx smap clflushopt intel_pt xsaveopt xsavec xgetbv1 xsaves dtherm ida arat pln pts hwp hwp_notify hwp_act_window hwp_epp md_clear flush_l1d
CPU>
Output of lscpu

עד כאן מצגת זו