

# Digital Design & Computer Arch.

## Lecture 19a: VLIW

Prof. Onur Mutlu

ETH Zürich

Spring 2021

7 May 2021

# Approaches to (Instruction-Level) Concurrency

---

- Pipelining
- Fine-Grained Multithreading
- Out-of-order Execution
- Dataflow (at the ISA level)
- Superscalar Execution
- VLIW
- Systolic Arrays
- Decoupled Access Execute
- SIMD Processing (Vector and array processors, GPUs)

# VLIW Architectures

## (Very Long Instruction Word)

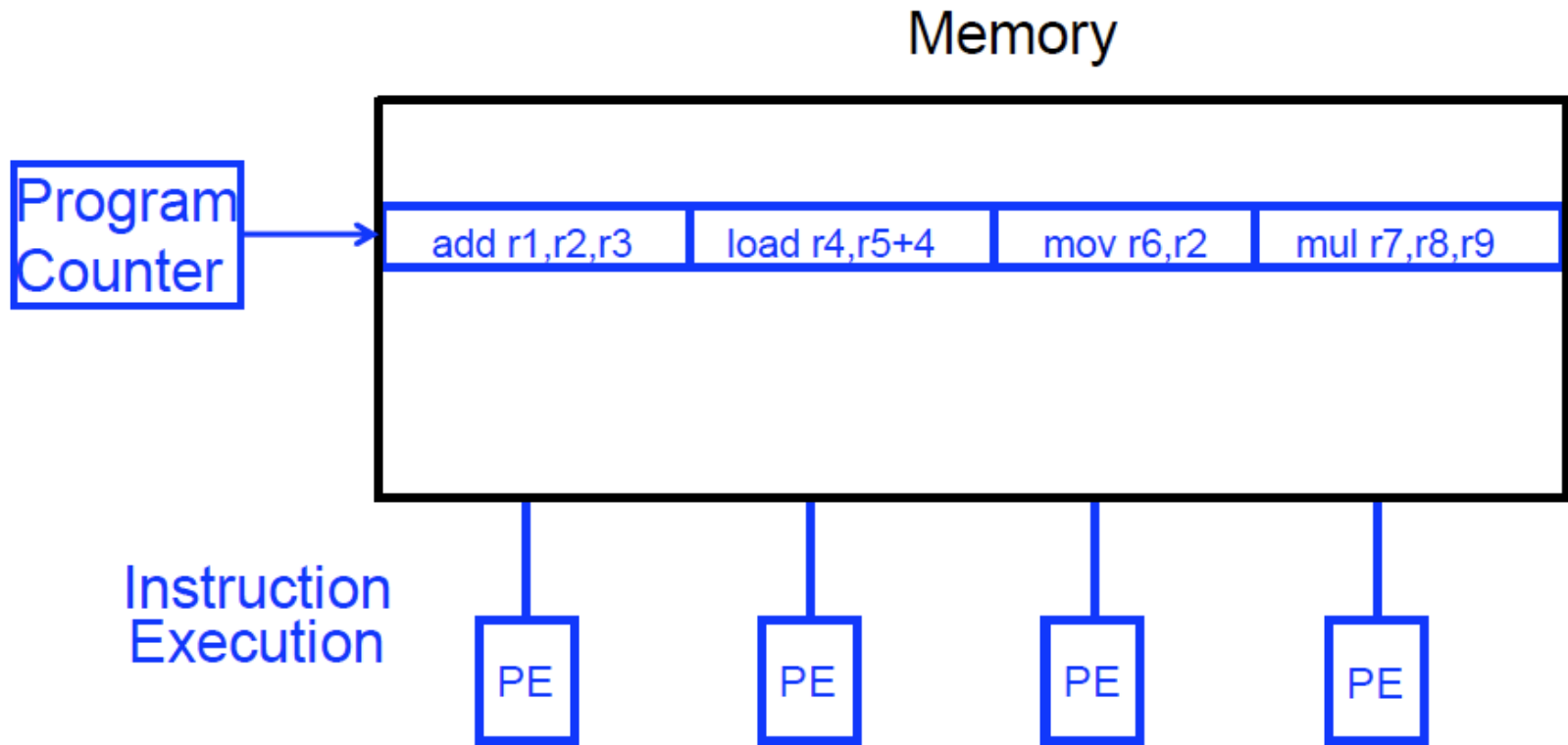
# VLIW Concept

---

- Superscalar
  - **Hardware** fetches multiple instructions and checks dependencies between them
- VLIW (Very Long Instruction Word)
  - **Software (compiler) packs independent instructions** in a larger “instruction bundle” to be fetched and executed concurrently
  - Hardware fetches and executes the instructions in the bundle concurrently
- **No need for hardware dependency checking** between concurrently-fetched instructions in the VLIW model

# VLIW Concept

---



- Fisher, “**Very Long Instruction Word architectures and the ELI-512**,” ISCA 1983.
  - ELI: Enormously longword instructions (512 bits)

# VLIW (Very Long Instruction Word)

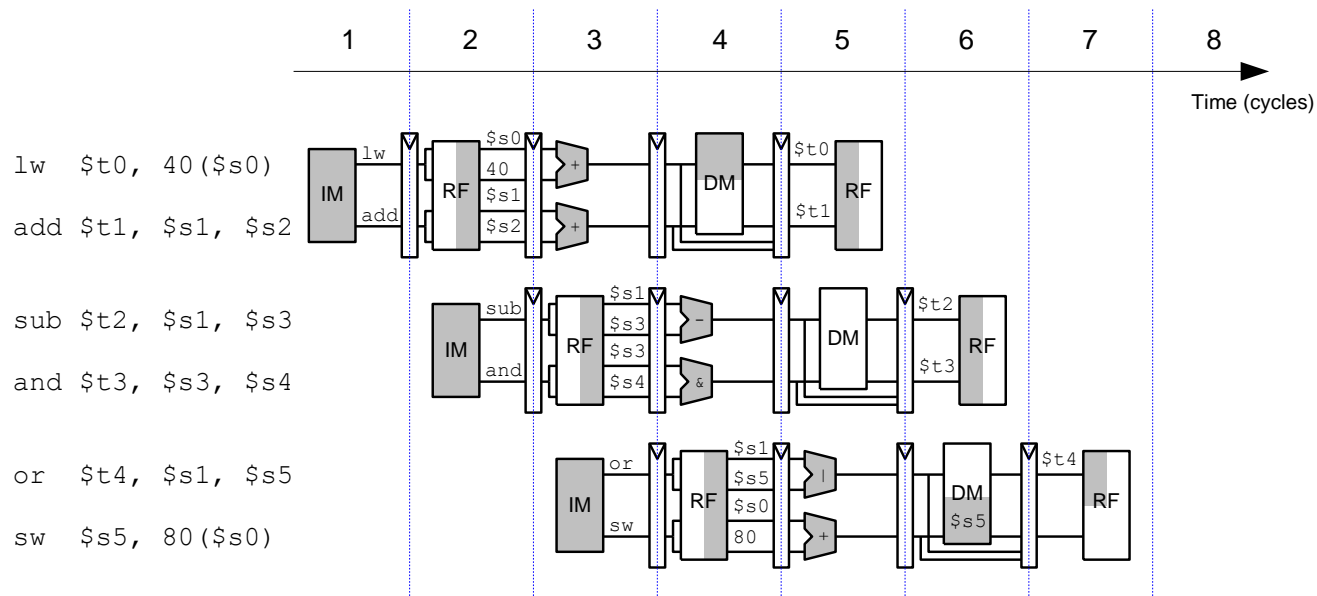
---

- A very long instruction word consists of **multiple independent instructions packed together by the compiler**
  - Packed instructions can be logically unrelated (contrast with SIMD/vector processors, which we will see soon)
- Idea: Compiler finds independent instructions and statically schedules (i.e. packs/bundles) them into a single VLIW instruction
- Traditional VLIW Characteristics
  - Multiple instruction fetch/execute, multiple functional units
  - All instructions in a bundle are executed in lock step
  - Instructions in a bundle statically aligned to be directly fed into the functional units

# VLIW Performance Example (2-wide bundles)

```
lw  $t0, 40($s0)    add $t1, $s1, $s2
sub $t2, $s1, $s3    and $t3, $s3, $s4
or  $t4, $s1, $s5    sw  $s5, 80($s0)
```

*Ideal IPC = 2*



*Actual IPC = 2* (6 instructions issued in 3 cycles)

# VLIW Lock-Step Execution

---

- Lock-step (all or none) execution
  - If any operation in a VLIW instruction stalls, all concurrent operations stall
- In a **truly VLIW machine**:
  - the compiler handles all dependency-related stalls
  - hardware does **not** perform dependency checking
  - What about variable latency operations? Memory stalls?



# VLIW Philosophy & Principles

---

Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction,  
*SIGPLAN Notices Vol. 19, No. 6, June 1984*

## Parallel Processing: A Smart Compiler and a Dumb Machine

Joseph A. Fisher, John R. Ellis,  
John C. Ruttenberg, and Alexandru Nicolau

Department of Computer Science, Yale University  
New Haven, CT 06520

### Abstract

Multiprocessors and vector machines, the only successful parallel architectures, have coarse-grained parallelism that is hard for compilers to take advantage of. We've developed a new fine-grained parallel architecture and a compiler that together offer order-of-magnitude speedups for ordinary scientific code.

future, and we're building a VLIW machine, the ELI (Enormously Long Instructions) to prove it.

In this paper we'll describe some of the compilation techniques used by the Bulldog compiler. The ELI project and the details of Bulldog are described elsewhere [4, 6, 7, 15, 17].

# VLIW Philosophy & Principles

---

- Philosophy similar to RISC (simple instructions and hardware)
  - Except multiple instructions in parallel
- RISC (John Cocke+, 1970s, IBM 801 minicomputer)
  - Compiler does the hard work to translate high-level language code to simple instructions (John Cocke: control signals)
    - And, to reorder simple instructions for high performance
  - Hardware does little translation/decoding → very simple
- VLIW (Josh Fisher, ISCA 1983)
  - Compiler does the hard work to find instruction level parallelism
  - Hardware stays as simple and streamlined as possible
    - Executes each instruction in a bundle in lock step
    - Simple → higher frequency, easier to design

# VLIW Philosophy and Properties

---

More formally, VLIW architectures have the following properties:

There is one central control unit issuing a single long instruction per cycle.

Each long instruction consists of many tightly coupled independent operations.

Each operation requires a small, statically predictable number of cycles to execute.

Operations can be pipelined. These properties distinguish VLIWs from multiprocessors (with large asynchronous tasks) and dataflow machines (without a single flow of control, and without the tight coupling). VLIWs have none of the required regularity of a vector processor, or true array processor.

# Commercial VLIW Machines

---

- Multiflow TRACE, Josh Fisher (7-wide, 28-wide)
- Cydrome Cydra 5, Bob Rau
- Transmeta Crusoe: x86 binary-translated into internal VLIW
- TI C6000, Trimedia, STMicro (DSP & embedded processors) and some ATI/AMD GPUs
  - Most successful commercially
- Intel IA-64
  - Not fully VLIW, but based on VLIW principles
  - EPIC (Explicitly Parallel Instruction Computing)
  - Instruction bundles can have dependent instructions
  - A few bits in the instruction format specify explicitly which instructions in the bundle are dependent on which other ones

# VLIW Tradeoffs

---

## ■ Advantages

- + No need for dynamic scheduling hardware → **simple hardware**
- + No need for dependency checking within a VLIW instruction → **simple hardware** for multiple instruction issue + no renaming
- + No need for instruction alignment/distribution after fetch to different functional units → **simple hardware**

## ■ Disadvantages

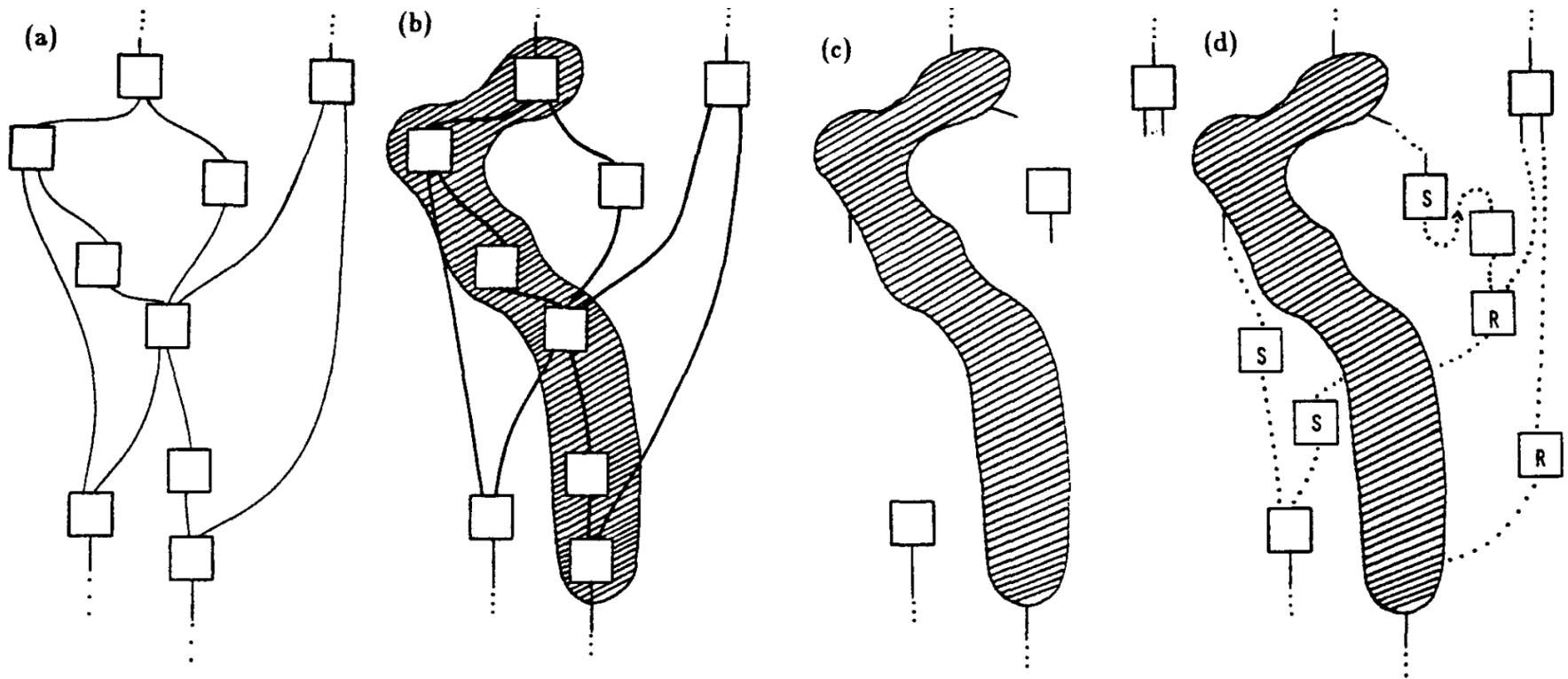
- **Compiler** needs to find N independent operations per cycle
  - If it cannot, inserts **NOPs** in a VLIW instruction
  - Parallelism loss AND code size increase
- **Recompilation required** when execution width (N), instruction latencies, functional units change (Unlike superscalar processing)
- **Lockstep execution** causes independent operations to stall
  - No instruction can progress until the longest-latency instruction completes

# VLIW Summary

---

- VLIW simplifies hardware, but requires complex compiler techniques
- Solely-compiler approach of VLIW has several downsides that reduce performance
  - Too many NOPs (not enough parallelism discovered)
  - Static schedule intimately tied to microarchitecture
    - Code optimized for one generation performs poorly for next
  - No tolerance for variable or long-latency operations (lock step)
- ++ Most compiler optimizations developed for VLIW employed in optimizing compilers (for superscalar compilation)
  - Enable code optimizations
- ++ VLIW very successful when parallelism is easier to find by the compiler (traditionally embedded markets, DSPs, GPUs)

# Example Work: Trace Scheduling



TRACE SCHEDULING LOOP-FREE CODE

(a) A flow graph, with each block representing a basic block of code. (b) A trace picked from the flow graph. (c) The trace has been scheduled but it hasn't been relinked to the rest of the code. (d) The sections of unscheduled code that allow re-linking.

# Recommended Paper

---

## VERY LONG INSTRUCTION WORD ARCHITECTURES AND THE ELI-512

JOSEPH A. FISHER  
YALE UNIVERSITY  
NEW HAVEN, CONNECTICUT 06520

### ABSTRACT

By compiling ordinary scientific applications programs with a radical technique called trace scheduling, we are generating code for a parallel machine that will run these programs faster than an equivalent sequential machine — we expect 10 to 30 times faster.

Trace scheduling generates code for machines called Very Long Instruction Word architectures. In Very Long Instruction Word machines, many statically scheduled, tightly coupled, fine-grained operations execute in parallel within a single instruction stream. VLIWs are more parallel extensions of several current architectures.

These current architectures have never cracked a fundamental barrier. The speedup they get from parallelism is never more than a factor of 2 to 3. Not that we couldn't build more parallel machines of this type; but until trace scheduling we didn't know how to generate code for them. Trace scheduling finds sufficient parallelism in ordinary code to justify thinking about a highly parallel VLIW.

At Yale we are actually building one. Our machine, the ELI-512, has a horizontal instruction word of over 500 bits and

are presented in this paper. How do we put enough tests in each cycle without making the machine too big? How do we put enough memory references in each cycle without making the machine too slow?

### WHAT IS A VLIW?

Everyone wants to use cheap hardware in parallel to speed up computation. One obvious approach would be to take your favorite Reduced Instruction Set Computer, let it be capable of executing 10 to 30 RISC-level operations per cycle controlled by a very long instruction word. (In fact, call it a VLIW.) A VLIW looks like very parallel horizontal microcode.

More formally, VLIW architectures have the following properties:

- There is one central control unit issuing a single long instruction per cycle.

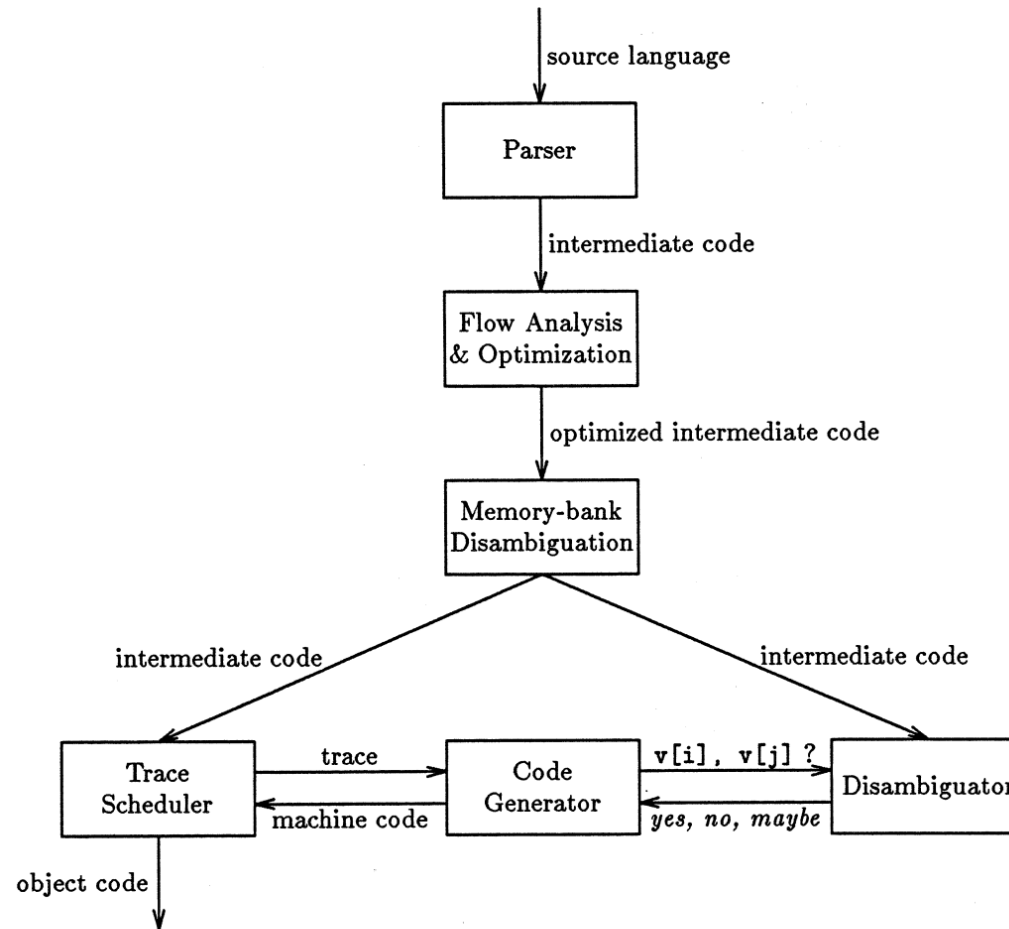
- Each long instruction consists of many tightly coupled independent operations.

- Each operation requires a small, statically predictable number of cycles to execute.

- Operations can be pipelined. These properties distinguish



# The Bulldog VLIW Compiler



**Figure 1.5.** The Bulldog compiler.

# Another Example Work: Superblock

---

## The Superblock: An Effective Technique for VLIW and Superscalar Compilation

Wen-mei W. Hwu      Scott A. Mahlke      William Y. Chen      Pohua P. Chang

Nancy J. Warter      Roger A. Bringmann      Roland G. Ouellette      Richard E. Hank

Tokuzo Kiyohara      Grant E. Haab      John G. Holm      Daniel M. Lavery \*

Hwu et al., [The superblock: An effective technique for VLIW and superscalar compilation.](#)  
The Journal of Supercomputing, 1993.

- Lecture Video on Static Instruction Scheduling
  - <https://www.youtube.com/watch?v=isBEVkIjgGA>

# Another Example Work: IMPACT

---

## IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors

*Pohua P. Chang*

*Scott A. Mahlke*

*William Y. Chen*

*Nancy J. Warter*

*Wen-mei W. Hwu*

Center for Reliable and High-Performance Computing  
University of Illinois  
Urbana, IL 61801

The performance of multiple-instruction-issue processors can be severely limited by the compiler's ability to generate efficient code for concurrent hardware. In the IMPACT project, we have developed IMPACT-I, a highly optimizing C compiler to exploit instruction level concurrency. The optimization capabilities of the IMPACT-I C compiler are summarized in this paper. Using the IMPACT-I C compiler, we ran experiments to analyze the performance of multiple-instruction-issue processors executing some important non-numerical programs. The multiple-instruction-issue processors achieve solid speedup over high-performance single-instruction-issue processors.

# Another Example Work: Hyperblock

---

## Effective Compiler Support for Predicated Execution Using the Hyperblock

Scott A. Mahlke   David C. Lin\*   William Y. Chen   Richard E. Hank   Roger A. Bringmann

Center for Reliable and High-Performance Computing  
University of Illinois  
Urbana-Champaign, IL 61801

- Lecture Video on Static Instruction Scheduling
  - <https://www.youtube.com/watch?v=isBEVkIjgGA>

# Lecture on Static Instruction Scheduling

Trace Scheduling Idea

(a) (b) (c) (d)

TRACE SCHEDULING LOOP-FREE CODE

43

Lecture 16. Static Instruction Scheduling - Carnegie Mellon - Comp. Arch. 2015 - Onur Mutlu

7,136 views • Feb 26, 2015

46 0 SHARE SAVE ...



**Carnegie Mellon Computer Architecture**  
23K subscribers

Lecture 16: Static Instruction Scheduling  
Lecturer: Prof. Onur Mutlu (<http://users.ece.cmu.edu/~omutlu/>)  
Date: Feb 23rd, 2015

Lecture 16 slides (pdf): <http://www.ece.cmu.edu/~ece447/s15/li...>

<https://www.youtube.com/onurmutlulectures>

# Lectures on Static Instruction Scheduling

---

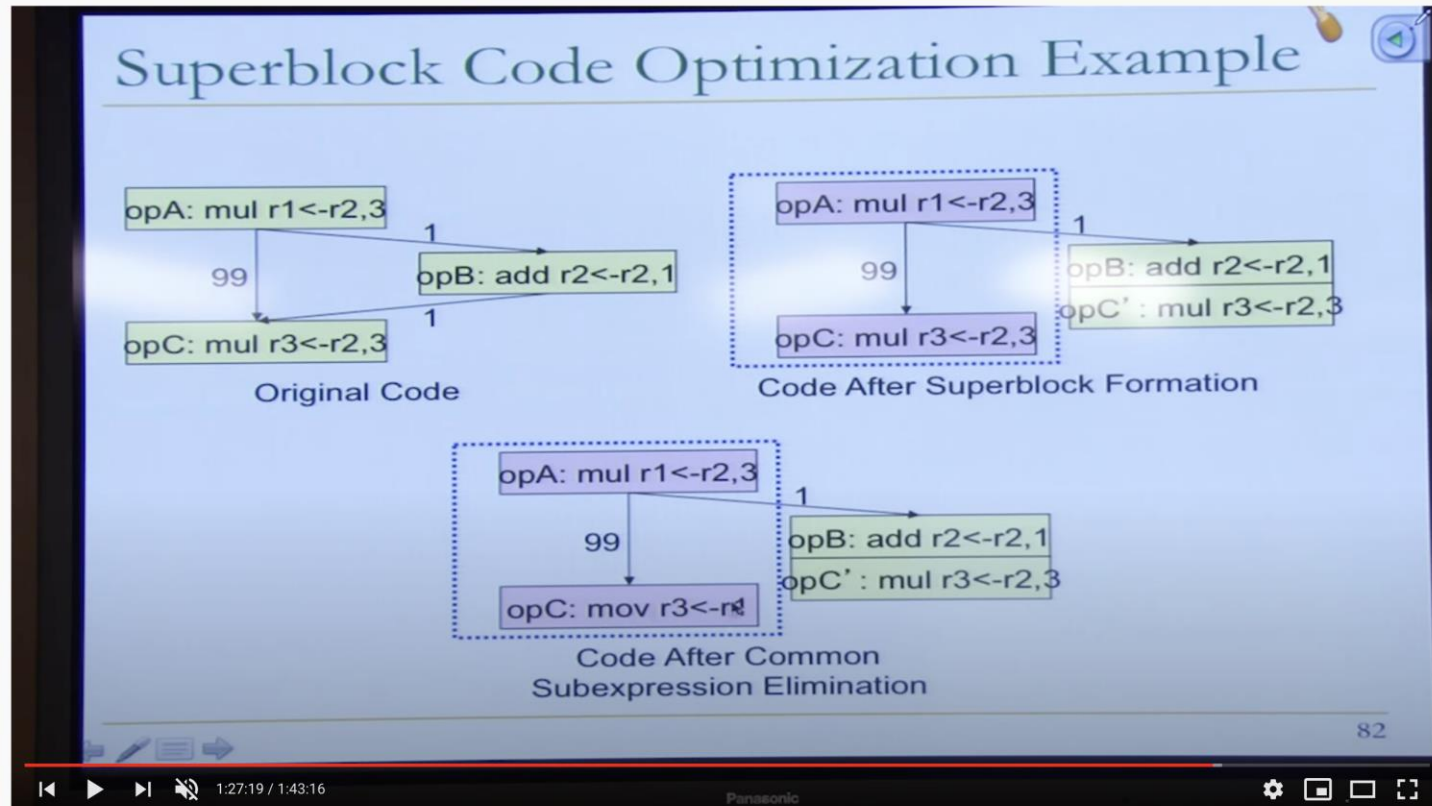
- Computer Architecture, Spring 2015, Lecture 16

- Static Instruction Scheduling (CMU, Spring 2015)
- <https://www.youtube.com/watch?v=isBEVkJjgGA&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=18>

- Computer Architecture, Spring 2013, Lecture 21

- Static Instruction Scheduling (CMU, Spring 2013)
- <https://www.youtube.com/watch?v=XdDUn2WtkRg&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=21>

# A More Compact Version...



18-740 Computer Architecture - Advanced Branch Prediction - Lecture 5

4,696 views • Sep 23, 2015

41 0 SHARE SAVE ...



**Carnegie Mellon Computer Architecture**  
23K subscribers

Lecture 5: Advanced Branch Prediction

Lecturer: Prof. Onur Mutlu (<http://users.ece.cmu.edu/~omutlu/>)

Date: September 16, 2014.

Lecture 5 slides (pdf): <http://www.ece.cmu.edu/~ece740/f15/li...>

Lecture 5 slides (ppt): <http://www.ece.cmu.edu/~ece740/f15/li...>

<https://www.youtube.com/onurmutlulectures>

# A More Compact Version...

---

- Computer Architecture, Spring 2015, Lecture 5
  - Advanced Branch Prediction (CMU, Spring 2015)
  - <https://www.youtube.com/watch?v=yDjsr-jTOtk&list=PL5PHm2jkkXmgVhh8CHAu9N76TShJqfYDt&index=4>



# Aside: ISA Translation

---

- One can translate from one ISA to another *internal-ISA* to get to a better tradeoff space
  - Programmer-visible ISA (virtual ISA) → Implementation ISA
  - Complex instructions (CISC) → Simple instructions (RISC)
  - Scalar ISA → VLIW ISA
- Examples
  - Intel's and AMD's x86 implementations translate x86 instructions into programmer-invisible microoperations (simple instructions) in hardware
  - Transmeta's x86 implementations translated x86 instructions into "secret" VLIW instructions in software (code morphing software)
- Think about the tradeoffs

# Transmeta: x86 to VLIW Translation

---

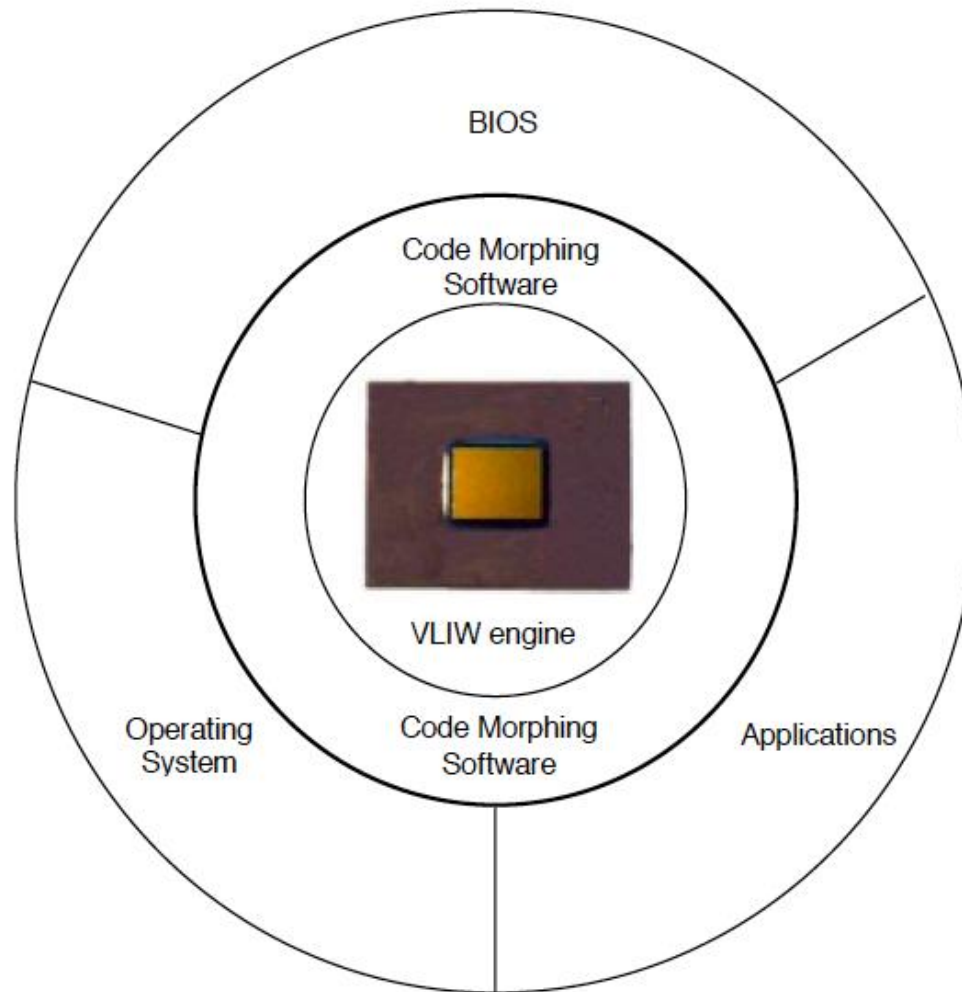
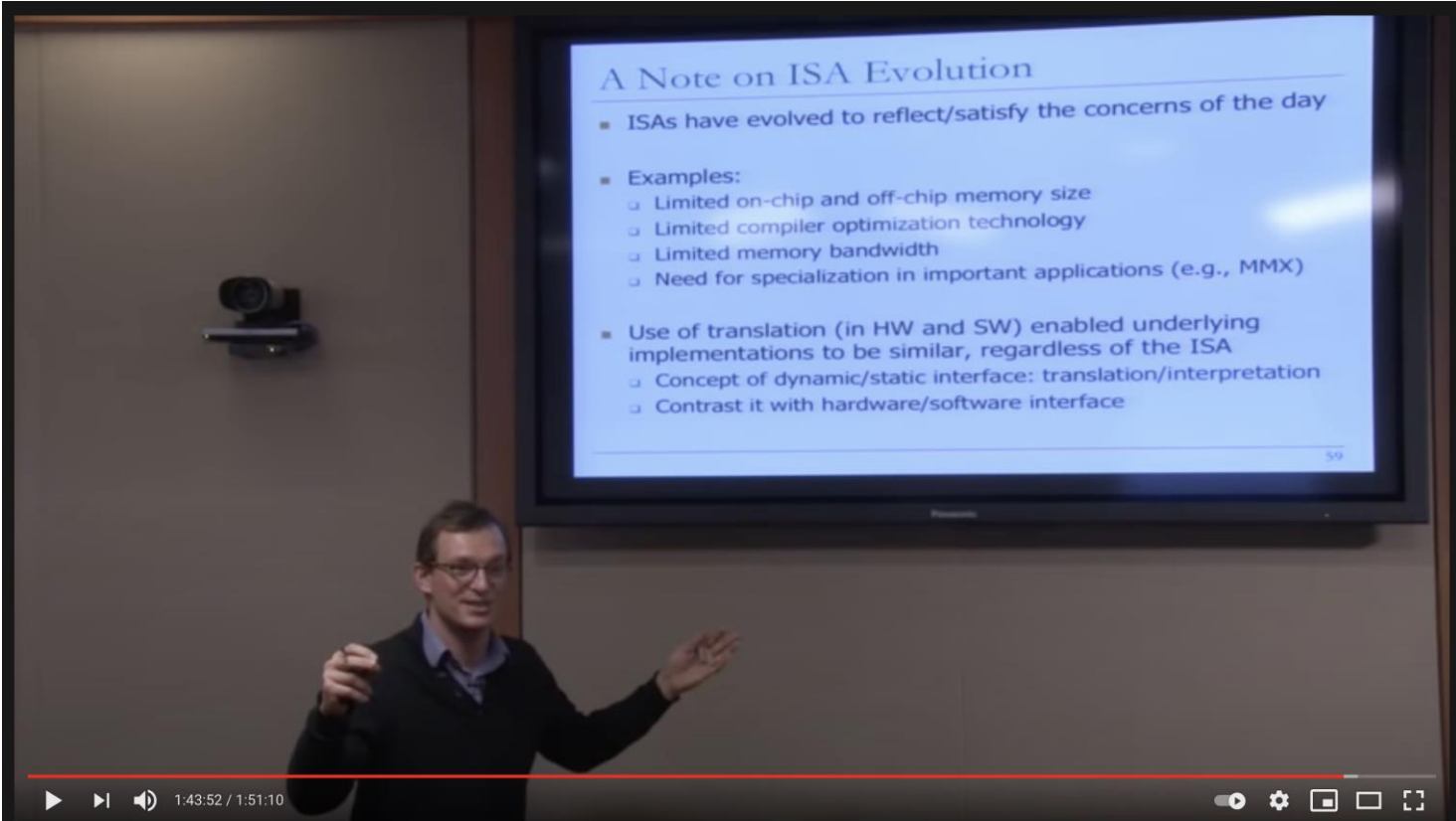


Figure 5. The Code Morphing software mediates between x86 software and the Crusoe processor.

Klaiber, [“The Technology Behind Crusoe Processors,”](#) Transmeta White Paper 2000.

---

# There Is A Lot More to Cover on ISAs



**A Note on ISA Evolution**

- ISAs have evolved to reflect/satisfy the concerns of the day
- Examples:
  - Limited on-chip and off-chip memory size
  - Limited compiler optimization technology
  - Limited memory bandwidth
  - Need for specialization in important applications (e.g., MMX)
- Use of translation (in HW and SW) enabled underlying implementations to be similar, regardless of the ISA
  - Concept of dynamic/static interface: translation/interpretation
  - Contrast it with hardware/software interface

59

Lecture 3. ISA Tradeoffs - Carnegie Mellon - Computer Architecture 2015 - Onur Mutlu

44,973 views • Jan 24, 2015

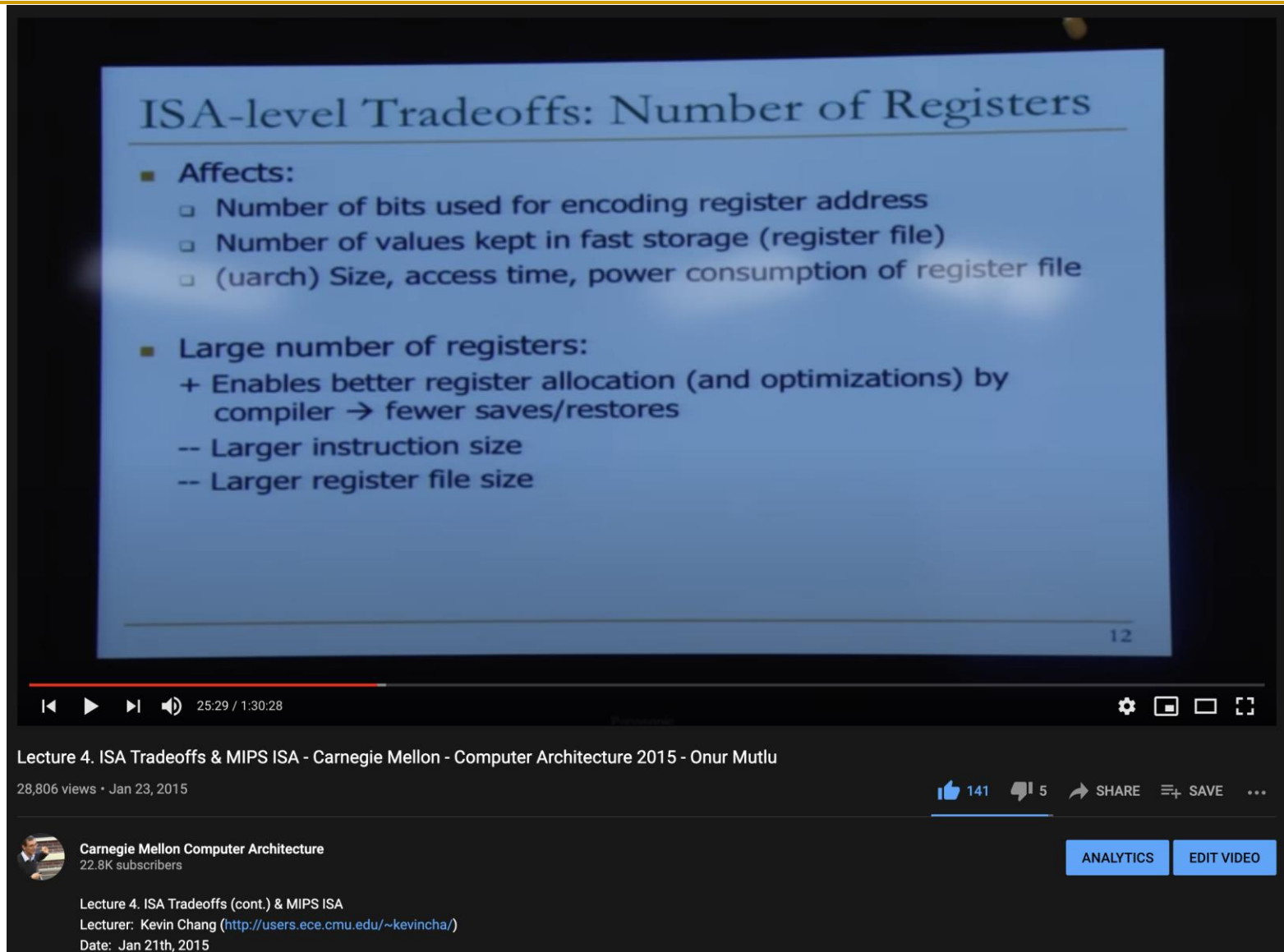
Carnegie Mellon Computer Architecture  
22.8K subscribers

Lecture 3. ISA Tradeoffs  
Lecturer: Prof. Onur Mutlu (<http://users.ece.cmu.edu/~omutlu/>)  
Date: Jan 16th, 2015

276 5 SHARE SAVE ...

ANALYTICS EDIT VIDEO

# There Is A Lot More to Cover on ISAs



The video player displays a slide with the following content:

## ISA-level Tradeoffs: Number of Registers


- Affects:
  - Number of bits used for encoding register address
  - Number of values kept in fast storage (register file)
  - (uarch) Size, access time, power consumption of register file
- Large number of registers:
  - + Enables better register allocation (and optimizations) by compiler → fewer saves/restores
  - Larger instruction size
  - Larger register file size

12

Lecture 4. ISA Tradeoffs & MIPS ISA - Carnegie Mellon - Computer Architecture 2015 - Onur Mutlu

28,806 views • Jan 23, 2015

141 5 SHARE SAVE ...

 Carnegie Mellon Computer Architecture  
22.8K subscribers

Lecture 4. ISA Tradeoffs (cont.) & MIPS ISA  
Lecturer: Kevin Chang (<http://users.ece.cmu.edu/~kevincha/>)  
Date: Jan 21th, 2015

ANALYTICS EDIT VIDEO

# Detailed Lectures on ISAs & ISA Tradeoffs

---

## ■ Computer Architecture, Spring 2015, Lecture 3

- ISA Tradeoffs (CMU, Spring 2015)
- <https://www.youtube.com/watch?v=QKdiZSfwg-g&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=3>

## ■ Computer Architecture, Spring 2015, Lecture 4

- ISA Tradeoffs & MIPS ISA (CMU, Spring 2015)
- <https://www.youtube.com/watch?v=RBgeCCW5Hjs&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=4>

## ■ Computer Architecture, Spring 2015, Lecture 2

- Fundamental Concepts and ISA (CMU, Spring 2015)
- <https://www.youtube.com/watch?v=NpC39uS4K4o&list=PL5PHm2jkkXmi5CxxI7b3JCL1TWybTDtKq&index=2>

# Digital Design & Computer Arch.

## Lecture 19a: VLIW

Prof. Onur Mutlu

ETH Zürich

Spring 2021

7 May 2021

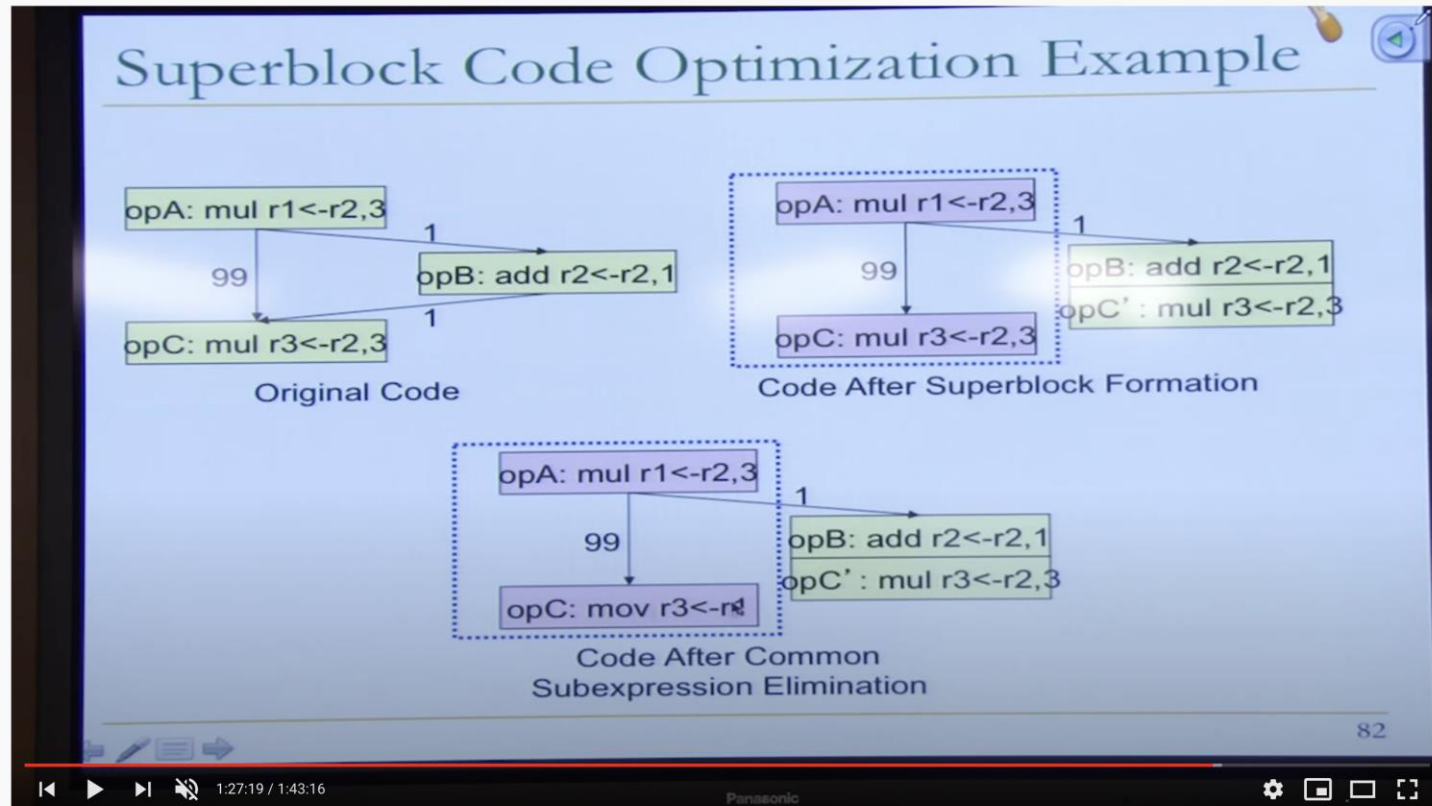
# Backup Slides

## (for Further Study)

# Issues in Fast & Wide Fetch Engines



# These Issues Covered in This Lecture...



18-740 Computer Architecture - Advanced Branch Prediction - Lecture 5

4,696 views • Sep 23, 2015

41 0 SHARE SAVE ...



Carnegie Mellon Computer Architecture  
23K subscribers

Lecture 5: Advanced Branch Prediction

Lecturer: Prof. Onur Mutlu (<http://users.ece.cmu.edu/~omutlu/>)

Date: September 16, 2014.

Lecture 5 slides (pdf): <http://www.ece.cmu.edu/~ece740/f15/li...>

Lecture 5 slides (ppt): <http://www.ece.cmu.edu/~ece740/f15/li...>

<https://www.youtube.com/onurmutlulectures>

# These Issues Covered in This Lecture...

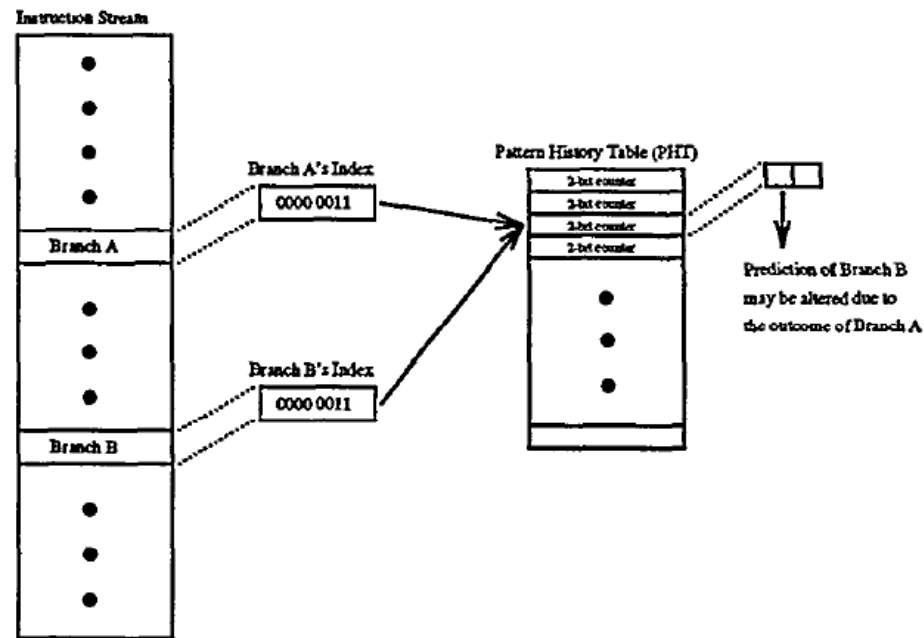
---

- Computer Architecture, Spring 2015, Lecture 5
  - Advanced Branch Prediction (CMU, Spring 2015)
  - <https://www.youtube.com/watch?v=yDjsr-jTOtk&list=PL5PHm2jkkXmgVhh8CHAu9N76TShJqfYDt&index=4>

# Interference in Branch Predictors

# An Issue: Interference in the PHTs

- Sharing the PHTs between histories/branches leads to interference
  - Different branches map to the same PHT entry and modify it
  - Interference can be positive, **negative**, or neutral



- Interference can be eliminated by dedicating a PHT per branch
  - Too much hardware cost
- How else can you eliminate or reduce interference?

# Reducing Interference in PHTs (I)

---

- Increase size of PHT
- Branch filtering
  - Predict highly-biased branches separately so that they do not consume PHT entries
  - E.g., static prediction or BTB based prediction
- Hashing/index-randomization
  - Gshare
  - Gskew
- Agree prediction

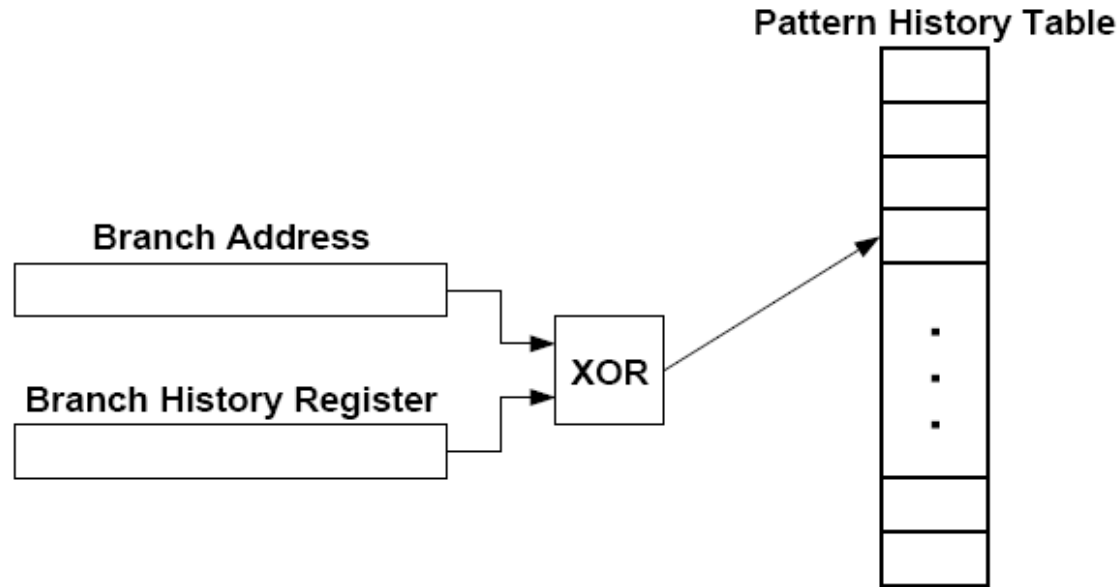
# Biased Branches and Branch Filtering

---

- Observation: Many branches are biased in one direction (e.g., 99% taken)
- Problem: These branches *pollute* the branch prediction structures → make the prediction of other branches difficult by causing “interference” in branch prediction tables and history registers
- Solution: Detect such biased branches, and predict them with a simpler predictor (e.g., last time, static, ...)
- Chang et al., “Branch classification: a new mechanism for improving branch predictor performance,” MICRO 1994.

# Reducing Interference: Gshare

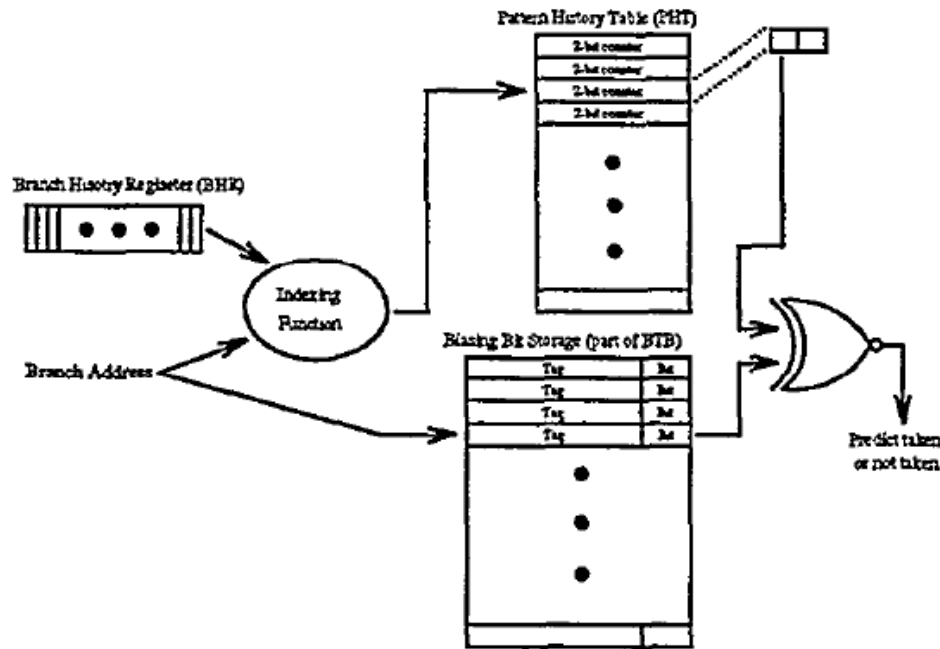
- Idea 1: Randomize the indexing function into the PHT such that probability of two branches mapping to the same entry reduces
  - Gshare predictor: GHR hashed with the Branch PC
    - + Better utilization of PHT + More context information
    - Increases access latency



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

# Reducing Interference: Agree Predictor

- Idea 2: Agree prediction
  - Each branch has a “bias” bit associated with it in BTB
    - Ideally, most likely outcome for the branch
  - High bit of the PHT counter indicates whether or not the prediction agrees with the bias bit (not whether or not prediction is taken)
- + Reduces negative interference (Why???)
- Requires determining bias bits (compiler vs. hardware)



Sprangle et al., “The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference,” ISCA 1997.



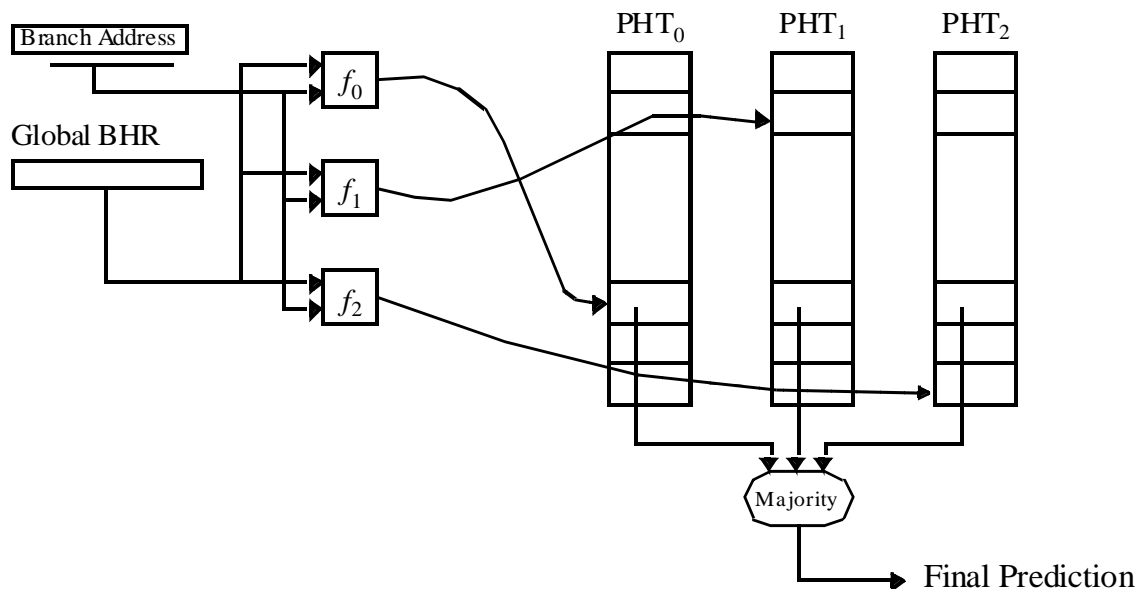
# Why Does Agree Prediction Make Sense?

---

- Assume two branches have taken rates of 85% and 15%.
- Assume they conflict in the PHT
- Let's compute the **probability they have opposite outcomes**
  - Baseline predictor:
    - $P(b1\ T, b2\ NT) + P(b1\ NT, b2\ T)$   
 $= (85\% * 85\%) + (15\% * 15\%) = 74.5\%$
  - Agree predictor:
    - Assume bias bits are set to T (b1) and NT (b2)
    - $P(b1\ agree, b2\ disagree) + P(b1\ disagree, b2\ agree)$   
 $= (85\% * 15\%) + (15\% * 85\%) = 25.5\%$
- Works because most branches are biased (not 50% taken)

# Reducing Interference: Gskew

- Idea 3: Gskew predictor
  - Multiple PHTs
  - Each indexed with a different type of hash function
  - Final prediction is a majority vote
- + Distributes interference patterns in a more randomized way (interfering patterns less likely in different PHTs at the same time)
- More complexity (due to multiple PHTs, hash functions)



Seznec, “An optimized 2bcgskew branch predictor,” IRISA Tech Report 1993.

Michaud, “Trading conflict and capacity aliasing in conditional branch predictors,” ISCA 1997

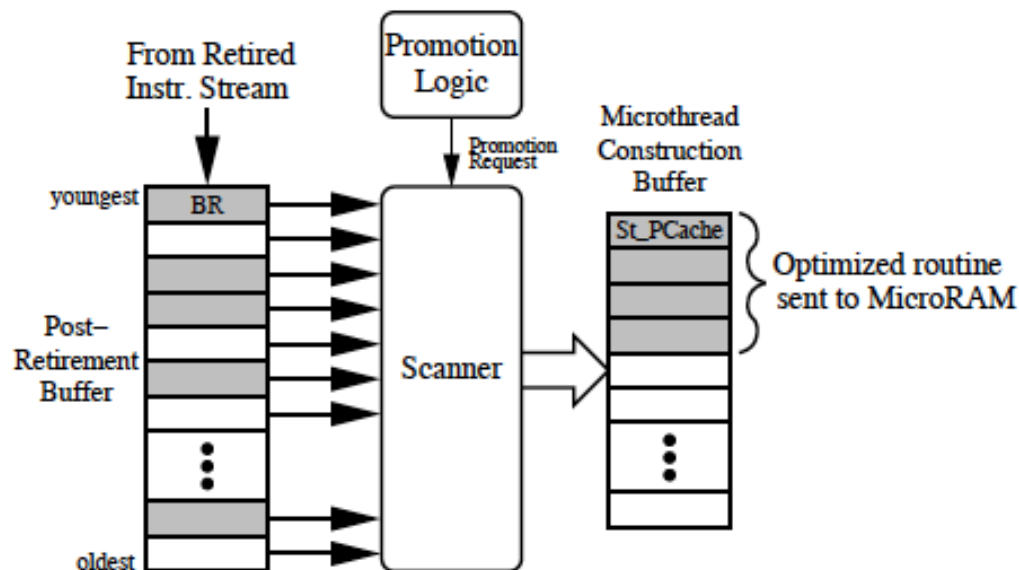
# More Techniques to Reduce PHT Interference

---

- The bi-mode predictor
  - Separate PHTs for mostly-taken and mostly-not-taken branches
  - Reduces negative aliasing between them
  - Lee et al., “The bi-mode branch predictor,” MICRO 1997.
- The YAGS predictor
  - Use a small tagged “cache” to predict branches that have experienced interference
  - Aims to not to mispredict them again
  - Eden and Mudge, “The YAGS branch prediction scheme,” MICRO 1998.
- Alpha EV8 (21464) branch predictor
  - Seznec et al., “Design tradeoffs for the Alpha EV8 conditional branch predictor,” ISCA 2002.

# Another Direction: Helper Threading

- Idea: Pre-compute the outcome of the branch with a separate, customized thread (i.e., a helper thread)



**Figure 3. The Microthread Builder**

- Chappell et al., “Difficult-Path Branch Prediction Using Subordinate Microthreads,” ISCA 2002.
- Chappell et al., “Simultaneous Subordinate Microthreading,” ISCA 1999.

# Issues in Wide & Fast Fetch

# I-Cache Line and Way Prediction

---

- Problem: Complex branch prediction can take too long (many cycles)
- Goal
  - Quickly generate (a reasonably accurate) next fetch address
  - Enable the fetch engine to run at high frequencies
  - Override the quick prediction with more sophisticated prediction
- Idea: Get the predicted next cache line and way at the time you fetch the current cache line
- Example Mechanism (e.g., Alpha 21264)
  - Each cache line tells which line/way to fetch next (prediction)
  - On a fill, line/way predictor points to next sequential line
  - On branch resolution, line/way predictor is updated
  - If line/way prediction is incorrect, one cycle is wasted

# Alpha 21264 Line & Way Prediction

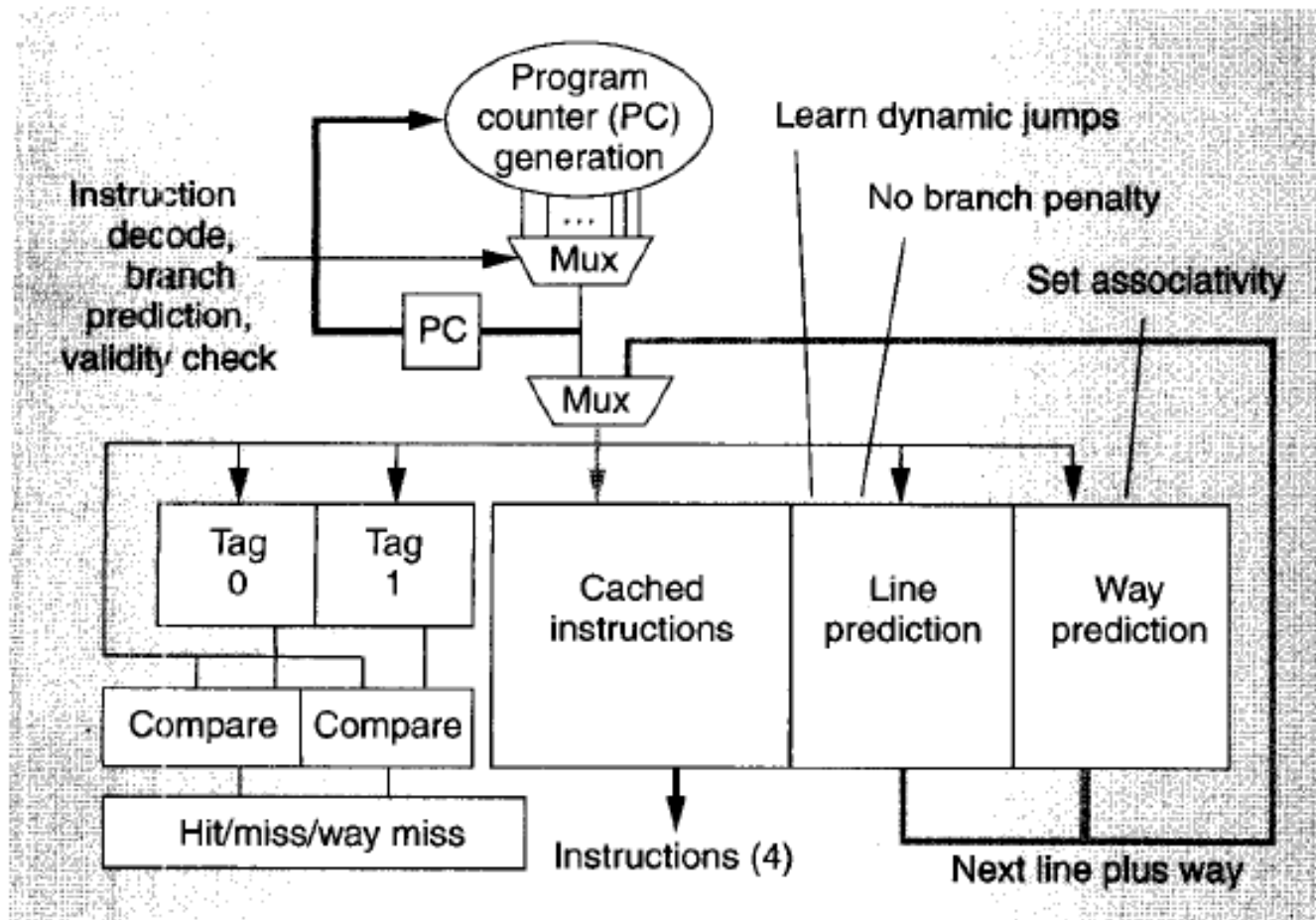
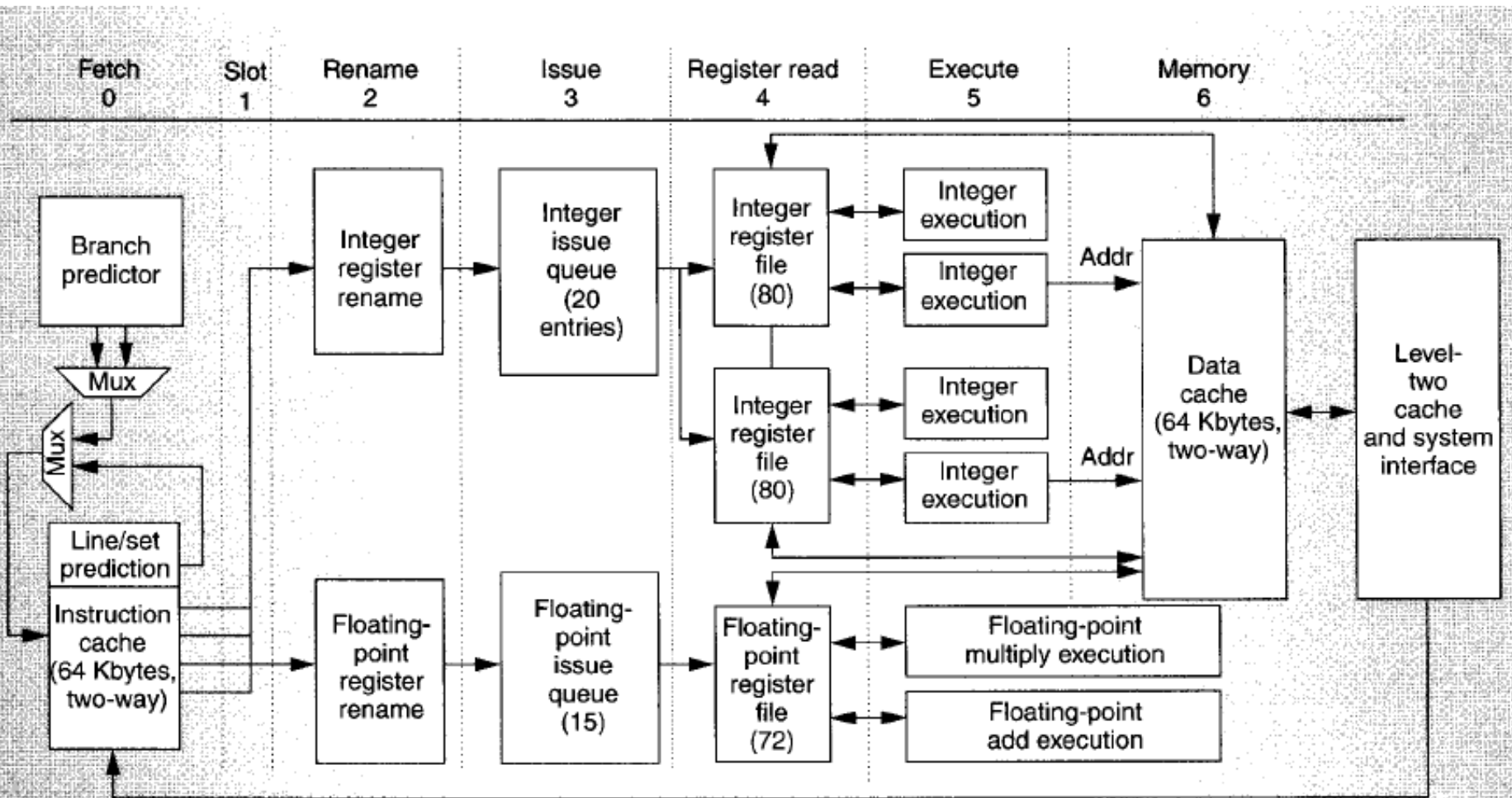


Figure 3. Alpha 21264 instruction fetch. The line and way prediction (wrap-around path on the right side) provides a fast instruction fetch path that avoids common fetch stalls when the predictions are correct.

# Alpha 21264 Line & Way Prediction





# Issues in Wide Fetch Engines

---

- Wide Fetch: Fetch multiple instructions per cycle
- Superscalar
- VLIW
- SIMT (GPUs' single-instruction multiple thread model)
- Wide fetch engines suffer from the branch problem:
  - How do you feed the wide pipeline with useful instructions in a single cycle?
  - What if there is a taken branch in the "fetch packet"?
  - What if there are "multiple (taken) branches" in the "fetch packet"?

# Fetching Multiple Instructions Per Cycle

---

- Two problems

1. **Alignment** of instructions in I-cache

- ❑ What if there are not enough (N) instructions in the cache line to supply the fetch width?

2. **Fetch break**: Branches present in the fetch block

- ❑ Fetching sequential instructions in a single cycle is easy
- ❑ What if there is a control flow instruction in the N instructions?
- ❑ Problem: **The direction of the branch is not known but we need to fetch more instructions**

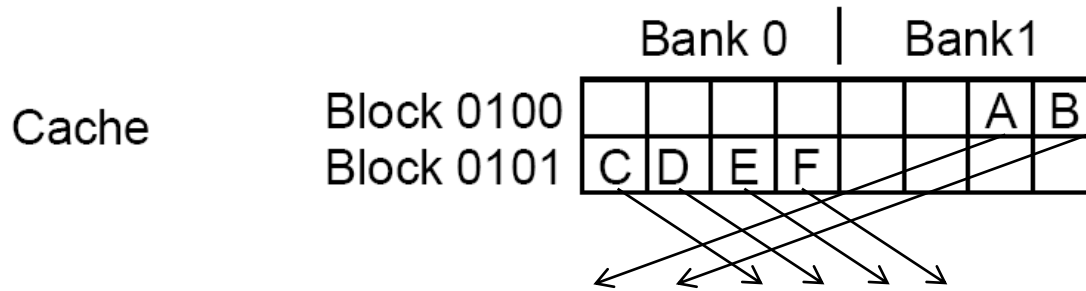
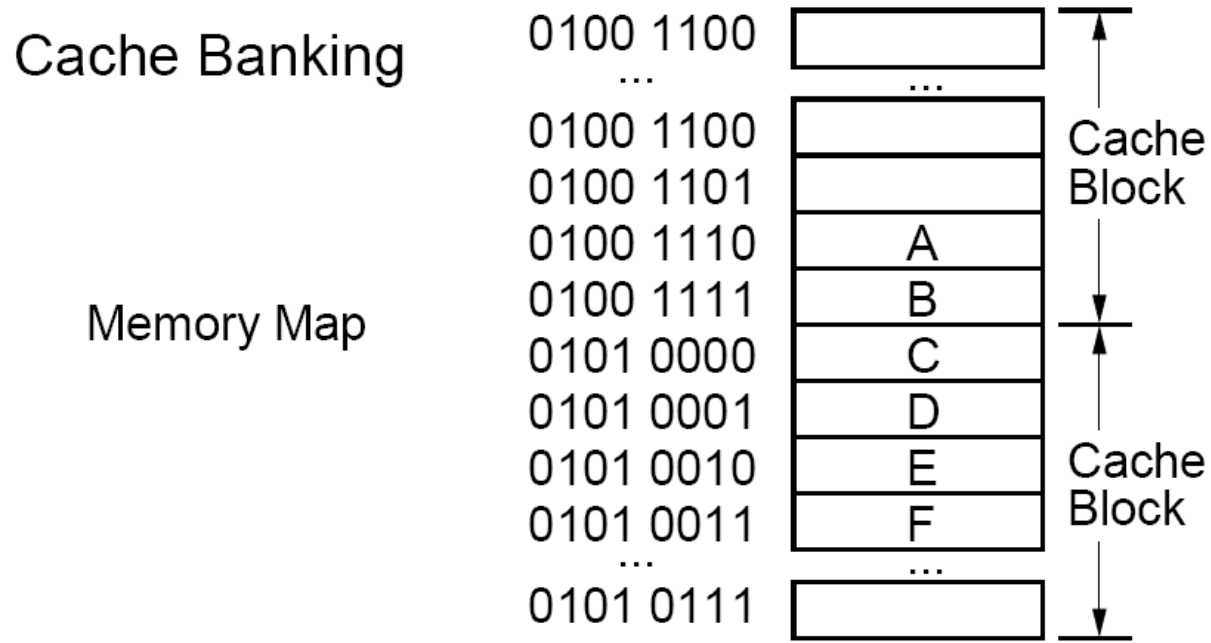
- These can cause effective fetch width < peak fetch width

# Wide Fetch Solutions: Alignment

---

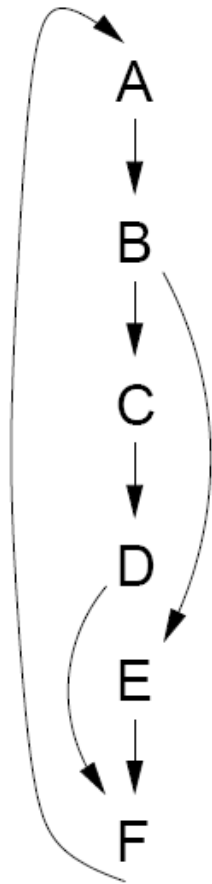
- **Large cache blocks:** Hope N instructions contained in the block
- **Split-line fetch:** If address falls into second half of the cache block, fetch the first half of next cache block as well
  - ❑ Enabled by banking of the cache
  - ❑ Allows sequential fetch across cache blocks in one cycle
  - ❑ Intel Pentium and AMD K5

# Split Line Fetch



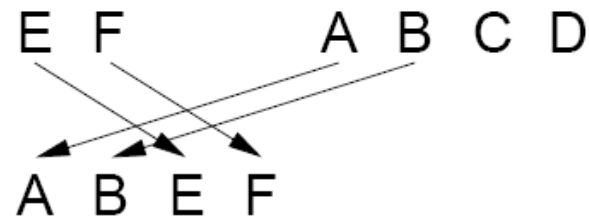
Need alignment logic:

# Short Distance Predicted-Taken Branches

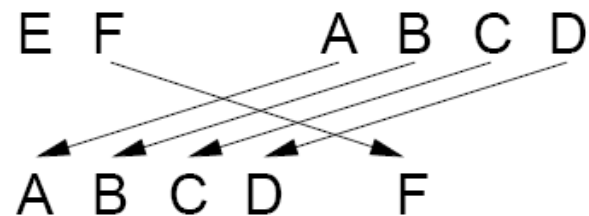


	Bank 0				Bank 1			
Block 0100					A	B	C	D
Block 0101	E	F						

First Iteration (Branch B taken to E)



Second Iteration (Branch B fall through to C)



# Techniques to Reduce Fetch Breaks

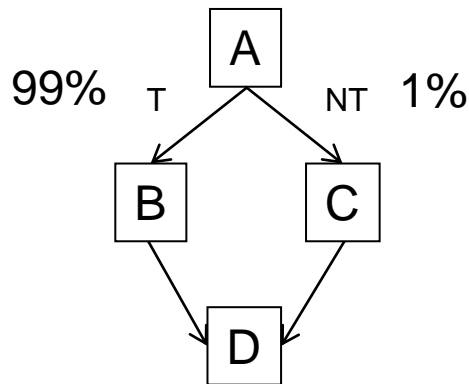
---

- Compiler
  - Code reordering (basic block reordering)
  - Superblock
- Hardware
  - Trace cache
- Hardware/software cooperative
  - Block structured ISA

# Basic Block Reordering

- Not-taken control flow instructions not a problem: no fetch break: **make the likely path the not-taken path**
- Idea: **Convert taken branches to not-taken ones**
  - i.e., **reorder basic blocks** (after profiling)
  - Basic block: code with a single entry and single exit point

Control Flow Graph



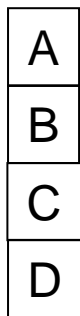
Code Layout 1



Code Layout 2



Code Layout 3



- Code Layout 1 leads to the fewest fetch breaks

# Basic Block Reordering

---

- Pettis and Hansen, “**Profile Guided Code Positioning**,” PLDI 1990.
- Advantages:
  - + Reduced fetch breaks (assuming profile behavior matches runtime behavior of branches)
  - + Increased I-cache hit rate
  - + Reduced page faults
- Disadvantages:
  - Dependent on compile-time profiling
  - Does not help if branches are not biased
  - Requires recompilation



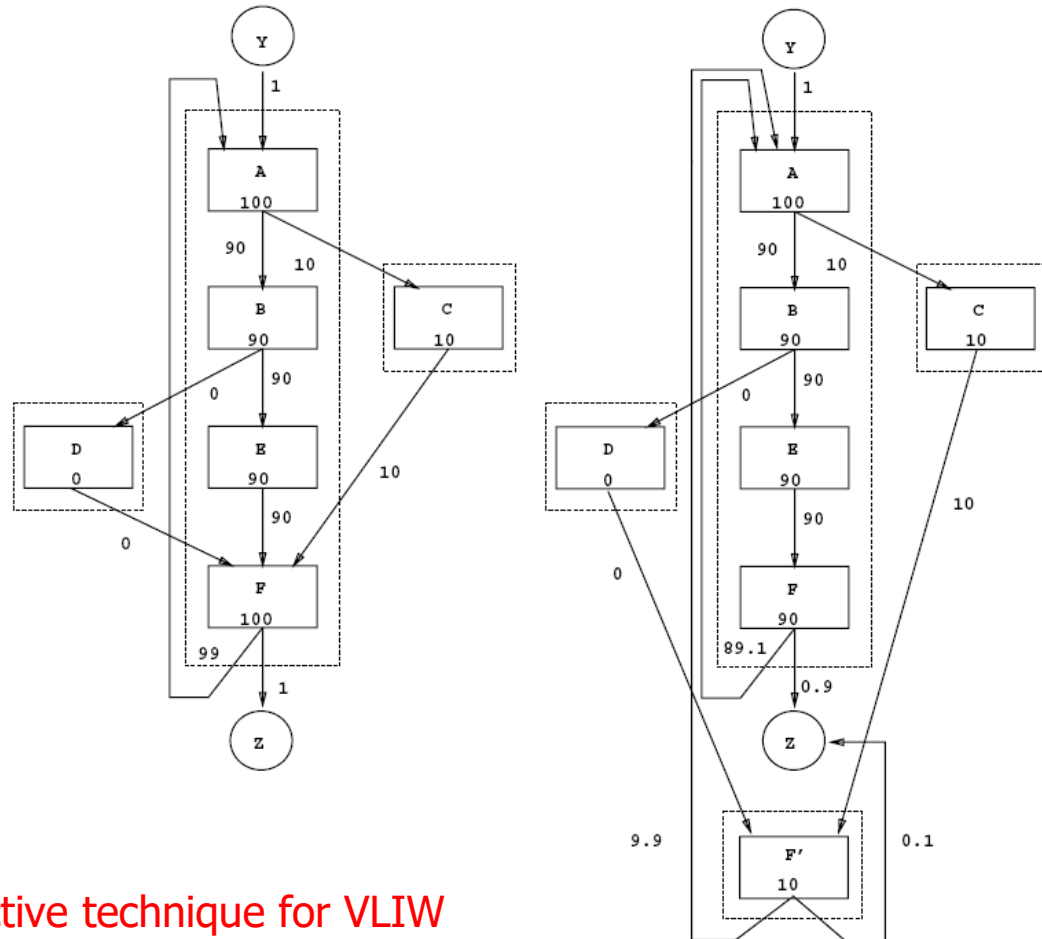
# Superblock

- Idea: Combine frequently executed basic blocks such that they form a **single-entry multiple exit larger block**, which is likely executed as straight-line code

- + Helps wide fetch
- + Enables aggressive compiler optimizations and code reordering within the superblock

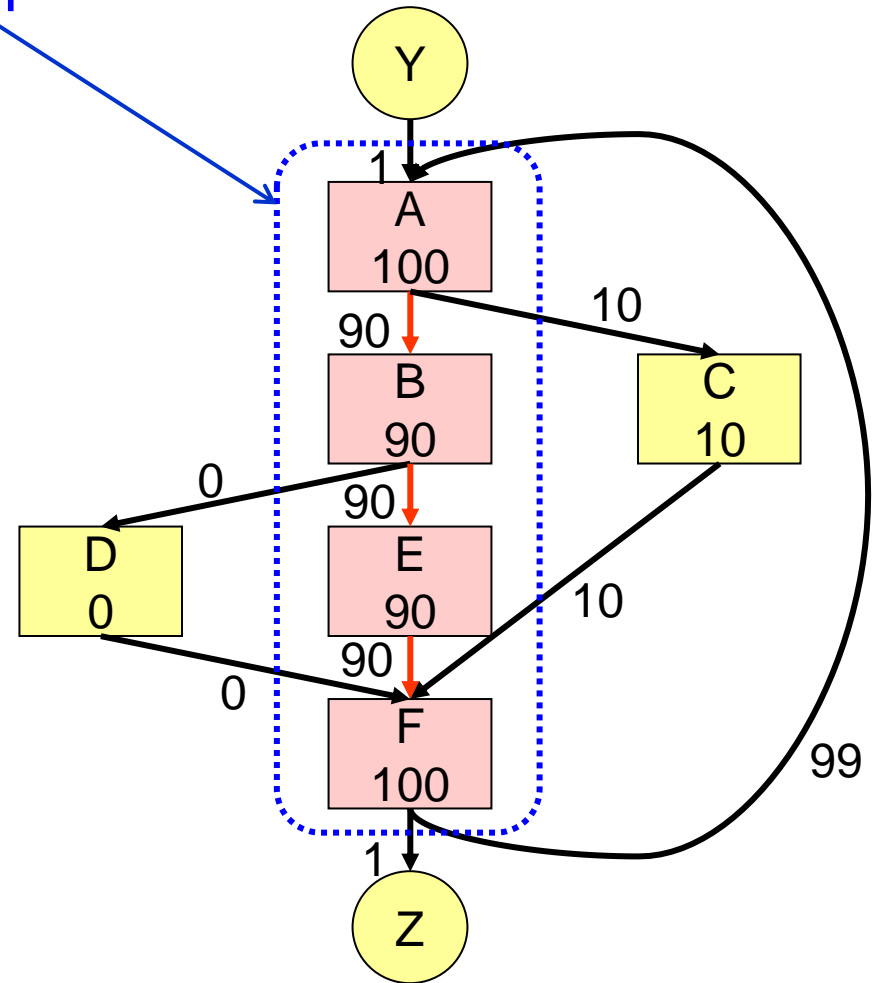
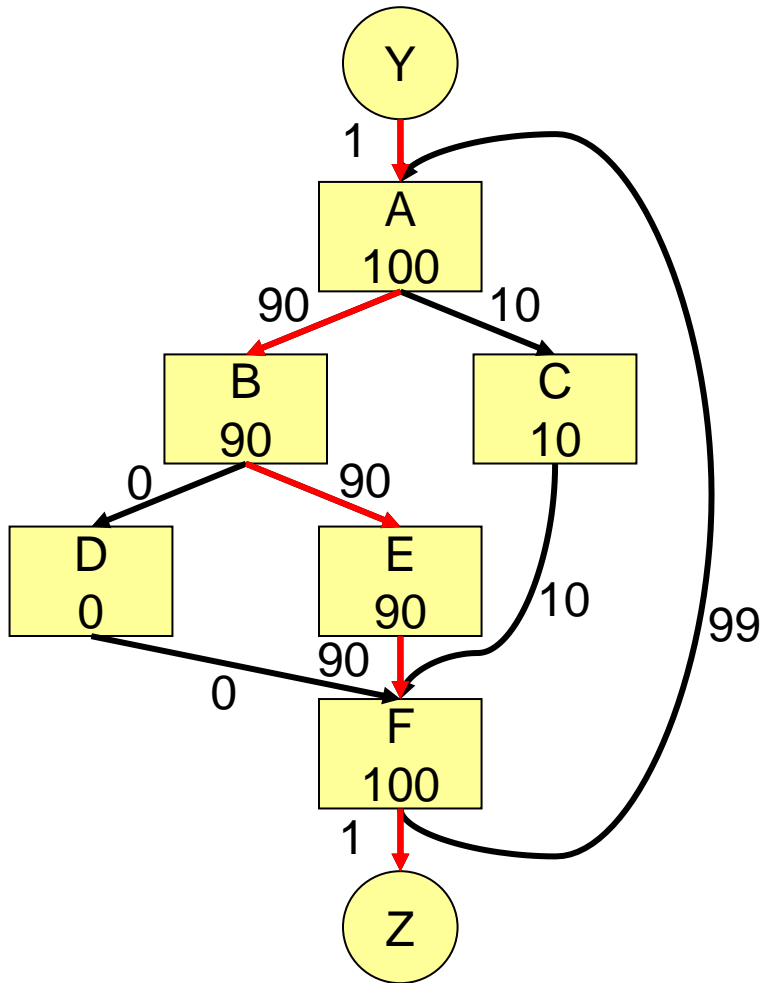
- Increased code size
- Profile dependent
- Requires recompilation

- Hwu et al. “**The Superblock: An effective technique for VLIW and superscalar compilation**,” Journal of Supercomputing, 1993.

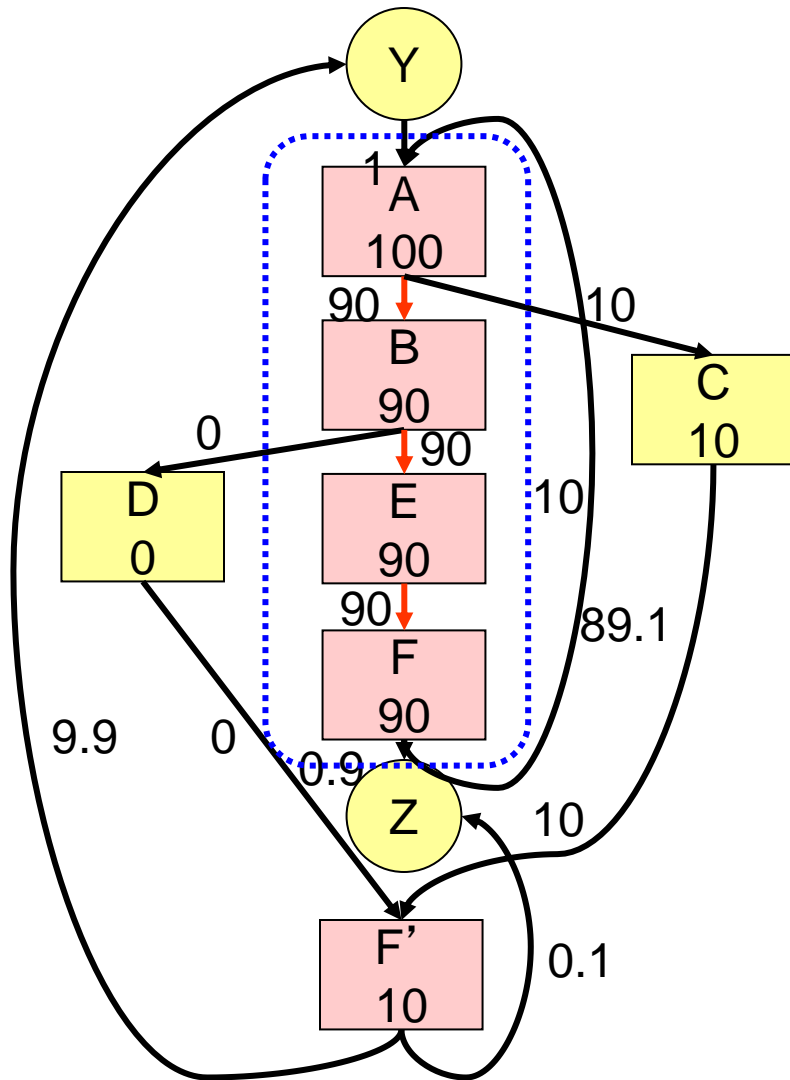


# Superblock Formation (I)

Is this a superblock?



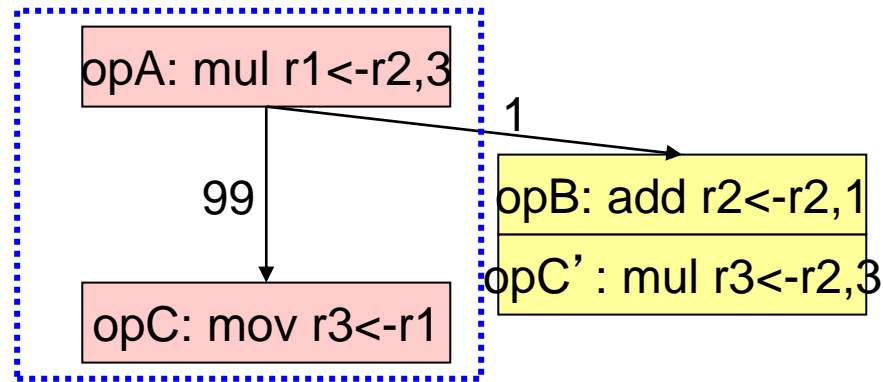
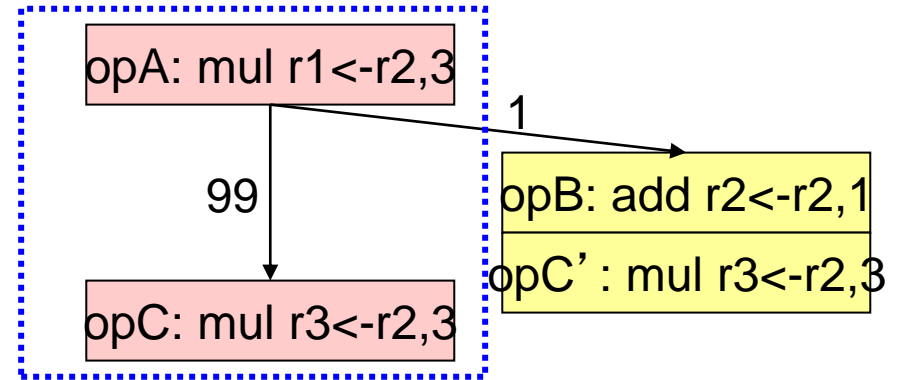
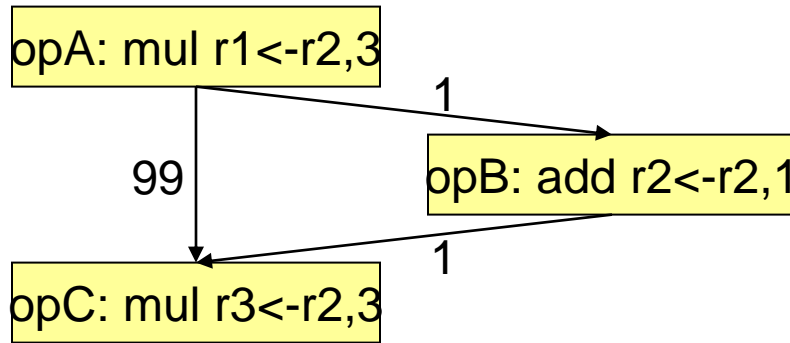
# Superblock Formation (II)



## Tail duplication:

duplication of basic blocks  
after a side entrance to  
eliminate side entrances  
→ transforms  
a **trace** into a **superblock**.

# Superblock Code Optimization Example



# Techniques to Reduce Fetch Breaks

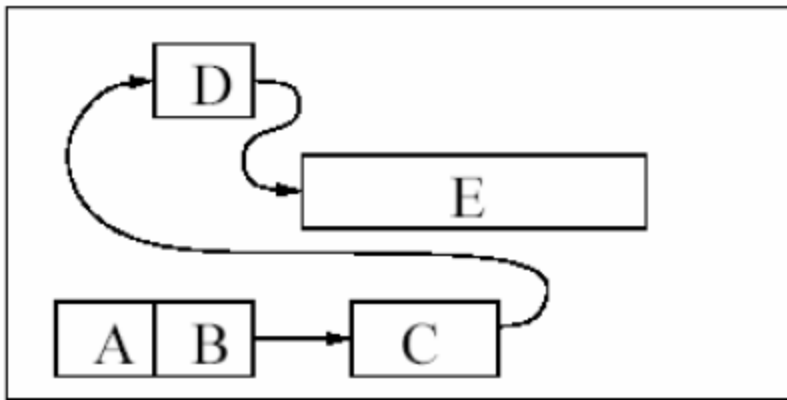
---

- Compiler
  - Code reordering (basic block reordering)
  - Superblock
- Hardware
  - Trace cache
- Hardware/software cooperative
  - Block structured ISA

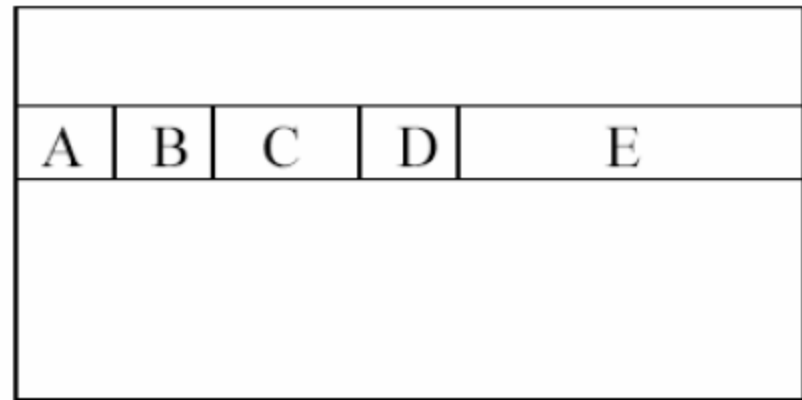
# Trace Cache: Basic Idea

---

- A trace is a sequence of executed instructions.
- It is specified by a start address and the branch outcomes of control transfer instructions.
- Traces repeat: programs have frequently executed paths
- Trace cache idea: Store the dynamic instruction sequence in the same physical location.



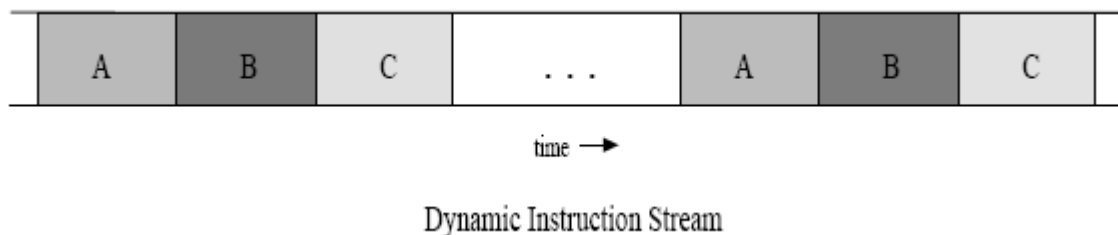
(a) Instruction cache.



(b) Trace cache.

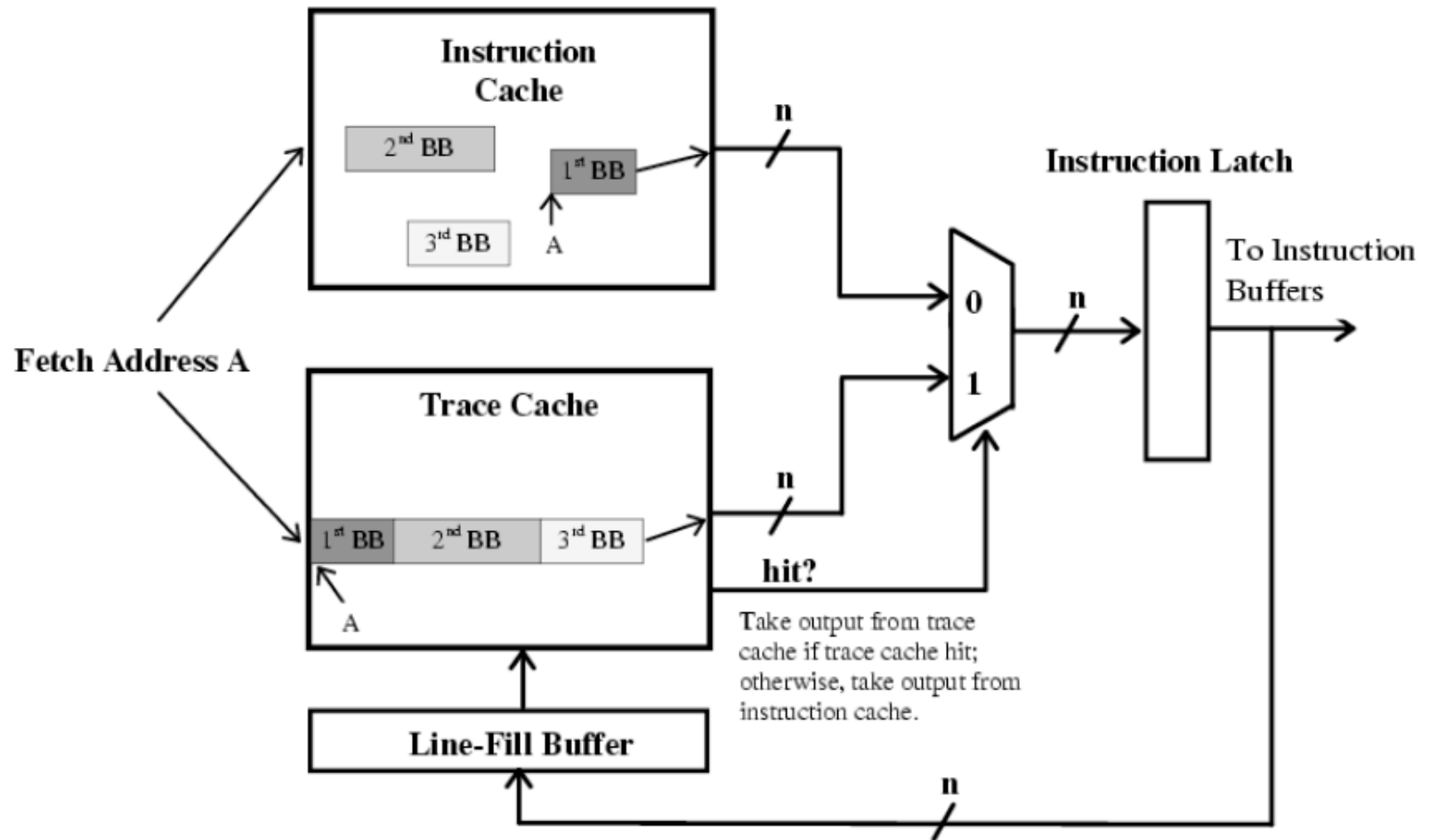
# Reducing Fetch Breaks: Trace Cache

- Dynamically determine the basic blocks that are executed consecutively
- Trace: Consecutively executed basic blocks
- Idea: Store consecutively-executed basic blocks in physically-contiguous internal storage (called trace cache)



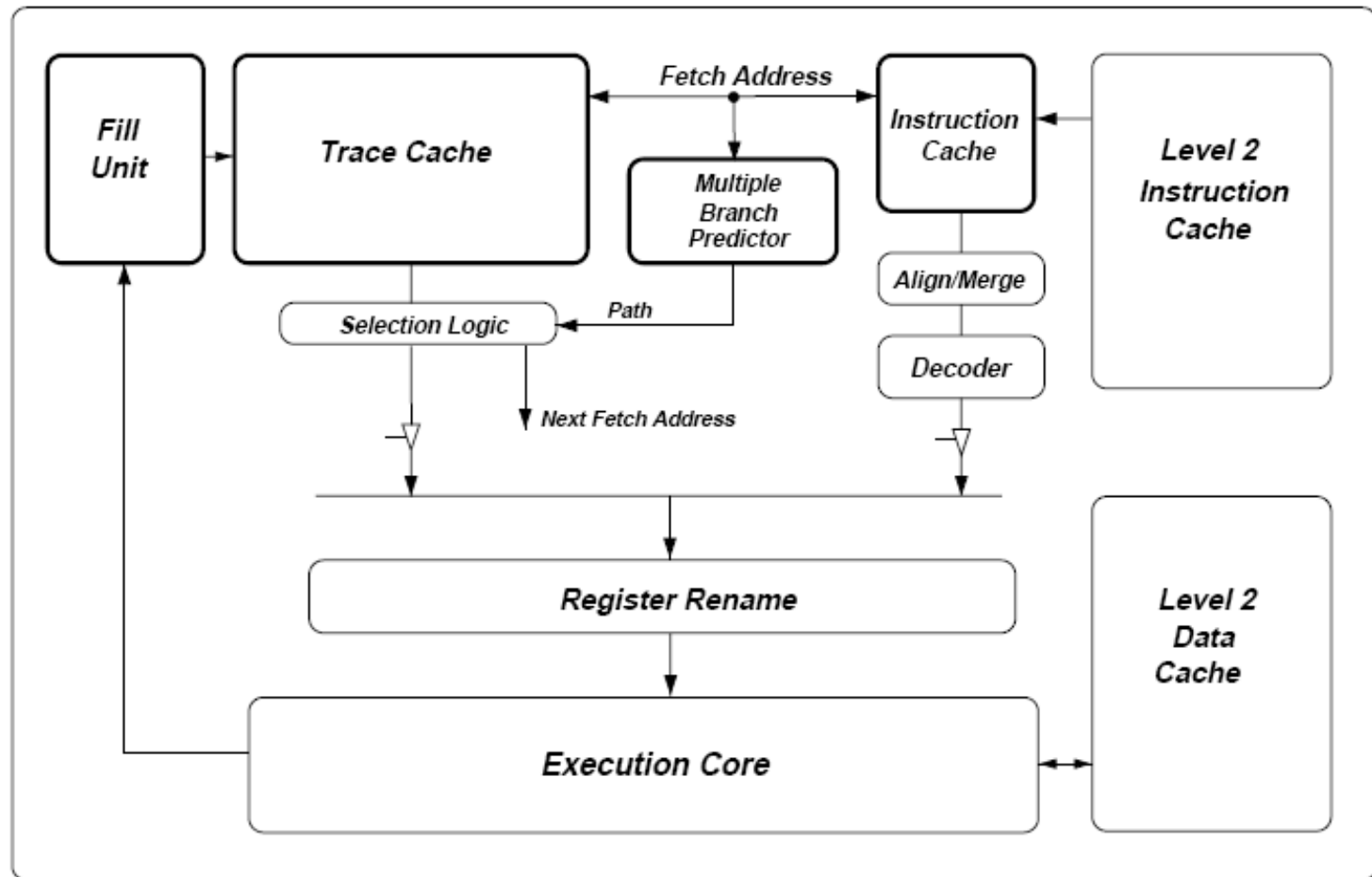
- Basic trace cache operation:
  - Fetch from consecutively-stored basic blocks (predict next trace or branches)
  - Verify the executed branch directions with the stored ones
  - If mismatch, flush the remaining portion of the trace
- Rotenberg et al., “Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching,” MICRO 1996.
- Patel et al., “Critical Issues Regarding the Trace Cache Fetch Mechanism,” Umich TR, 1997.

# Trace Cache: Example





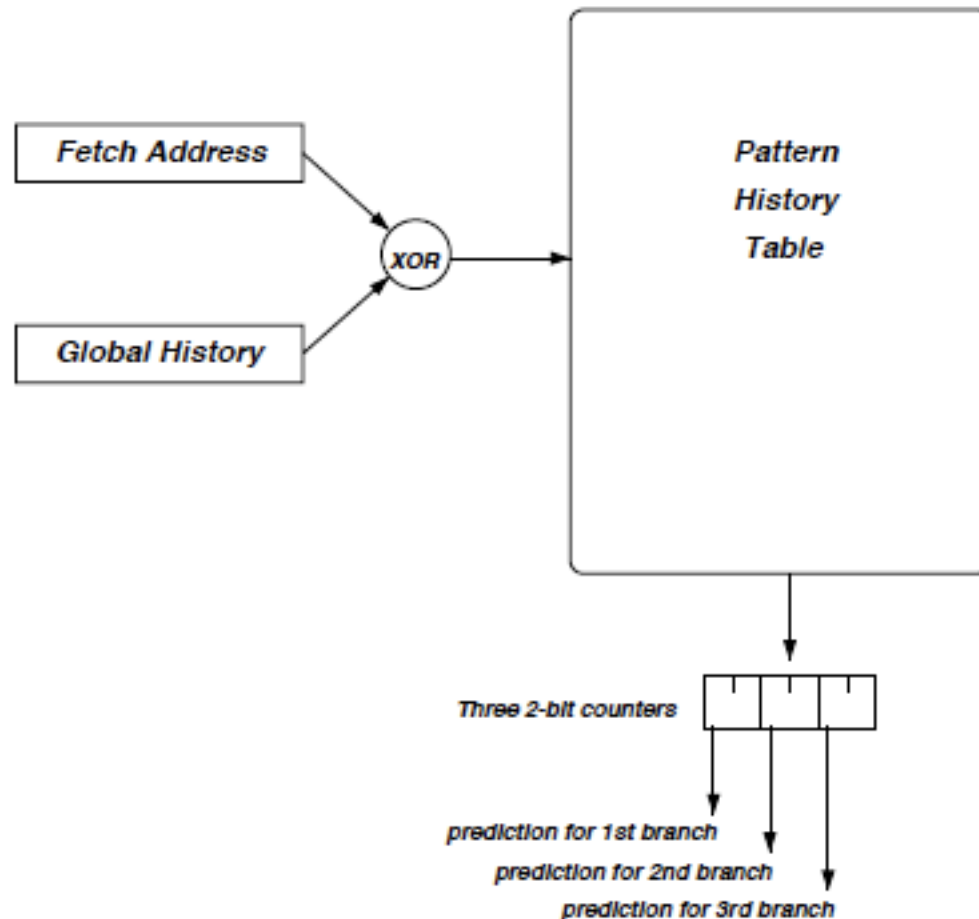
# An Example Trace Cache Based Processor



- From Patel's PhD Thesis: "**Trace Cache Design for Wide Issue Superscalar Processors**," University of Michigan, 1999.

# Multiple Branch Predictor

- S. Patel, “**Trace Cache Design for Wide Issue Superscalar Processors**,” PhD Thesis, University of Michigan, 1999.



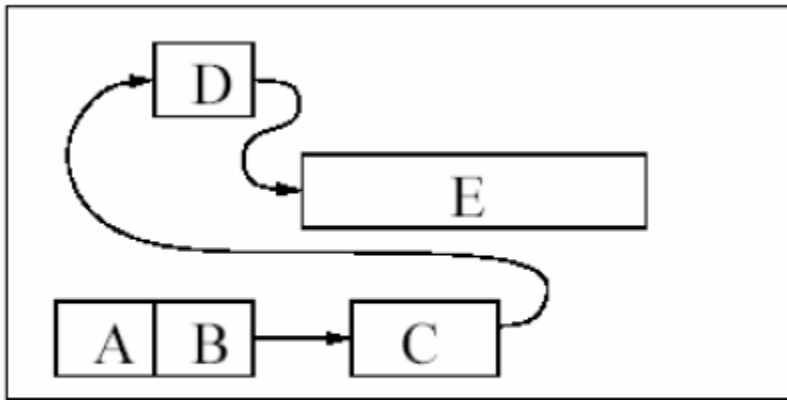
# What Does A Trace Cache Line Store?

---

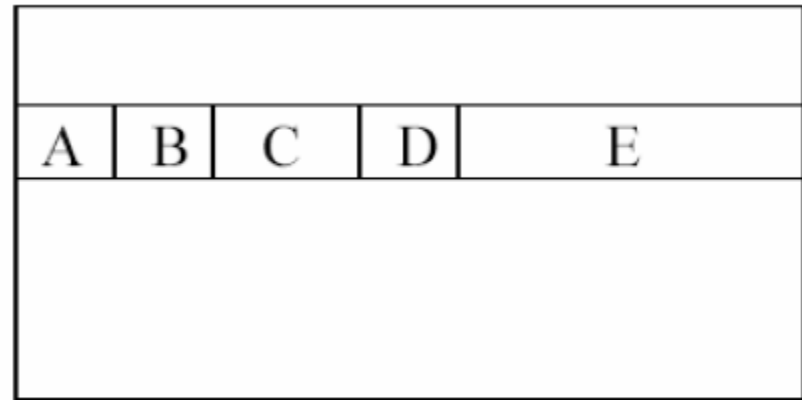
- 16 slots for instructions. Instructions are stored in decoded form and occupy approximately five bytes for a typical ISA. Up to three branches can be stored per line. Each instruction is marked with a two-bit tag indicating to which block it belongs.
- Four target addresses. With three basic blocks per segment and the ability to fetch partial segments, there are four possible targets to a segment. The four addresses are explicitly stored allowing immediate generation of the next fetch address, even for cases where only a partial segment matches.
- Path information. This field encodes the number and directions of branches in the segment and includes bits to identify whether a segment ends in a branch and whether that branch is a return from subroutine instruction. In the case of a return instruction, the return address stack provides the next fetch address.

- Patel et al., “Critical Issues Regarding the Trace Cache Fetch Mechanism,” Umich TR, 1997.

# Trace Cache: Advantages/Disadvantages



(a) Instruction cache.



(b) Trace cache.

- + Reduces fetch breaks (assuming branches are biased)
- + No need for decoding (instructions can be stored in decoded form)
- + Can enable dynamic optimizations within a trace
- Requires hardware to form traces (more complexity) → called fill unit
- Results in duplication of the same basic blocks in the cache
- Can require the prediction of multiple branches per cycle
  - If multiple cached traces have the same start address
  - What if XYZ and XYT are both likely traces?

# Intel Pentium 4 Trace Cache

---

- A 12K-uop trace cache replaces the L1 I-cache
- Trace cache stores decoded and cracked instructions
  - Micro-operations (uops): returns 6 uops every other cycle
- x86 decoder can be simpler and slower
- A. Peleg, U. Weiser; "[Dynamic Flow Instruction Cache Memory Organized Around Trace Segments Independent of Virtual Address Line](#)", United States Patent No. 5,381,533, Jan 10, 1995

