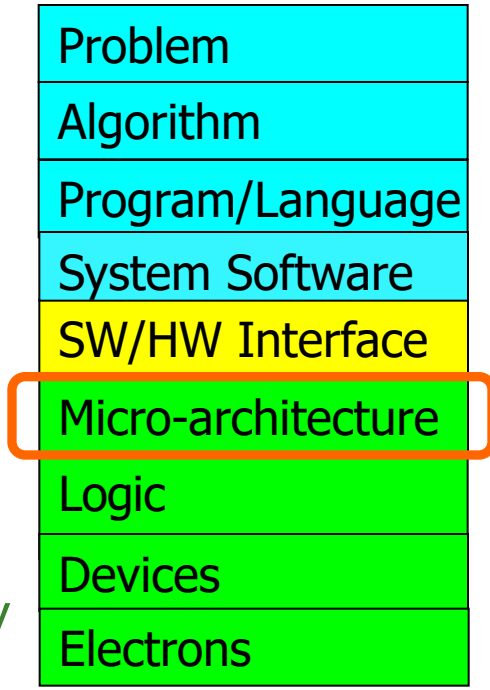
Computer Architecture
Lecture 27: VLIW Architectures

Prof. Onur Mutlu
ETH Zürich
Fall 2023
31 January 2024

We Are Done with All This!

- Microarchitecture Fundamentals
  - Single-cycle Microarchitectures
  - Multi-cycle Microarchitectures
- Pipelining & Precise Exceptions
  - Pipelining
  - Pipelined Processor Design
    - Control & Data Dependence Handling
    - Precise Exceptions: State Maintenance & Recovery
- Out-of-Order & Superscalar Execution
  - Out-of-Order Execution
  - Dataflow & Superscalar Execution
  - Branch Prediction



Approaches to (Instruction-Level) Concurrency

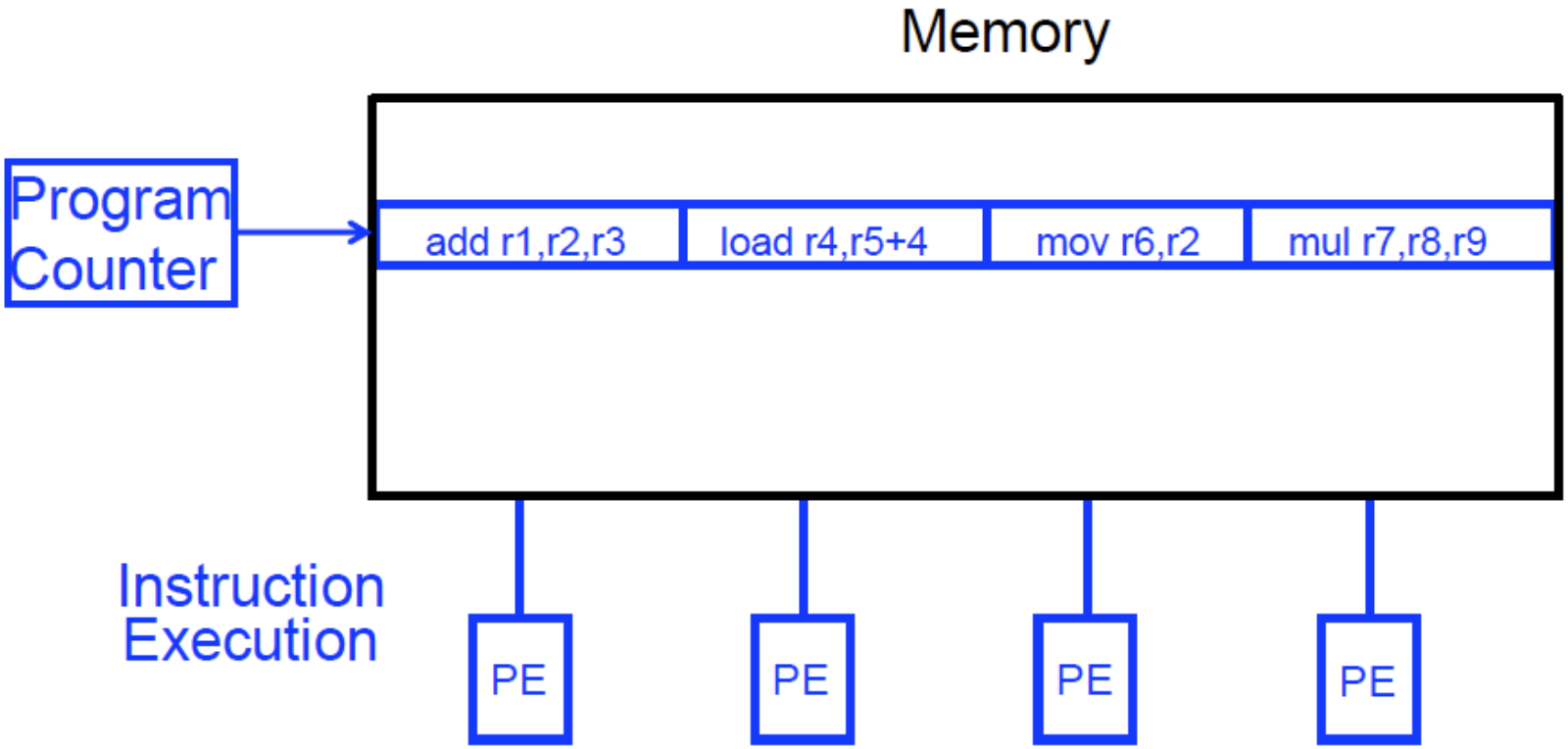
- Pipelining
- Fine-Grained Multithreading
- Out-of-order Execution
- Dataflow (at the ISA level)
- Superscalar Execution
- VLIW
- Systolic Arrays
- Decoupled Access Execute
- SIMD Processing (Vector and array processors, GPUs)

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
  - **Simple hardware, complex compiler**

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

  1. Multiple instruction fetch/execute, multiple functional units

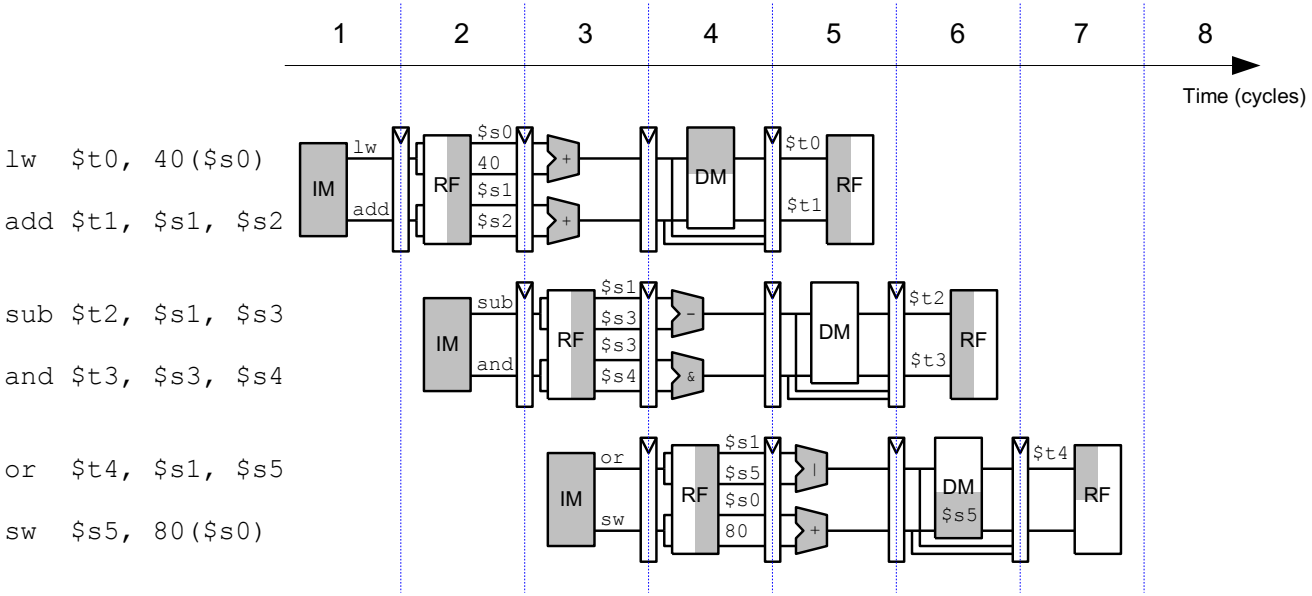
  2. All instructions in a bundle are executed in lock step

  3. Instructions in a bundle statically aligned to be directly supplied into the functional units

VLIW Performance Example (2-wide bundles)

```
lw  $t0, 40($s0)     add $t1, $s1, $s2    Bundle 1
sub $t2, $s1, $s3    and $t3, $s3, $s4    Bundle 2
or  $t4, $s1, $s5    sw  $s5, 80($s0)     Bundle 3
```

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].

Fisher et al., "Parallel Processing: A Smart Compiler and a Dumb Machine," CC 1984.

VLIW Philosophy & Principles

- Philosophy similar to RISC (simple instructions and hardware)
  - Except "multiple instructions in parallel: in VLIW

- 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 as possible
    - Executes each instruction in a bundle in lock step
    - Simple → higher frequency, easier to design, low power

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.

Fisher, "Very Long Instruction Word architectures and the ELI-512," ISCA 1983.

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

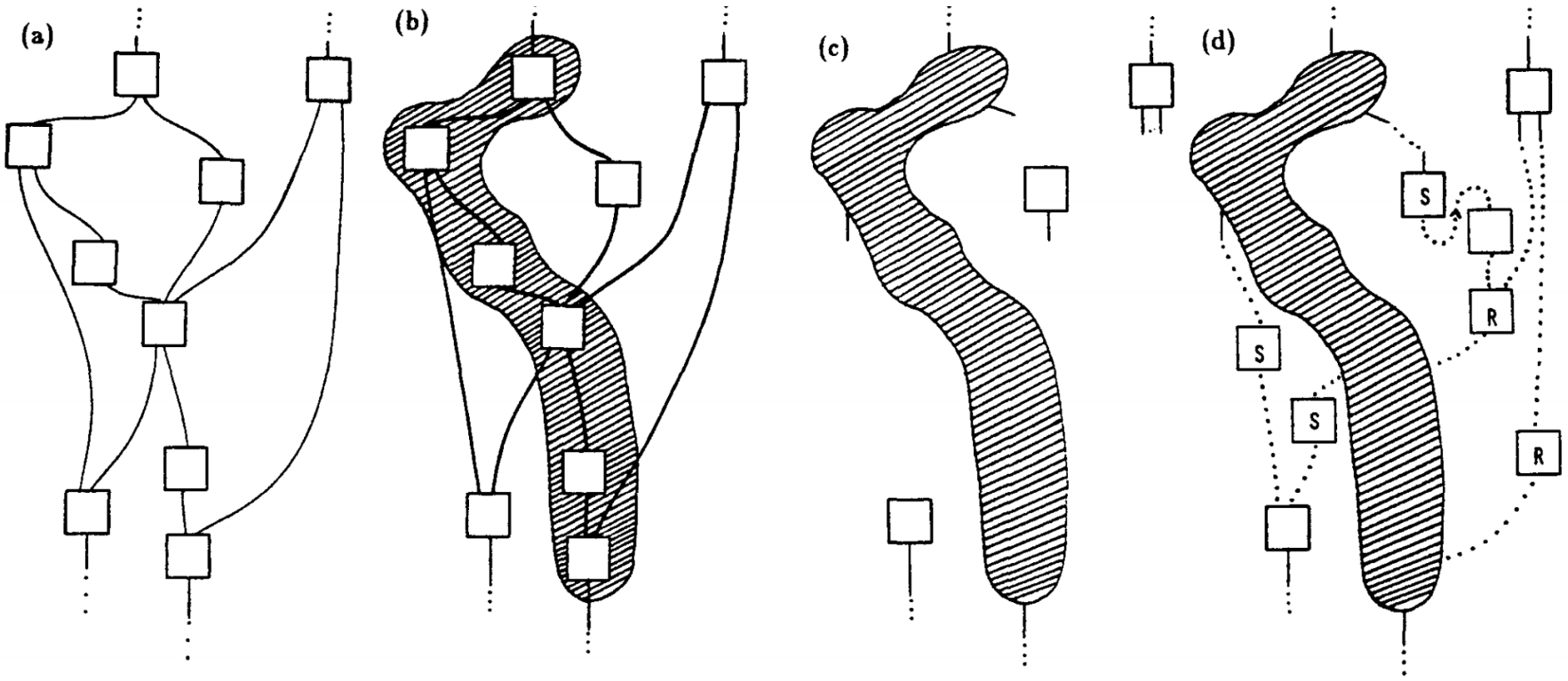
- VLIW simplifies hardware, but requires complex compiler techniques
- Solely-compiler approach of VLIW has several downsides that reduce performance
  - -- No tolerance for variable or long-latency operations (lock step)
  - -- Too many NOPs (not enough parallelism discovered)
  - -- Static schedule intimately tied to microarchitecture
    - -- Code optimized for one generation performs poorly for next

++ 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 relinking.

Fisher, "Very Long Instruction Word architectures and the ELI-512," ISCA 1983.

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.

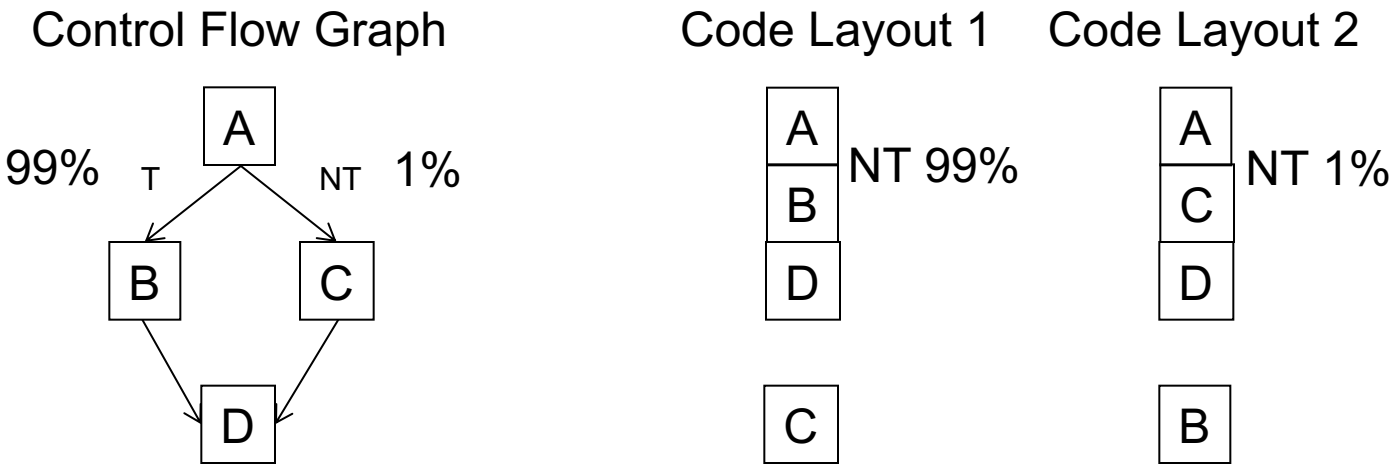
Each operation requires a small, statically predictable number of cycles to execute.

Operations can be pipelined. These properties distinguish

Fisher, "Very Long Instruction Word architectures and the ELI-512," ISCA 1983.

Recall: Basic Block Reordering

- Likely-taken branch instructions are a problem
  - They hurt the accuracy of "always not taken" branch prediction
  - They make static code reordering/scheduling difficult
- Idea: Convert likely-taken branch to a likely not-taken one
  - i.e., reorder basic blocks (after profiling)
  - Basic block: code with a single entry and single exit point



- Code Layout 1 leads to the fewest branch mispredictions

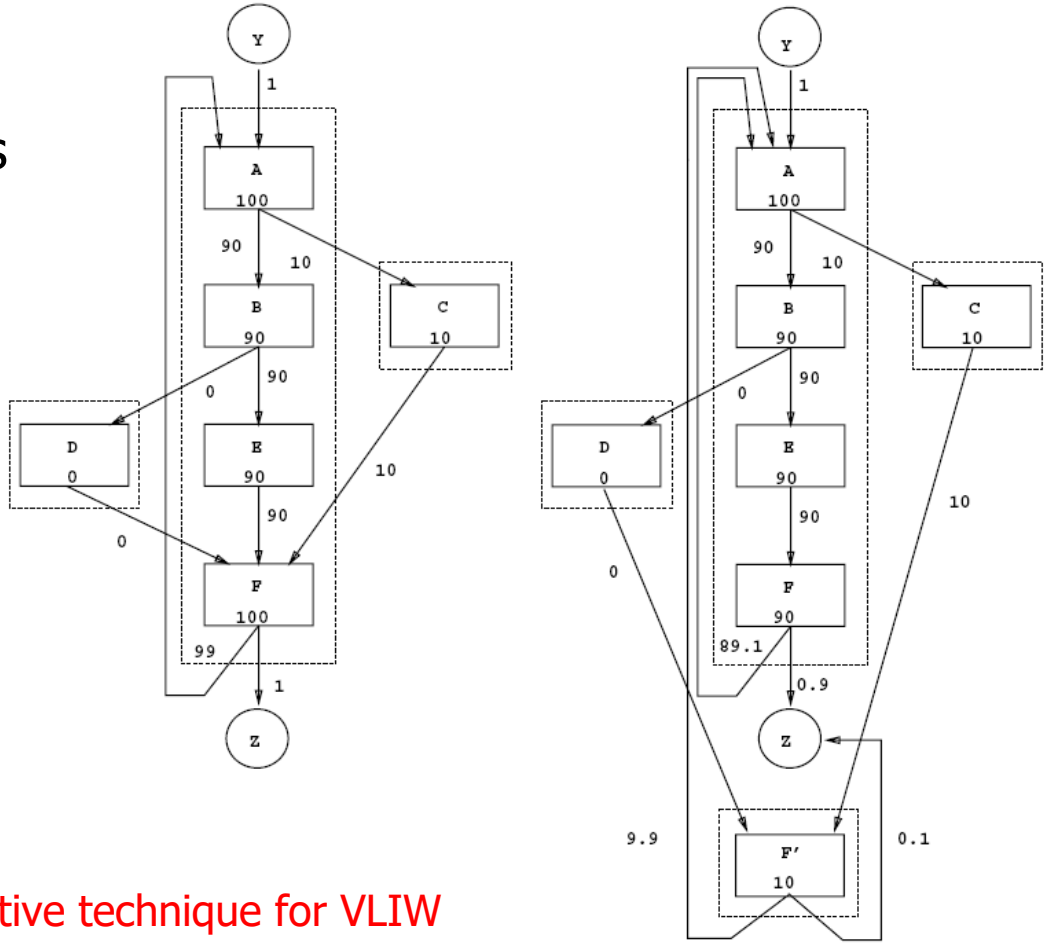
Pettis and Hansen, "Profile Guided Code Positioning," PLDI 1990.

Superblock: Can We Do Better?

- 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

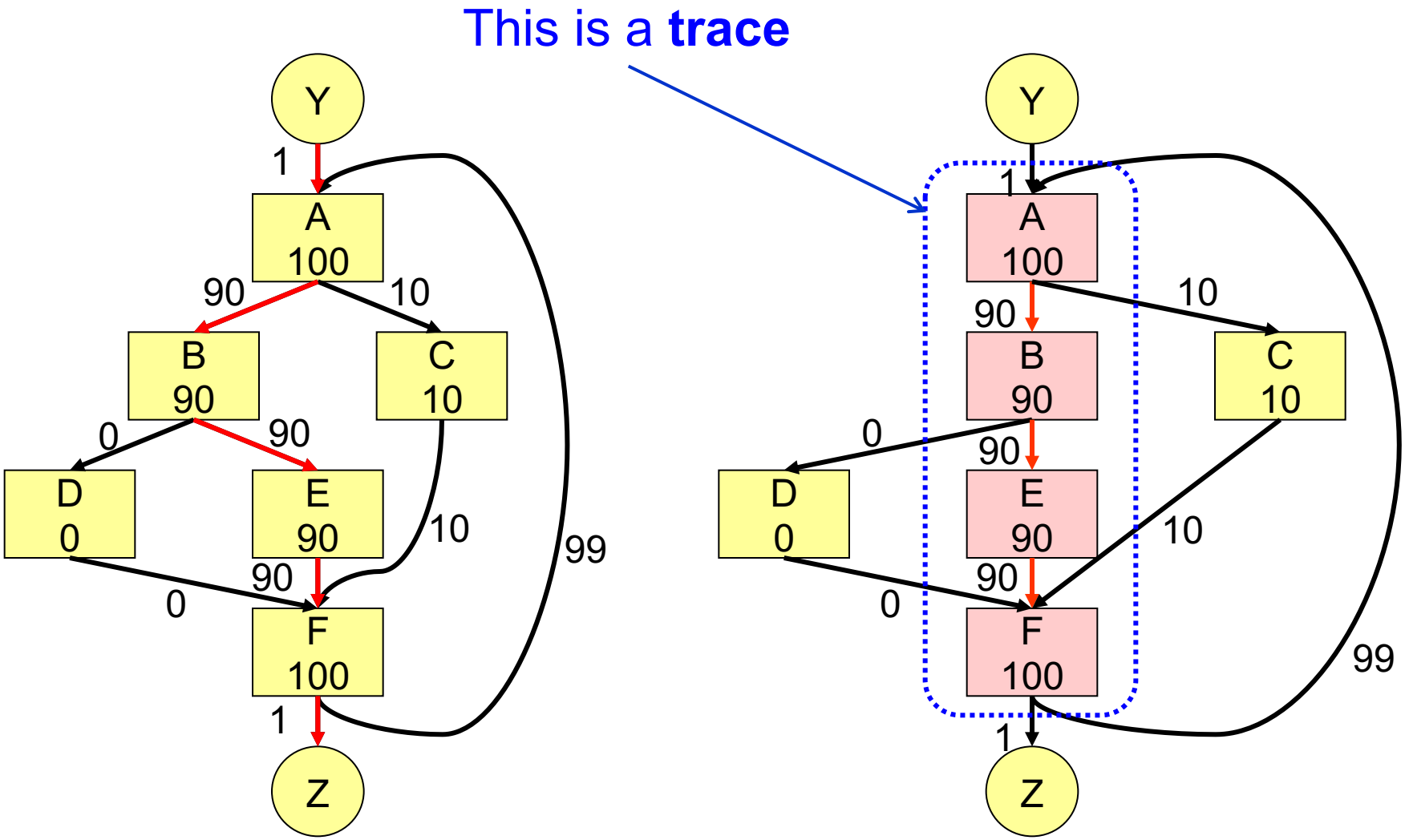
+ Reduces branch mispredictions
+ Enables aggressive
  compiler optimizations
  and code reordering
  within the superblock

-- Increased code size
-- Requires recompilation
-- Profile dependent

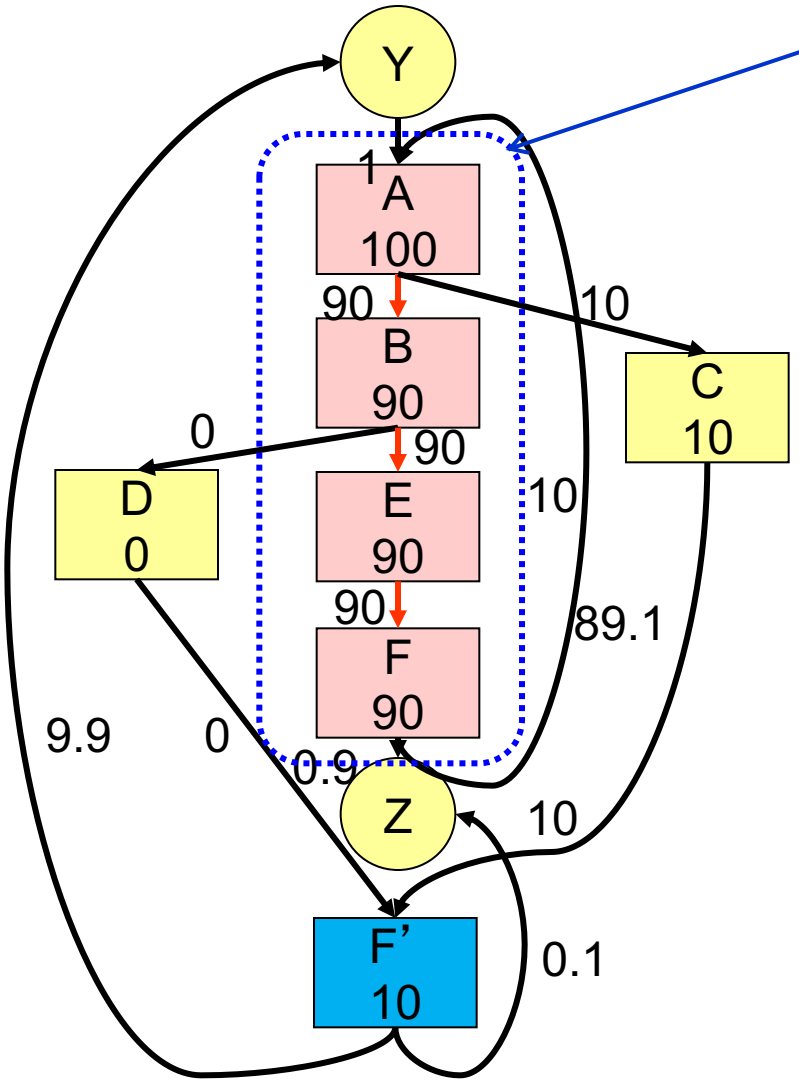


- Hwu et al. "The Superblock: An effective technique for VLIW and superscalar compilation," Journal of Supercomputing, 1993.

Superblock Formation (I)



Superblock Formation (II)



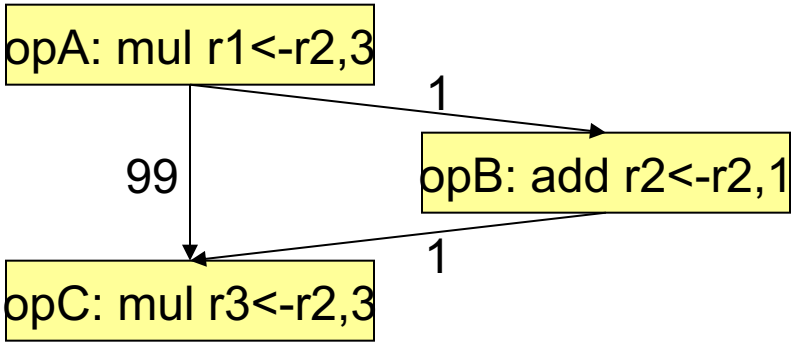
This is a **superblock**

(code with single entry point, multiple exit points)

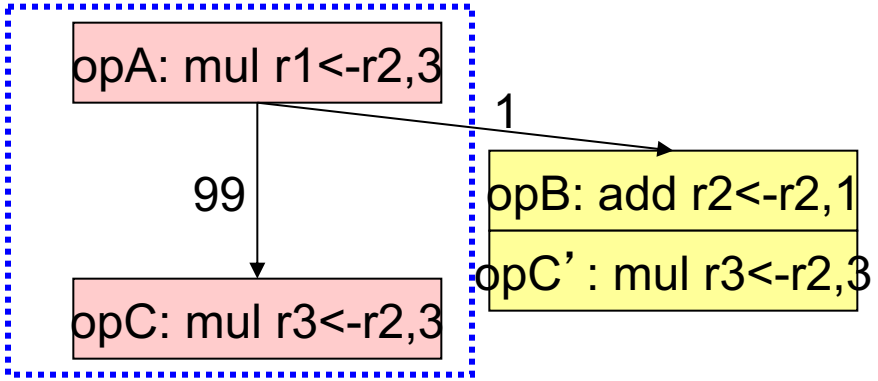
Tail duplication:
duplication of basic blocks after a side entrance to eliminate side entrances

→ transforms a trace into a superblock

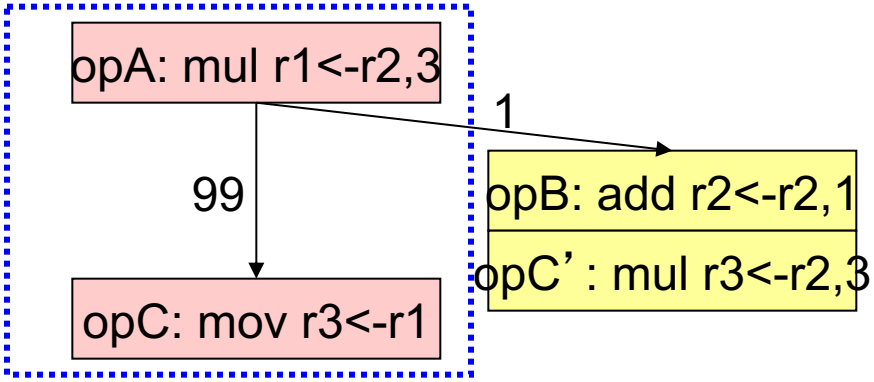
Superblock Code Optimization Example



Original Code



Code After Superblock Formation



Code After Common
Subexpression Elimination

Paper on Superblock Formation

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.

Chang et al., IMPACT: an architectural framework for multiple-instruction-issue processors. ISCA 1991.

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

The Bulldog VLIW Compiler

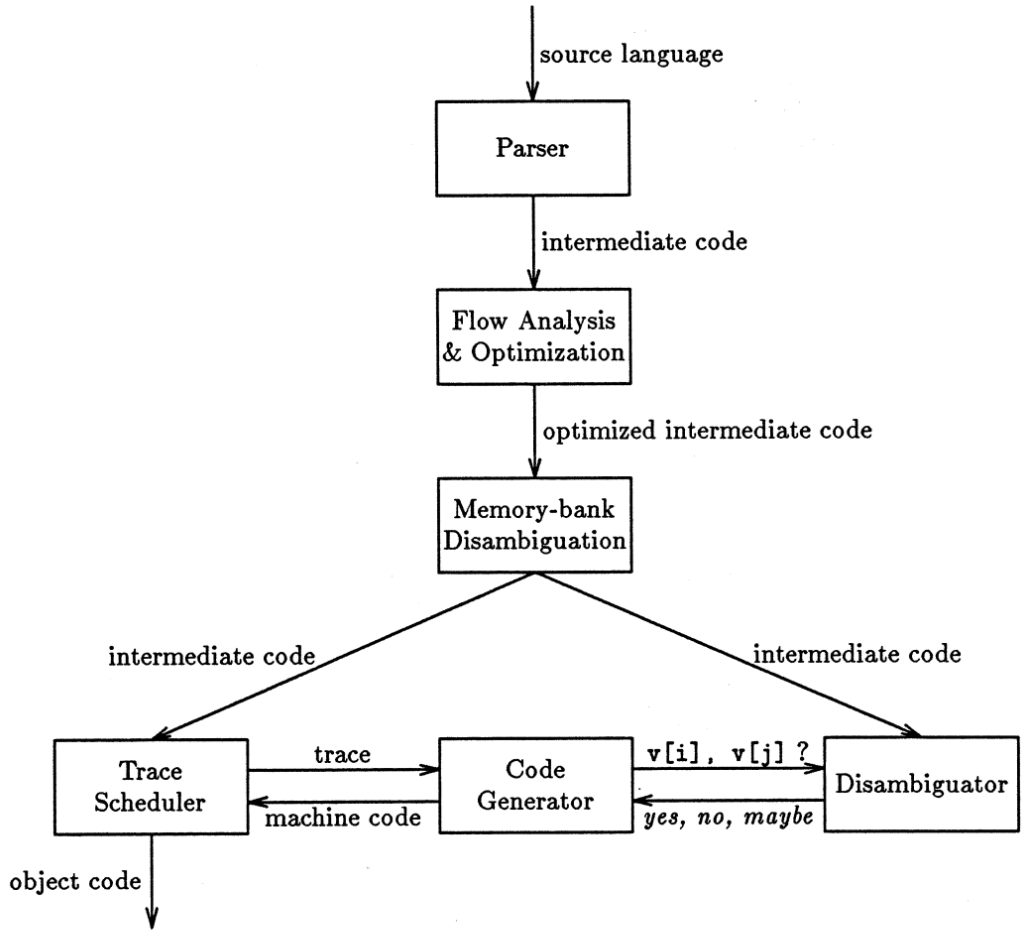
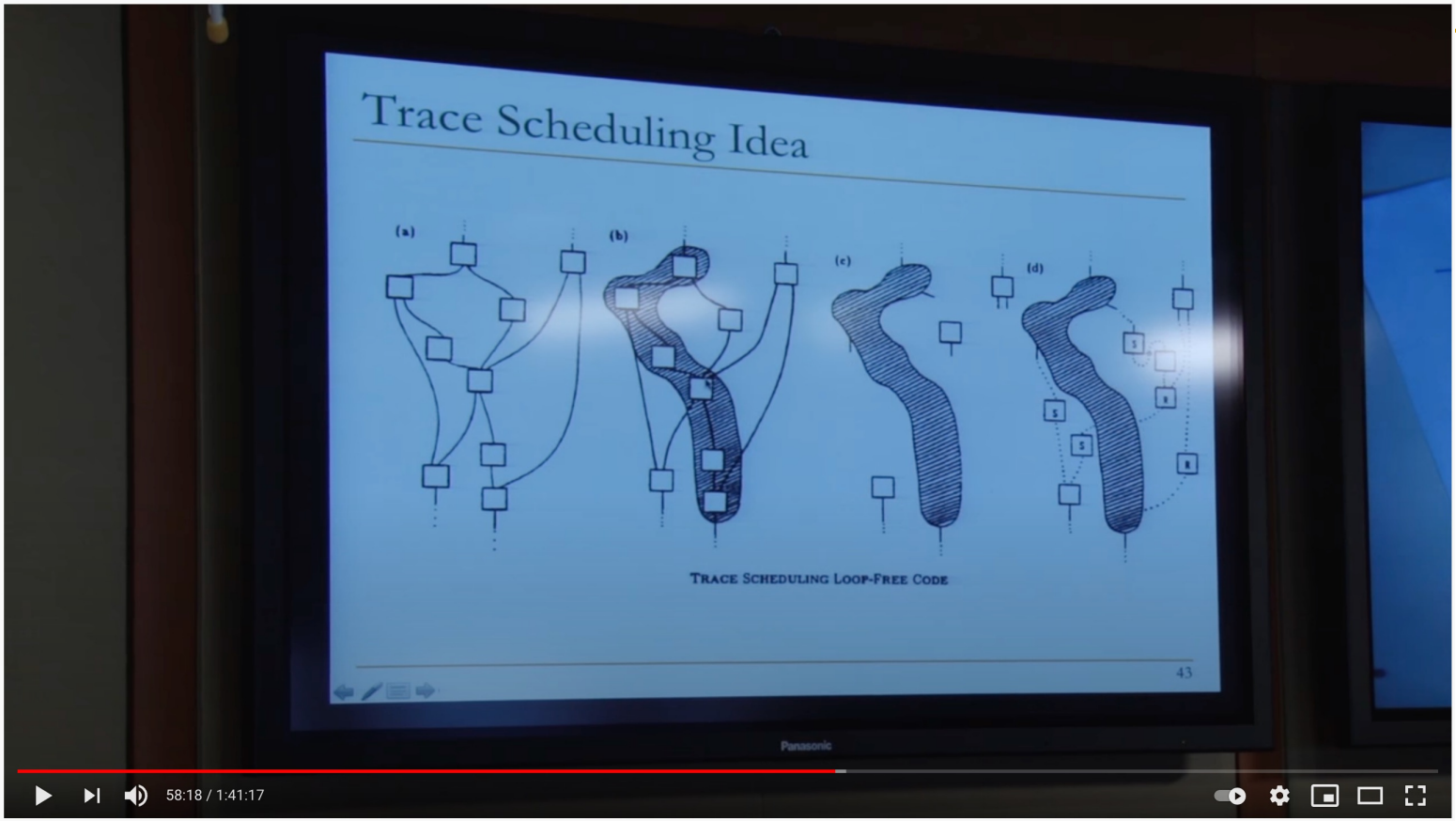


Figure 1.5. The Bulldog compiler.

John Ellis, "Bulldog: A Compiler for VLIW Architectures," PhD Thesis 1984.

Lecture on Static Instruction Scheduling



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

SUBSCRIBED

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...

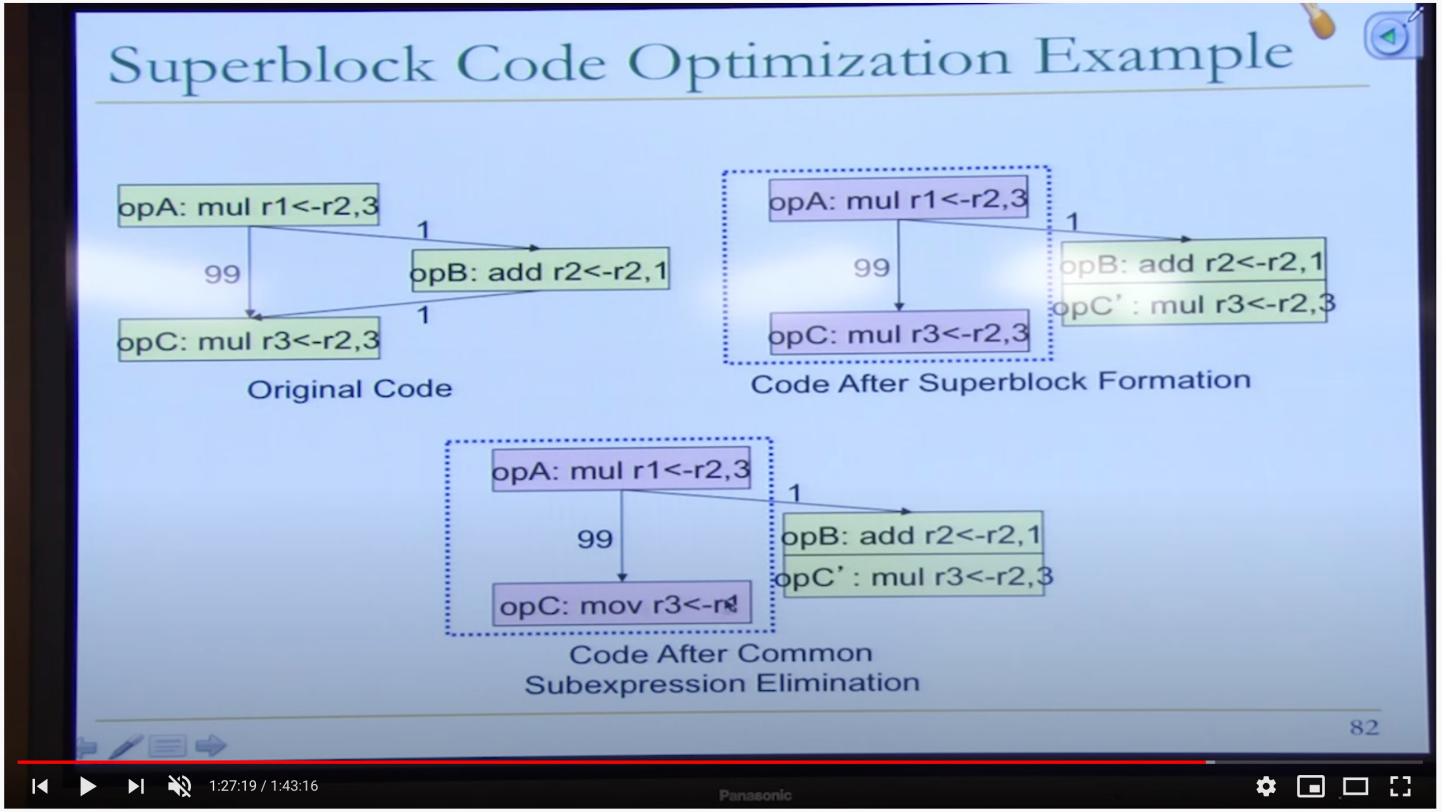
https://www.youtube.com/onurmutlulectures

Lectures on Static Instruction Scheduling

- Computer Architecture, Spring 2015, Lecture 16
  - Static Instruction Scheduling (CMU, Spring 2015)
  - https://www.youtube.com/watch?v=isBEVkIjgGA&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

SUBSCRIBED

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…

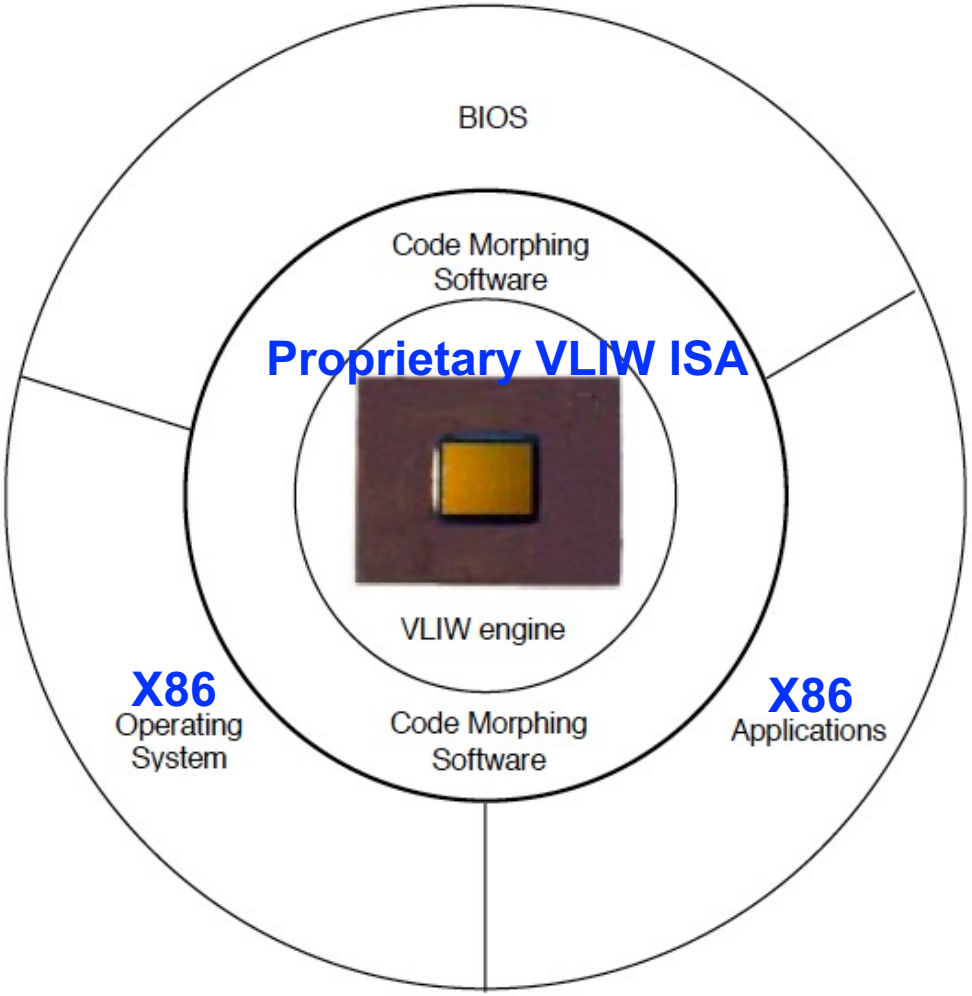
- Computer Architecture, Spring 2015, Lecture 5
  - Advanced Branch Prediction (CMU, Spring 2015)
  - https://www.youtube.com/watch?v=yDjsr-jTOtk&list=PL5PHm2jkkXmgVhh8CHAu9N76TShJqfYDt&index=4

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



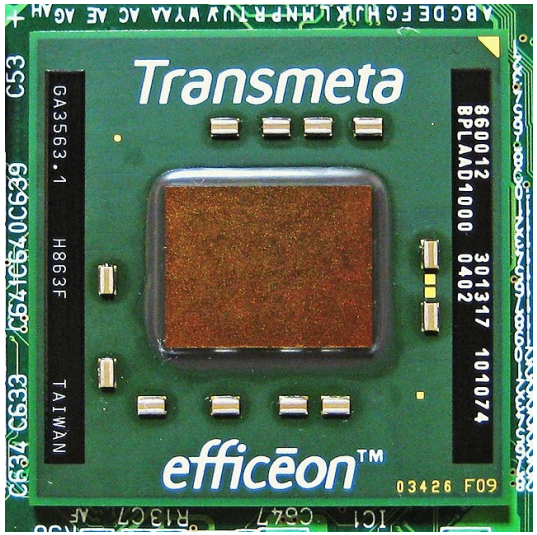


Figure 5. The Code Morphing software mediates between x86 software and the Crusoe processor.

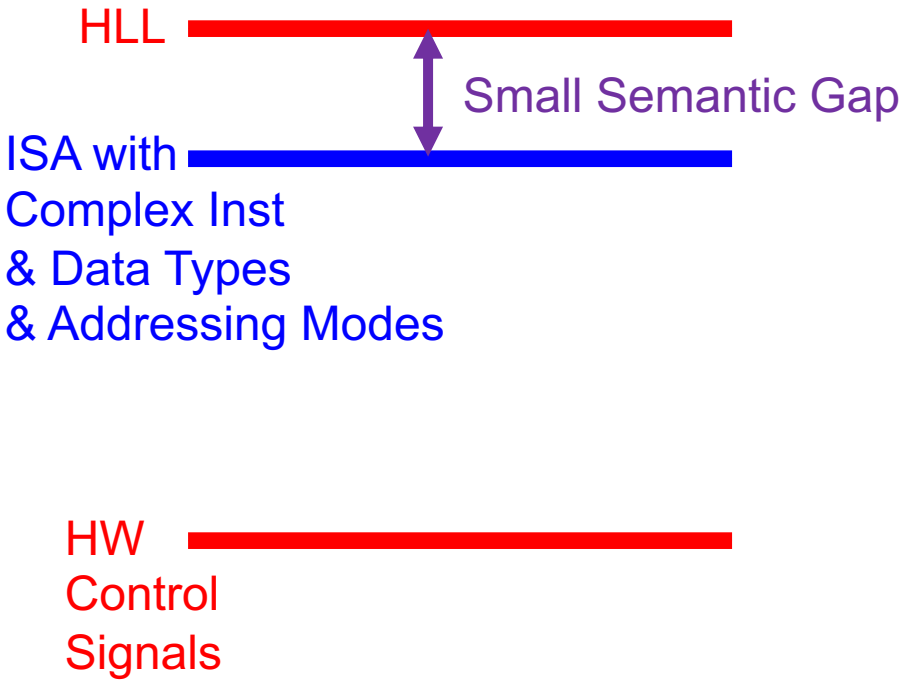
Klaiber, "The Technology Behind Crusoe Processors," Transmeta White Paper 2000.

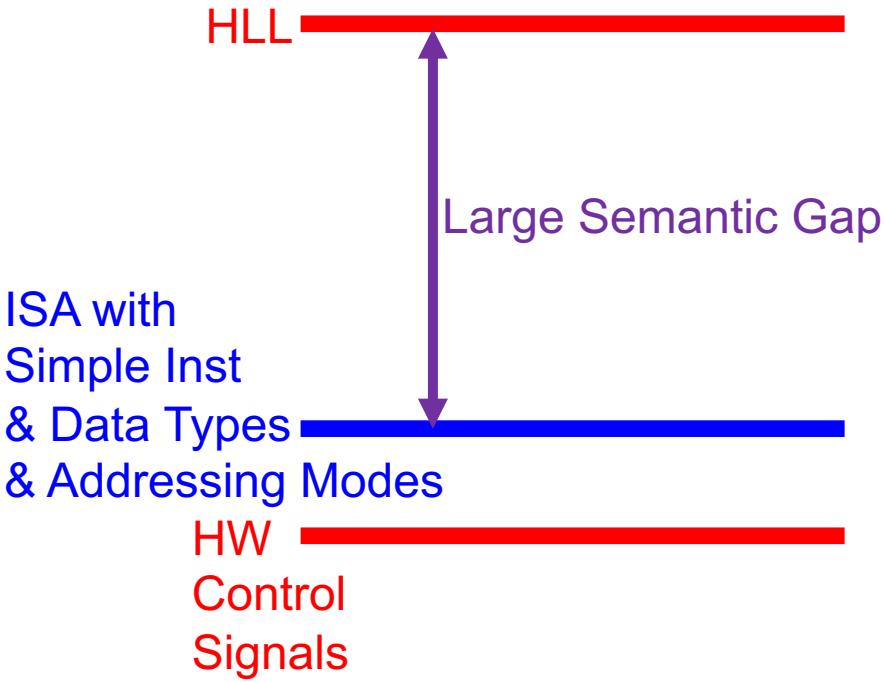
https://www.wikiwand.com/en/Transmeta_Efficeon

https://classes.engineering.wustl.edu/cse362/images/c/c7/Paper_aklaiber_19jan00.pdf

Recall: Semantic Gap

- How close instructions & data types & addressing modes are to high-level language (HLL)



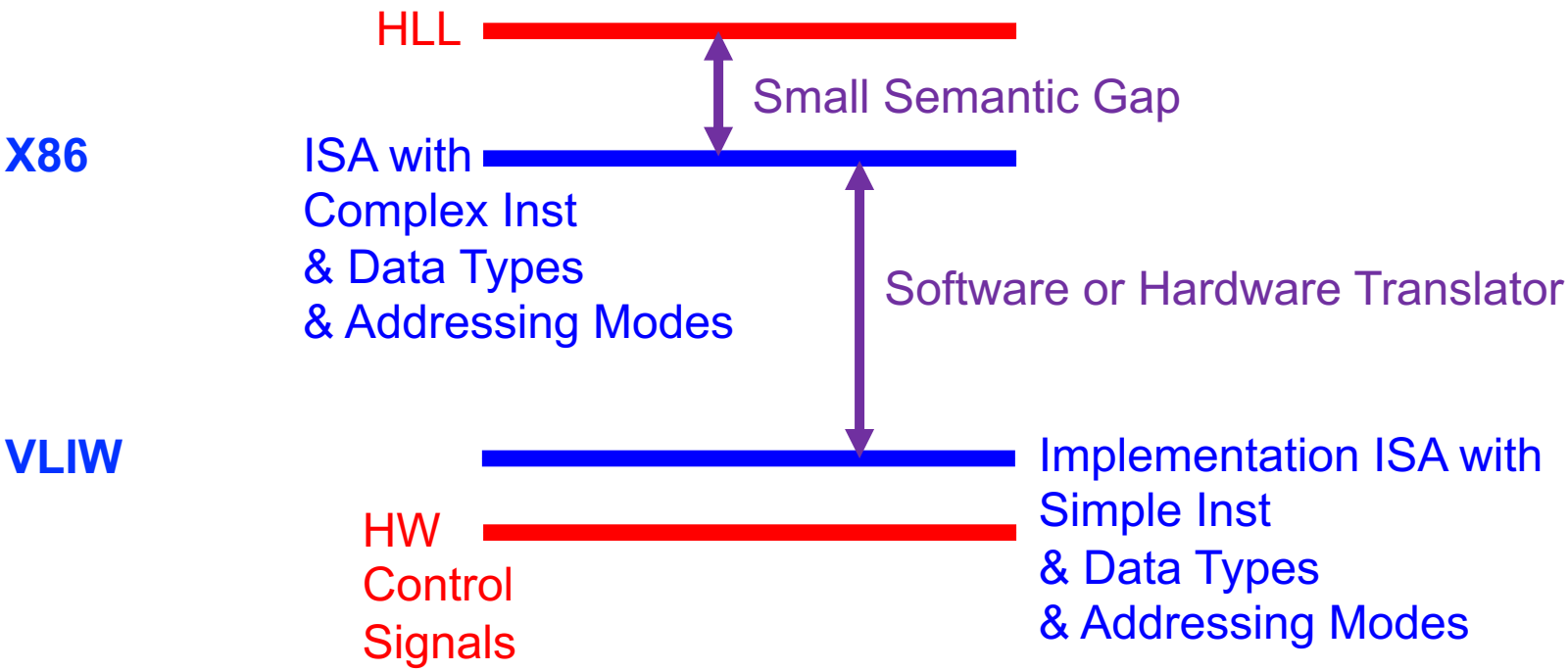


Easier mapping of HLL to ISA
Less work for software designer
More work for hardware designer
Optimization burden on HW

Harder mapping of HLL to ISA
More work for software designer
Less work for hardware designer
Optimization burden on SW

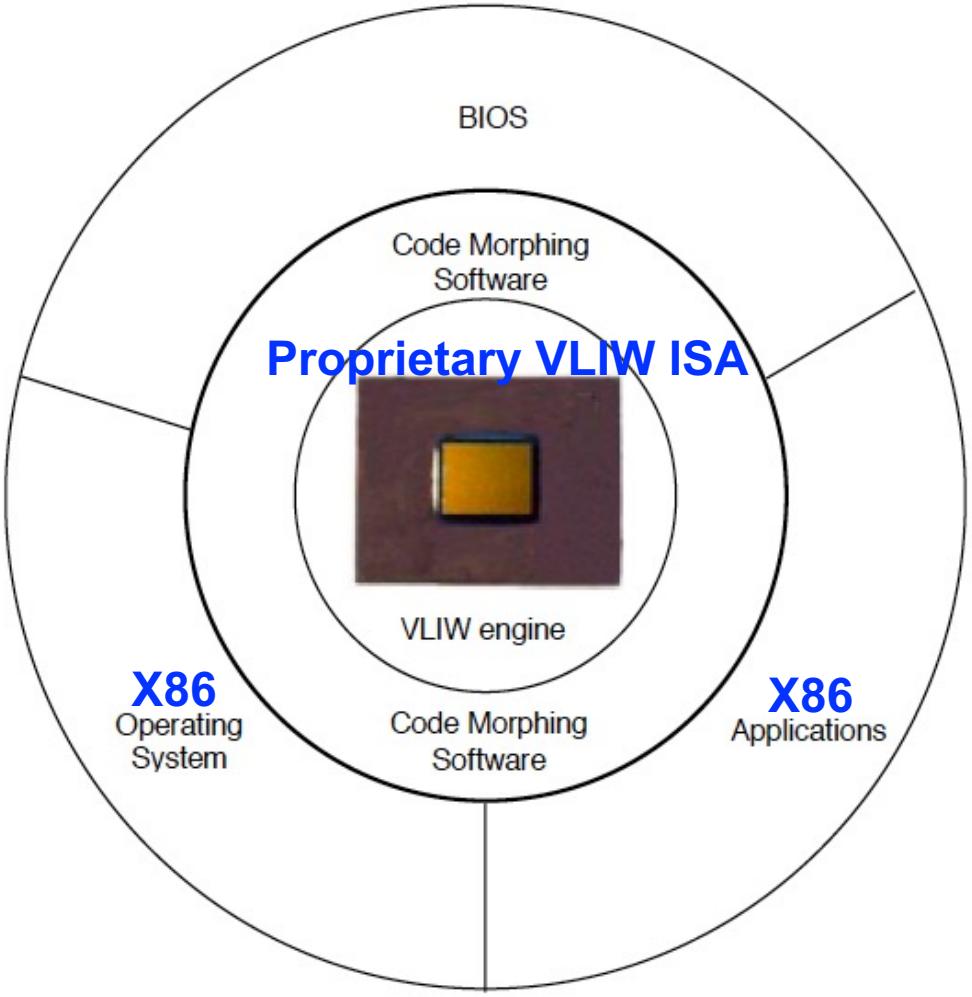
Recall: How to Change the Semantic Gap Tradeoffs

- Translate from one ISA into a different "implementation" ISA



SW, translator, and HW can all perform operation re-ordering

Transmeta: x86 to VLIW Translation



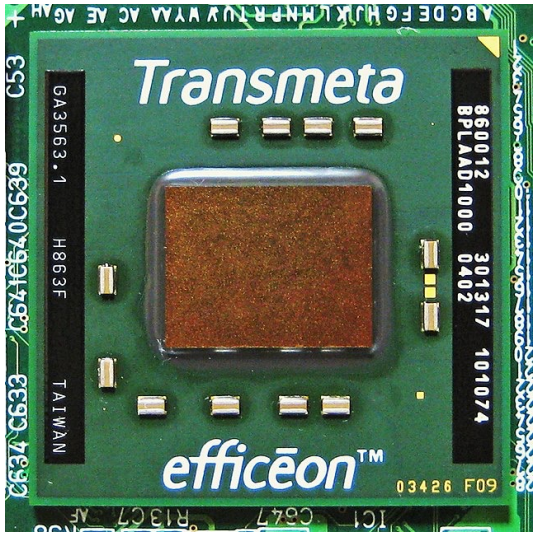


Figure 5. The Code Morphing software mediates between x86 software and the Crusoe processor.

Klaiber, "The Technology Behind Crusoe Processors," Transmeta White Paper 2000.

https://www.wikiwand.com/en/Transmeta_Efficeon

https://classes.engineering.wustl.edu/cse362/images/c/c7/Paper_aklaiber_19jan00.pdf

Another Example: Rosetta 2 Binary Translator

Rosetta 2 [edit]

In 2020, Apple announced Rosetta 2 would be bundled with macOS Big Sur, to aid in the Mac transition to Apple silicon. The software permits many applications compiled exclusively for execution on x86-64-based processors to be translated for execution on Apple silicon.[2][8]

In addition to the just-in-time (JIT) translation support, Rosetta 2 offers ahead-of-time compilation (AOT), with the x86-64 code fully translated, just once, when an application without a universal binary is installed on an Apple silicon Mac.[9]

Rosetta 2's performance has been praised greatly.[10][11] In some benchmarks, x86-64-only programs performed better under Rosetta 2 on a Mac with an Apple M1 SOC than natively on a Mac with an Intel x86-64 processor. One of the key reasons why Rosetta 2 provides such high level of translation efficiency is the support of x86-64 memory ordering in Apple M1 SOC.[12]

Mac transition to Apple silicon

Apple silicon · ARM architecture · Universal 2 binary · Rosetta 2 · Developer Transition Kit

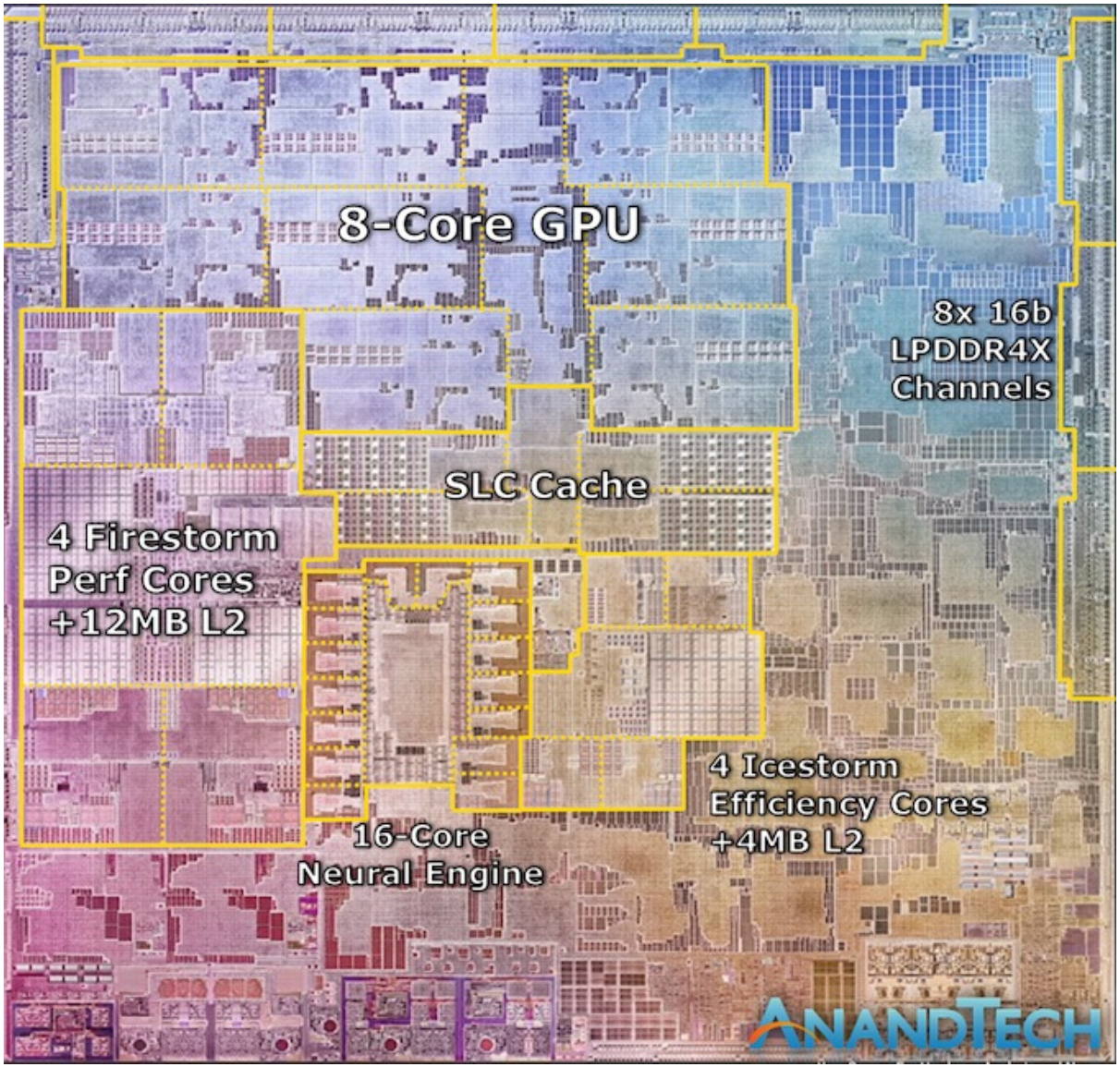
V · T · E

Although Rosetta 2 works for most software, some software doesn't work at all[13] or is reported to be "sluggish".[14] A lot of software can be made compatible with the new Macs by the vendor recompiling the software, often a simple task; while for some software (such as software that includes assembly language code, or that generates machine code), the changes to make them work aren't simple and cannot be automated.

Similar to the first version, Rosetta 2 does not normally require user intervention. When a user attempts to launch an x86-64-only application for the first time, macOS prompts them to install Rosetta 2 if it is not already available. Subsequent launches of x86-64 programs will execute via translation automatically. An option also exists to force a universal binary to run as x86-64 code through Rosetta 2, even on an ARM-based machine.[15]

https://en.wikipedia.org/wiki/Rosetta_(software)#Rosetta_2

Another Example: Rosetta 2 Binary Translator



Apple M1, 2021

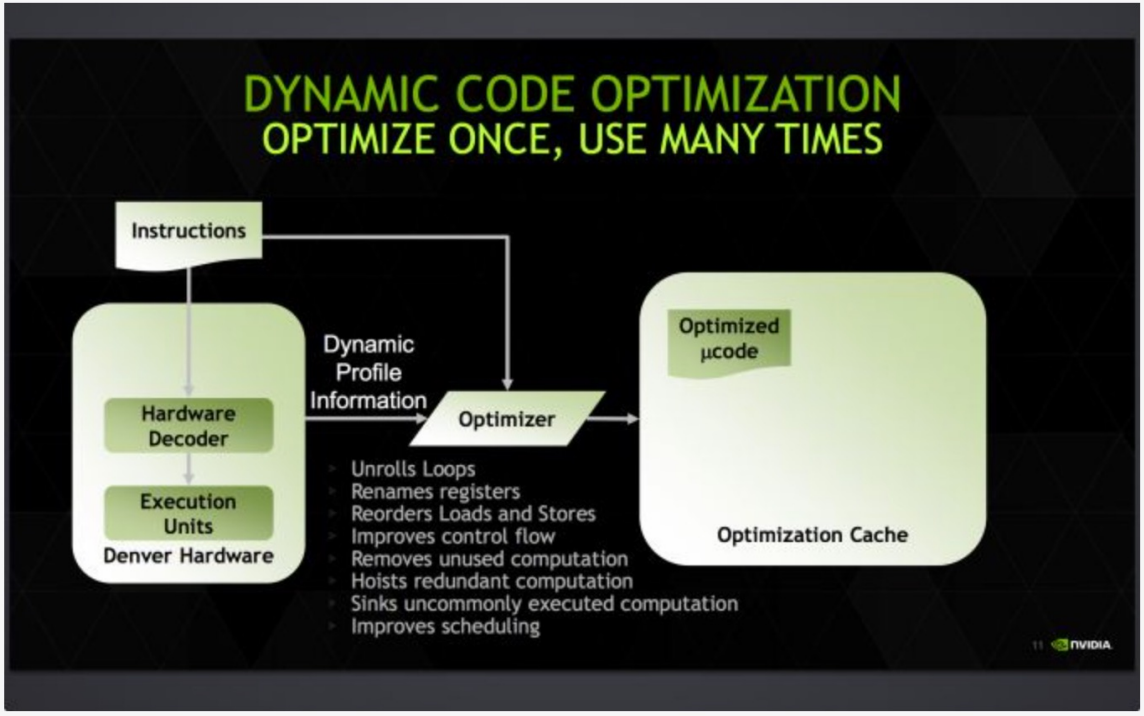
Source: https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested

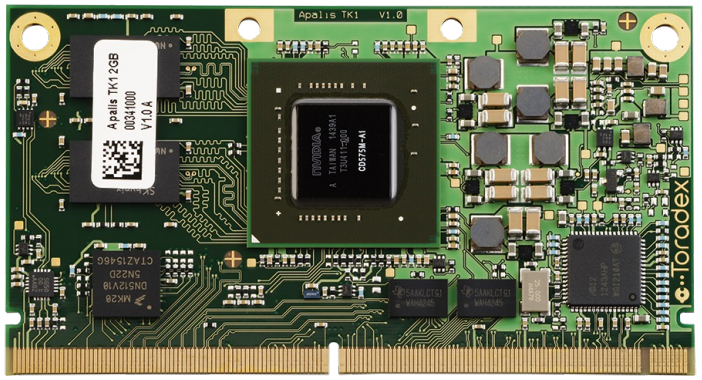
Another Example: NVIDIA Denver

The Secret of Denver: Binary Translation & Code Optimization

As we alluded to earlier, NVIDIA's decision to forgo a traditional out-of-order design for Denver means that much of Denver's potential is contained in its software rather than its hardware. The underlying chip itself, though by no means simple, is at its core a very large in-order processor. So it falls to the software stack to make Denver sing.

Accomplishing this task is NVIDIA's dynamic code optimizer (DCO). The purpose of the DCO is to accomplish two tasks: to translate ARM code to Denver's native format, and to optimize this code to make it run better on Denver. With no out-of-order hardware on Denver, it is the DCO's task to find instruction level parallelism within a thread to fill Denver's many execution units, and to reorder instructions around potential stalls, something that is no simple task.





The DCO system employed in the Denver CPU is codesigned software that extends ideas from prior system-level binary translators.[2] The primary function is to execute the user's code. The secondary function is to profile execution, create, optimize, and manage regions of tens to thousands of ARM instructions to form equivalent microcode-optimized regions that execute efficiently on the underlying microarchitecture.

https://www.anandtech.com/show/8701/the-google-nexus-9-review/4 https://www.toradex.com/computer-on-modules/apalis-arm-family/nvidia-tegra-k1
https://safari.ethz.ch/digitaltechnik/spring2021/lib/exe/fetch.php?media=boggs_ieeemicro_2015.pdf

More on NVIDIA Denver Code Optimizer

DENVER: NVIDIA'S FIRST 64-BIT ARM PROCESSOR

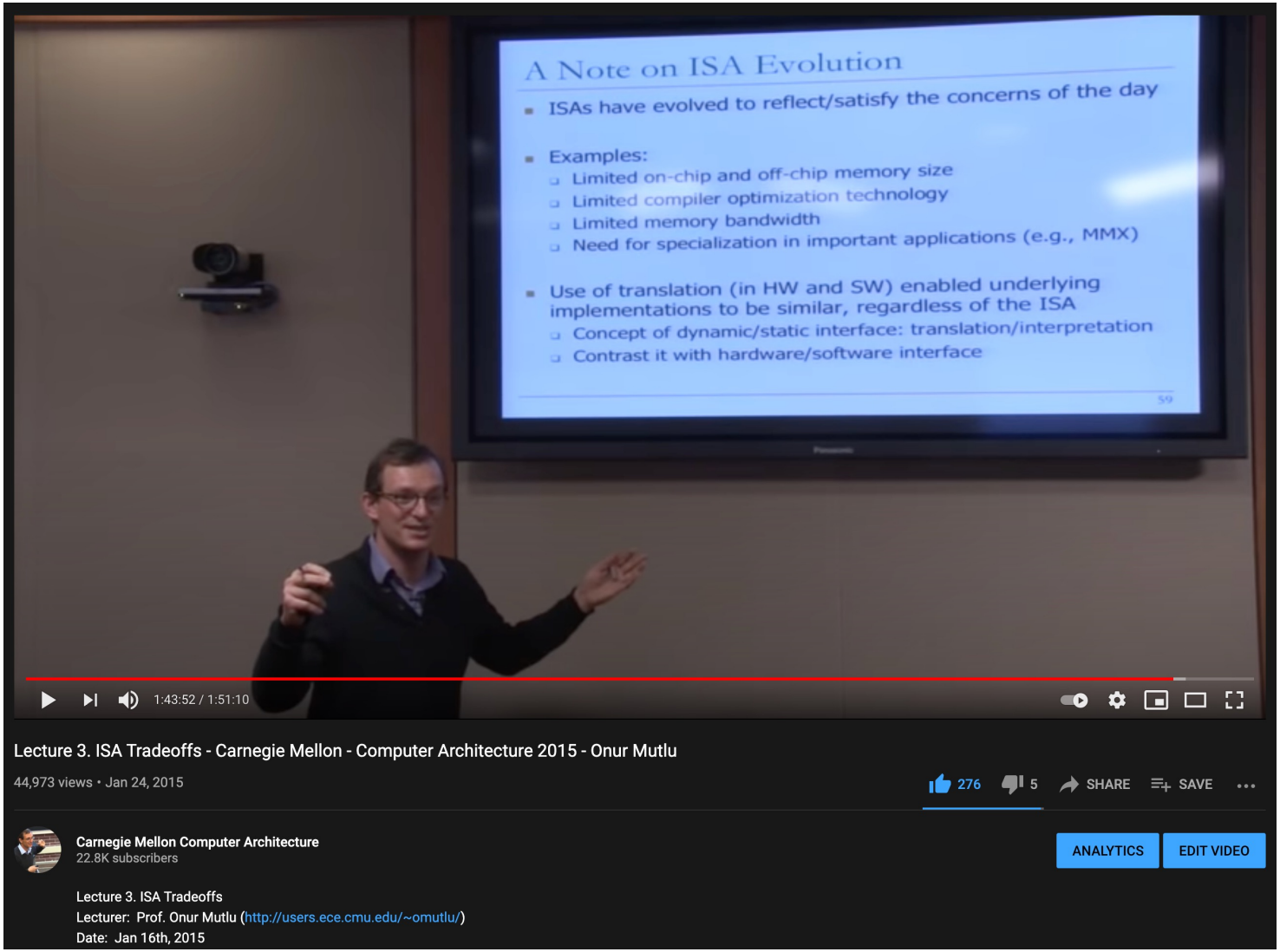
NVIDIA'S FIRST 64-BIT ARM PROCESSOR, CODE-NAMED DENVER, LEVERAGES A HOST OF NEW TECHNOLOGIES, SUCH AS DYNAMIC CODE OPTIMIZATION, TO ENABLE HIGH-PERFORMANCE MOBILE COMPUTING. IMPLEMENTED IN A 28-NM PROCESS, THE DENVER CPU CAN ATTAIN CLOCK SPEEDS OF UP TO 2.5 GHZ. THIS ARTICLE OUTLINES THE DENVER ARCHITECTURE, DESCRIBES ITS TECHNOLOGICAL INNOVATIONS, AND PROVIDES RELEVANT COMPARISONS AGAINST COMPETING MOBILE PROCESSORS.

Codesigning a hardware processor with a DCO software system creates both additional validation exposure and benefits. The DCO system can be upgraded in the field to address functionality, performance, or security issues.

The Denver hardware decoder provides a mechanism for periodically profiling recently taken branches. This branch history is moved into a shared buffer that can be processed from other cores, thereby minimizing the latency of the interruption. The DCO system will then run a thread that uses this profile to evaluate the dynamic properties of code executing and to assemble a picture of which code regions are hottest across all the processors. On finding sufficiently hot code, the DCO system will begin an optimization process to turn this input ARM code into a microcode execution region. The optimization process uses well-known traditional[3] and more speculative compiler techniques to reduce work and increase efficiency of execution on the underlying skewed pipeline. To keep the latency of interruptions to a minimum, the optimizer thread is time-sliced with ARM execution (if any) and runs in a mode that can be quickly interrupted.

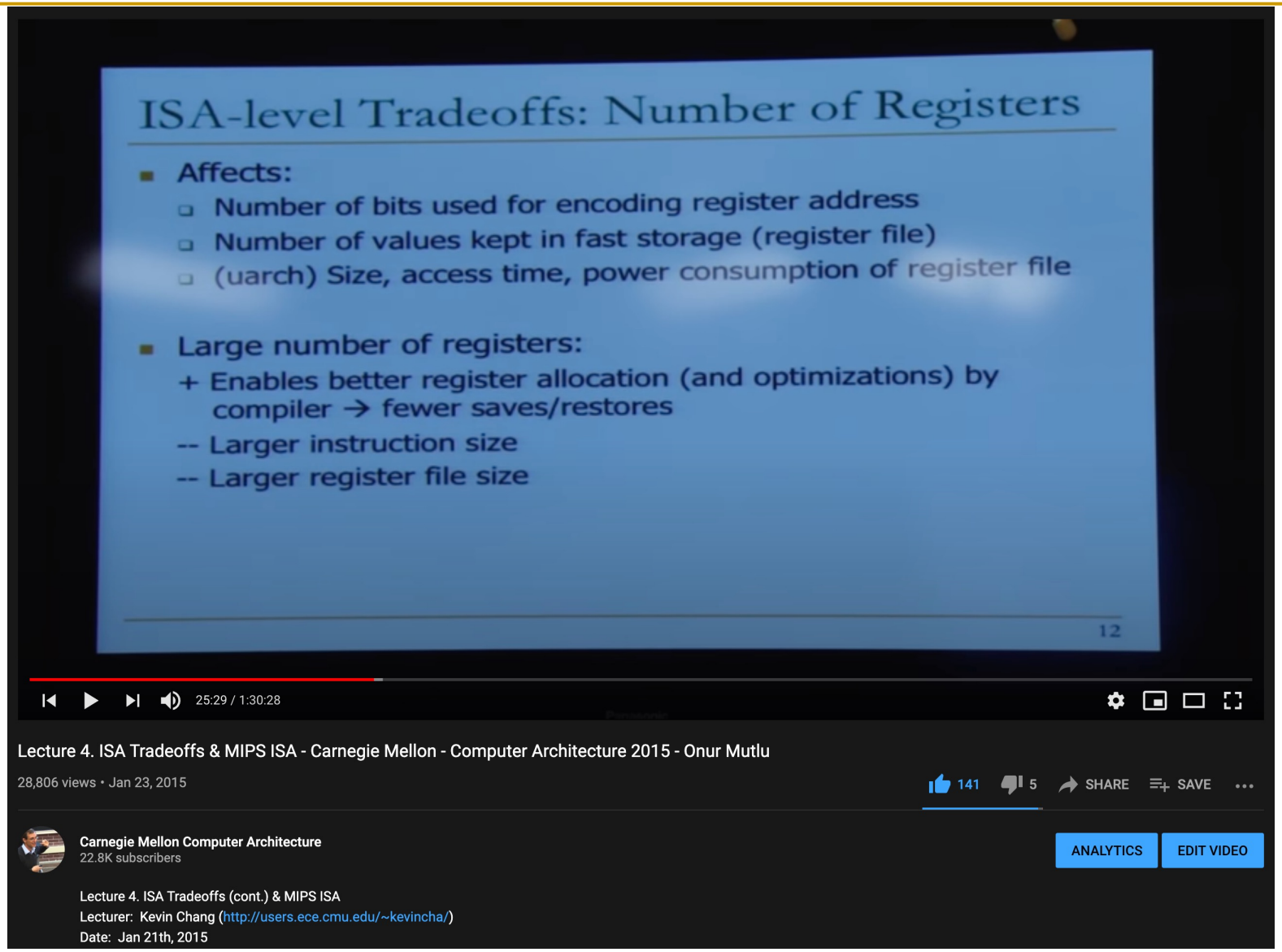
https://safari.ethz.ch/digitaltechnik/spring2021/lib/exe/fetch.php?media=boggs_ieeemicro_2015.pdf

There Is A Lot More to Cover on ISAs



https://www.youtube.com/onurmutlulectures

There Is A Lot More to Cover on ISAs



https://www.youtube.com/onurmutlulectures

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

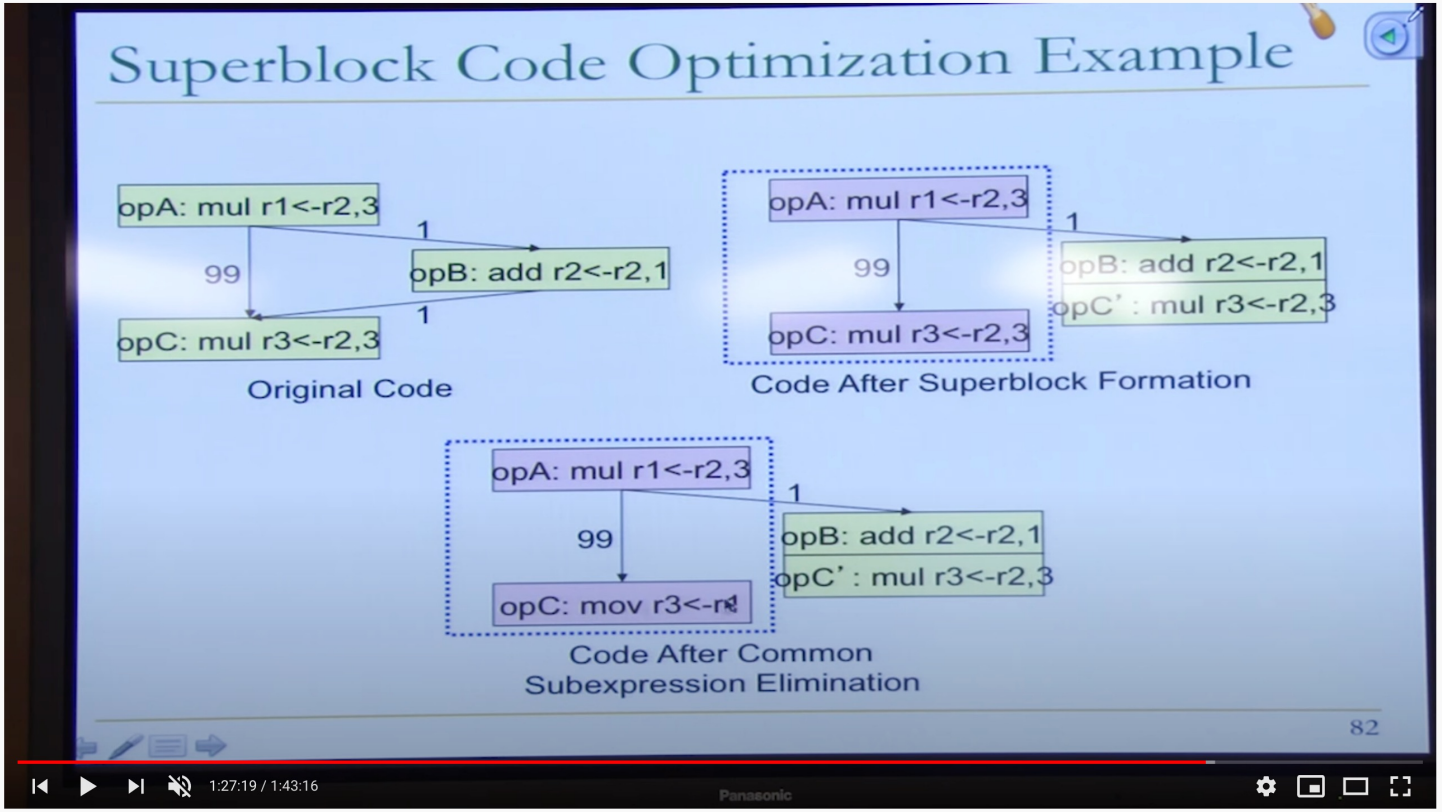
Computer Architecture
Lecture 27: VLIW Architectures

Prof. Onur Mutlu
ETH Zürich
Fall 2023
31 January 2024

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

SUBSCRIBED

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

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

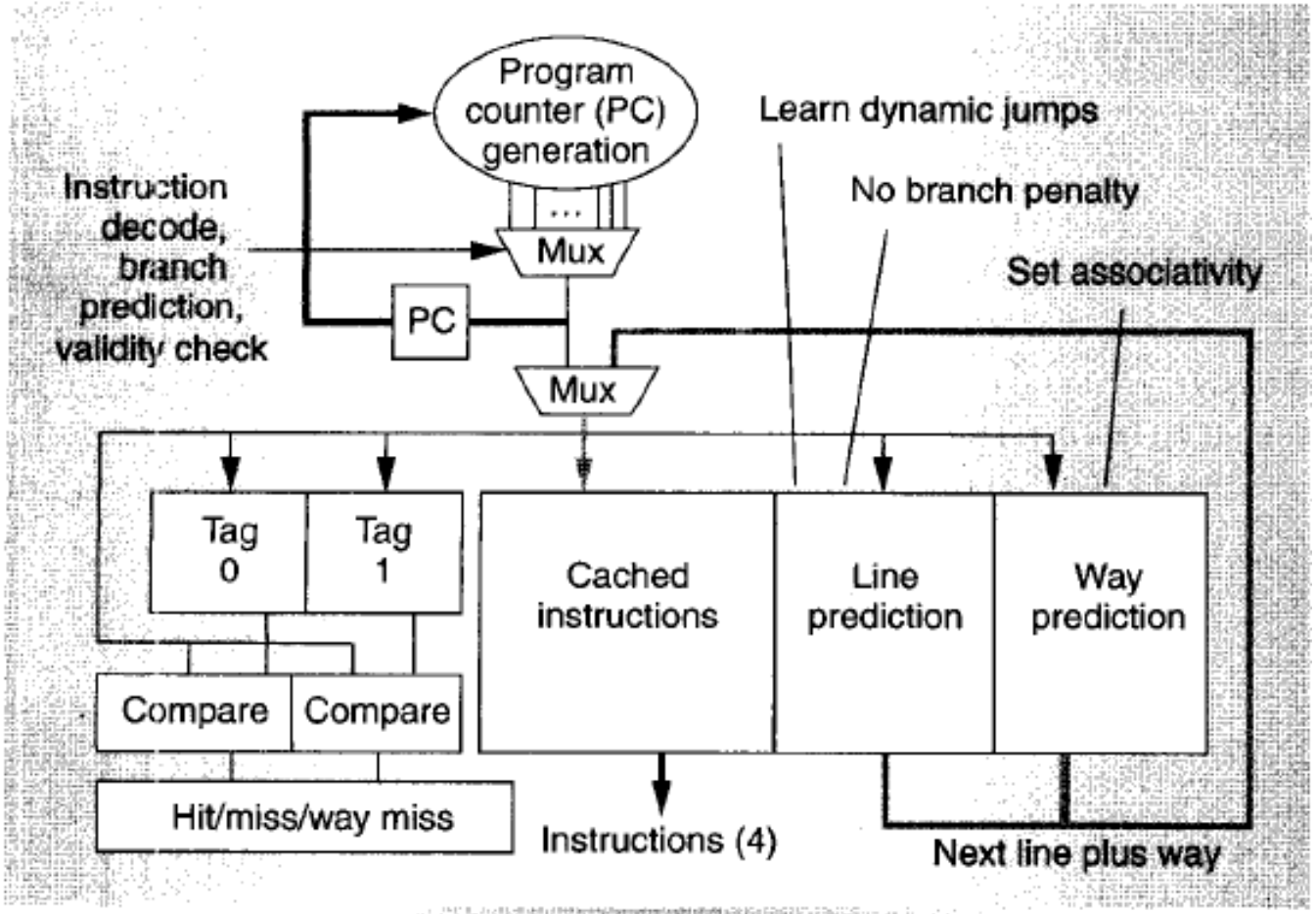
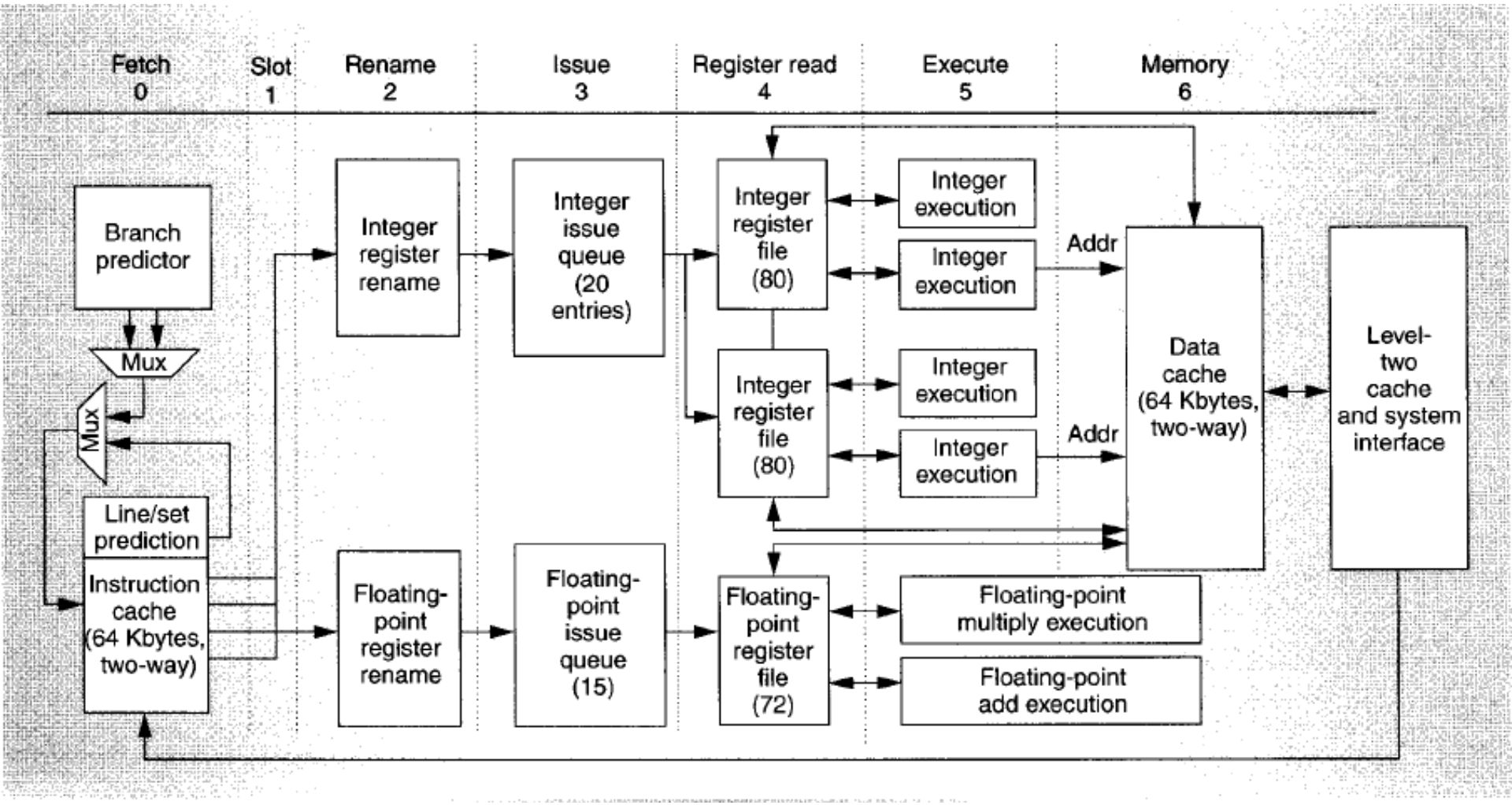


Figure 3. Alpha 21264 instruction fetch. The line and way prediction (wrap-around path on the right side) provides a fast instruction fetch path that avoids common fetch stalls when the predictions are correct.

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

Alpha 21264 Line & Way Prediction



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

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 is 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

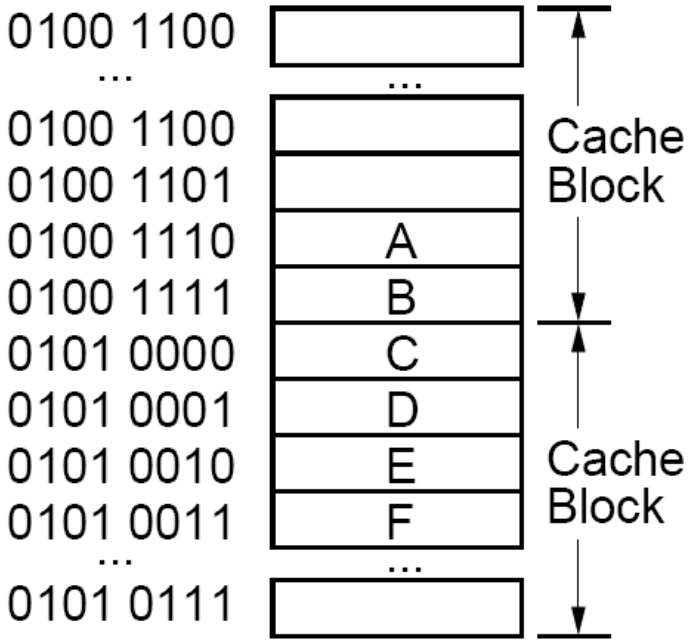
- Large cache blocks: Hope N instructions contained in the block

- 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

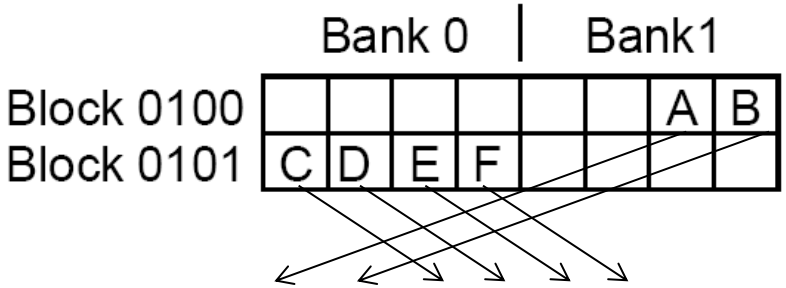
Split Line Fetch

Cache Banking

Memory Map

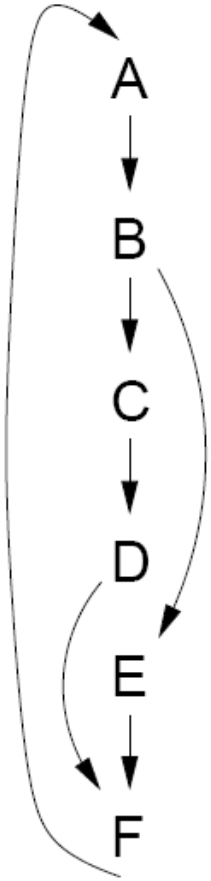


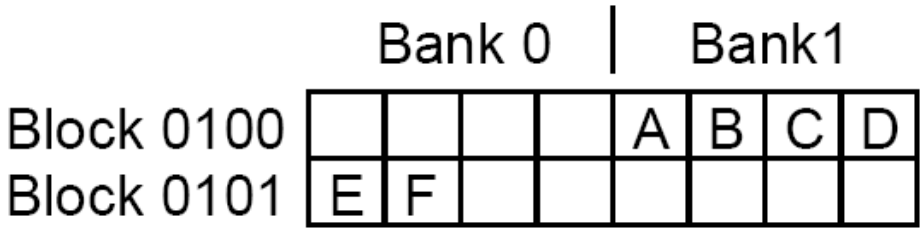
Cache



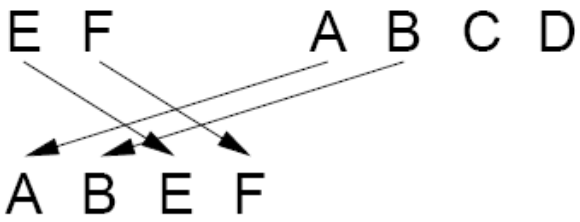
Need alignment logic:

Short Distance Predicted-Taken Branches

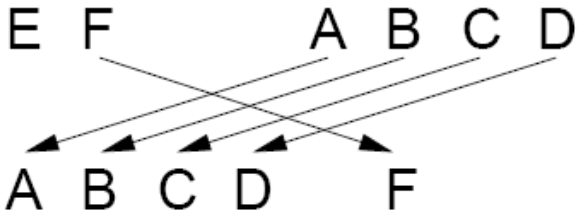




First Iteration (Branch B taken to E)



Second Iteration (Branch B fall through to C)

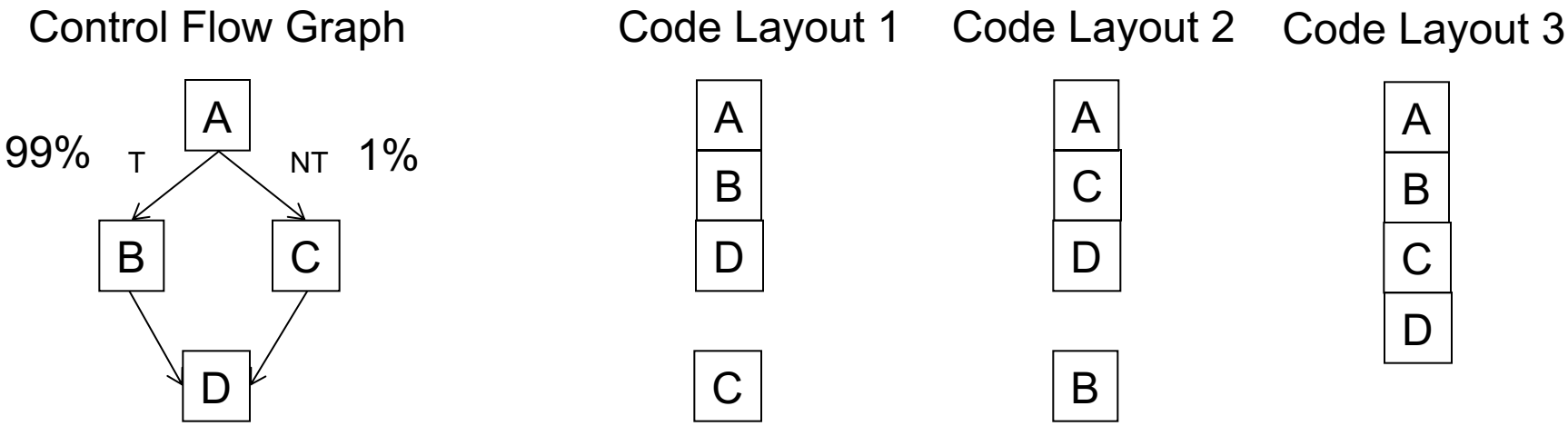


Techniques to Reduce Fetch Breaks

- 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



- 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

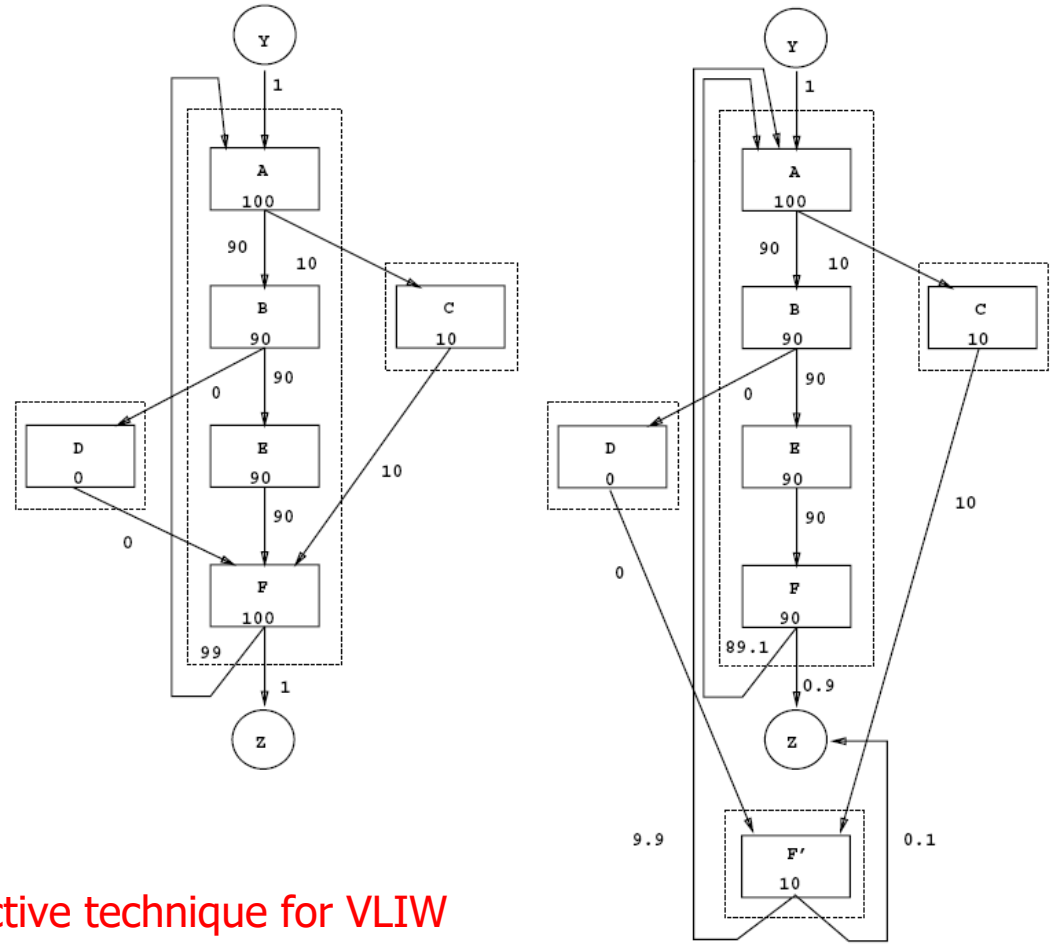
- Disadvantages:
  -- Dependent on compile-time profiling
  -- Does not help if branches are not biased
  -- Requires recompilation

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

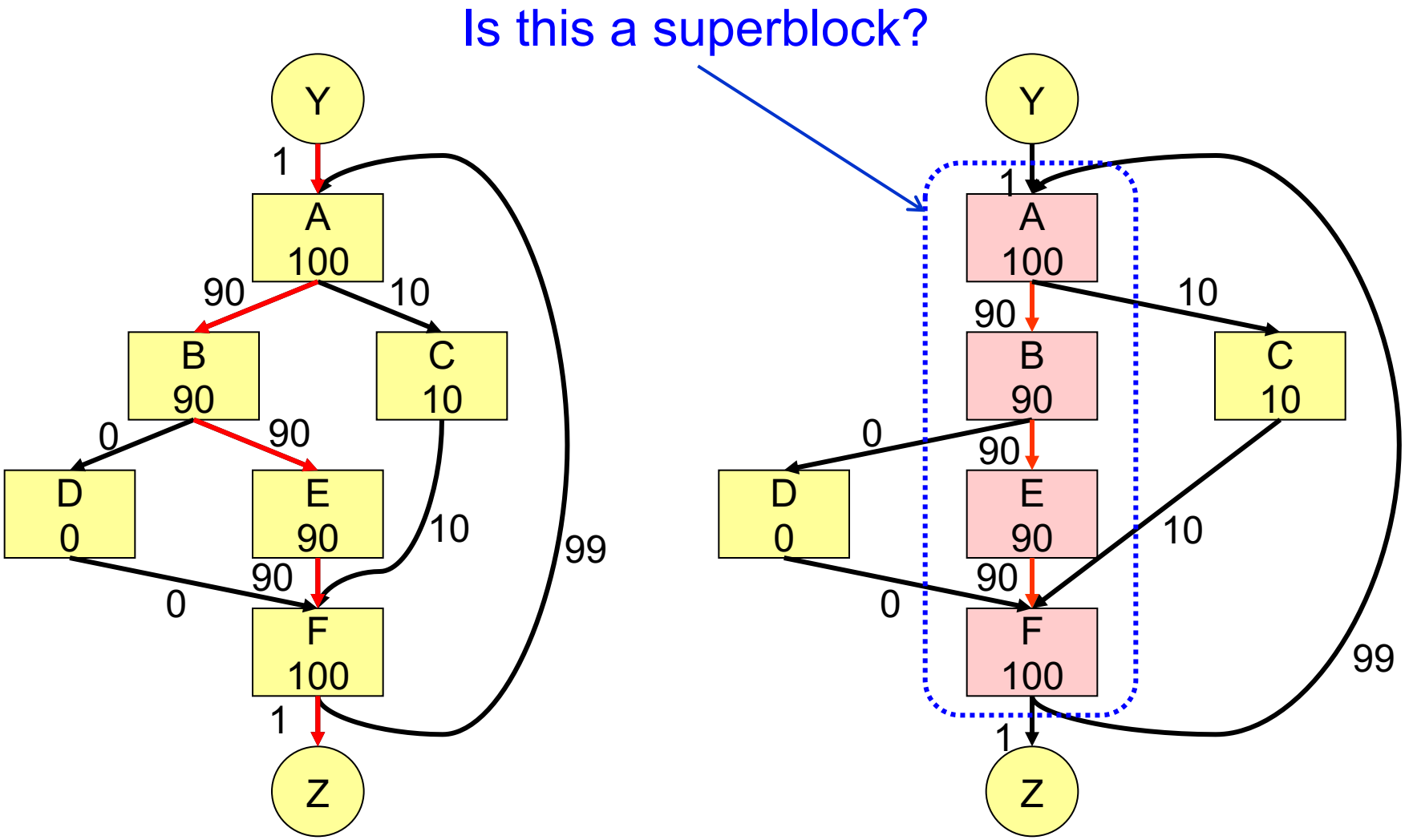
+ Helps wide fetch
+ Enables aggressive compiler optimizations and code reordering within the superblock

-- Increased code size
-- Profile dependent
-- Requires recompilation

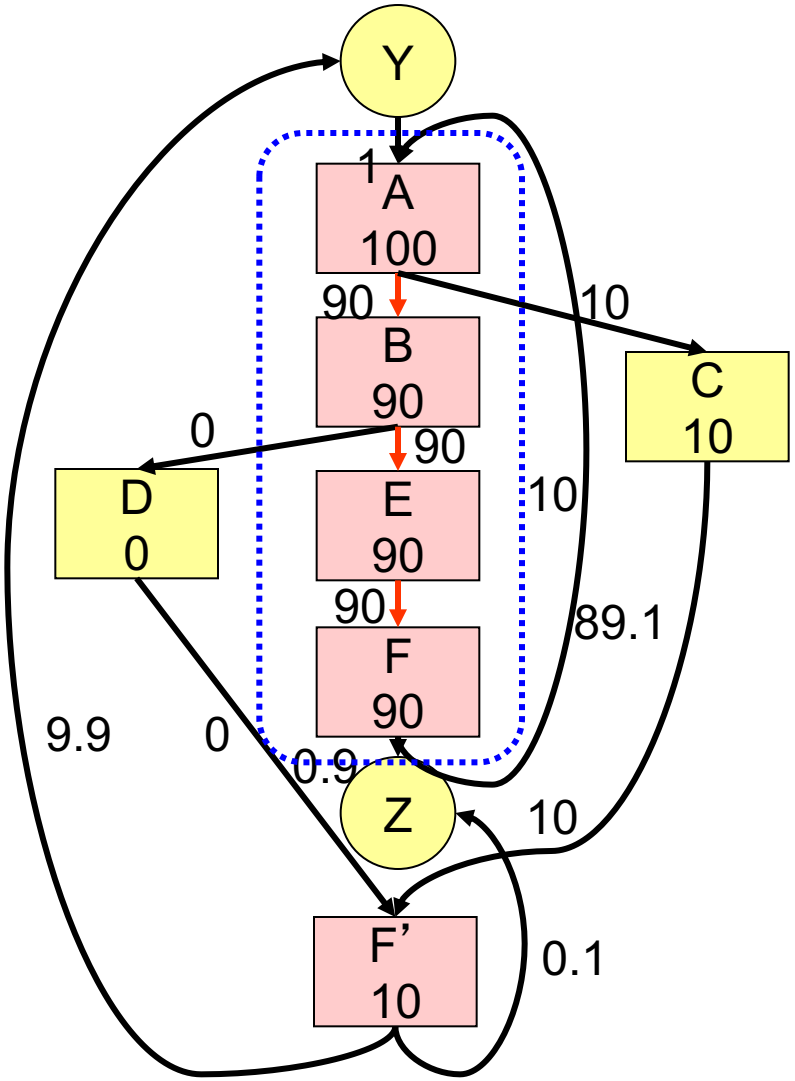


- Hwu et al. "The Superblock: An effective technique for VLIW and superscalar compilation," Journal of Supercomputing, 1993.

Superblock Formation (I)

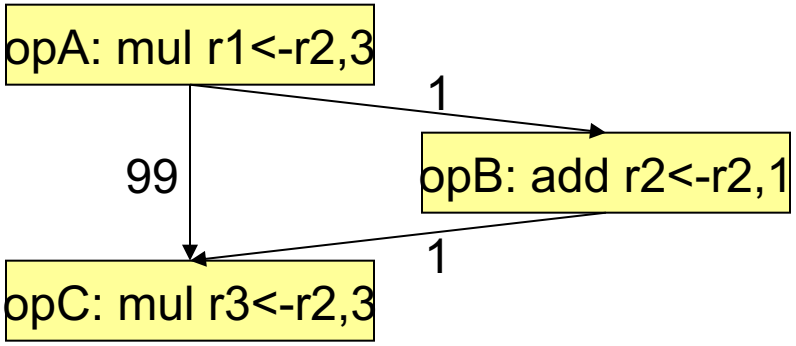


Superblock Formation (II)

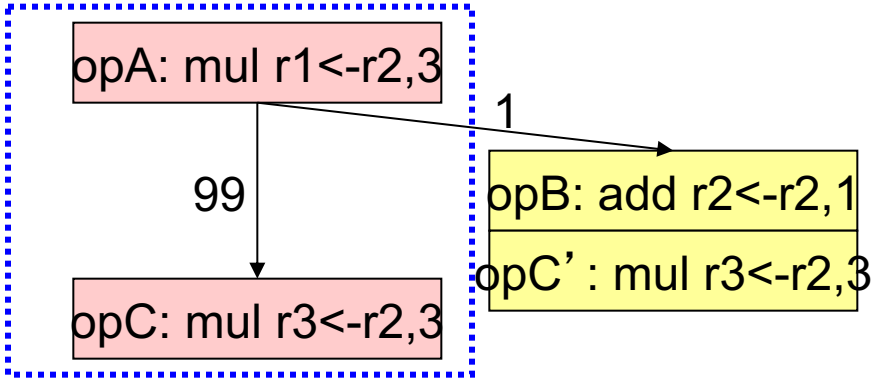


Tail duplication:
duplication of basic blocks after a side entrance to eliminate side entrances
→ transforms
a trace into a superblock.

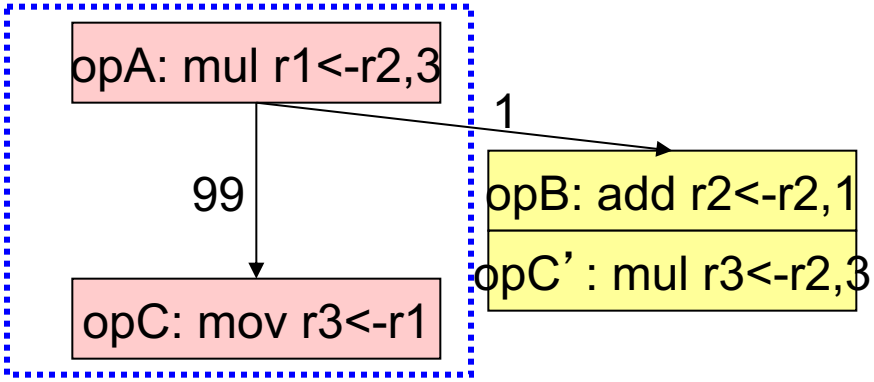
Superblock Code Optimization Example



Original Code



Code After Superblock Formation



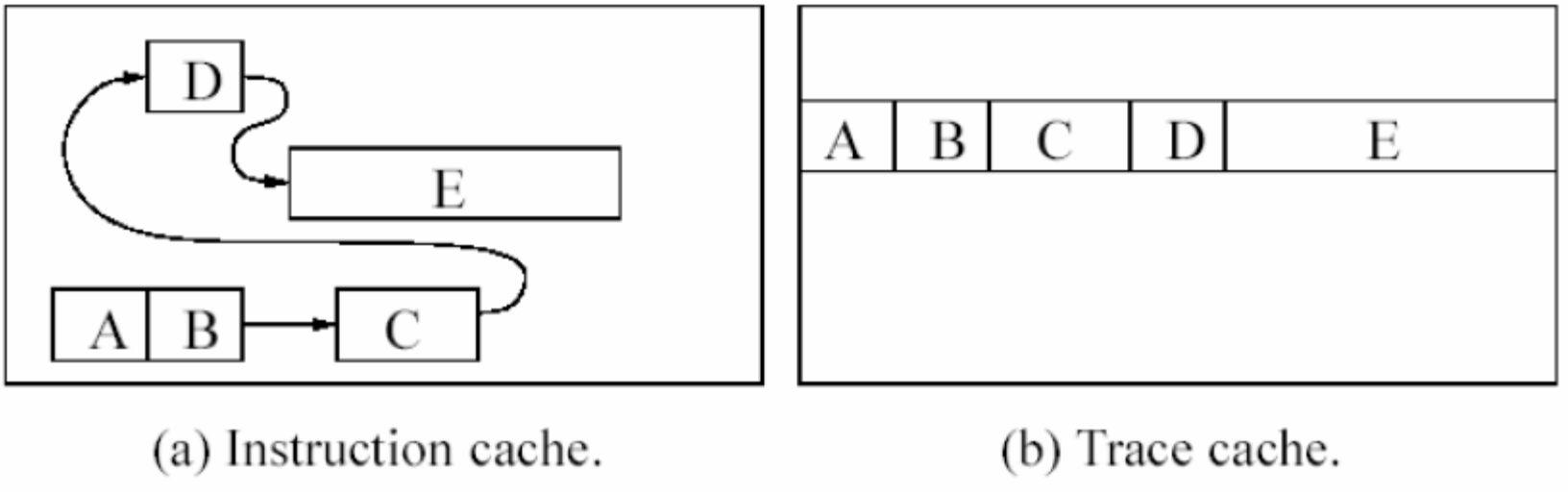
Code After Common
Subexpression Elimination

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

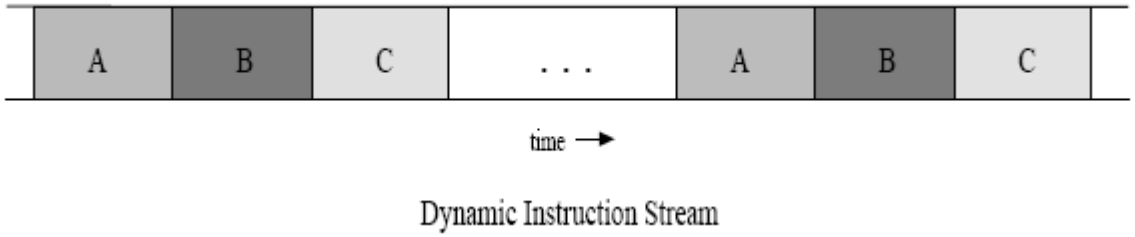
- A trace is a sequence of executed instructions.
- It is specified by a start address and the branch outcomes of control transfer instructions.
- Traces repeat: programs have frequently executed paths
- Trace cache idea: Store the dynamic instruction sequence in the same physical location.



(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)

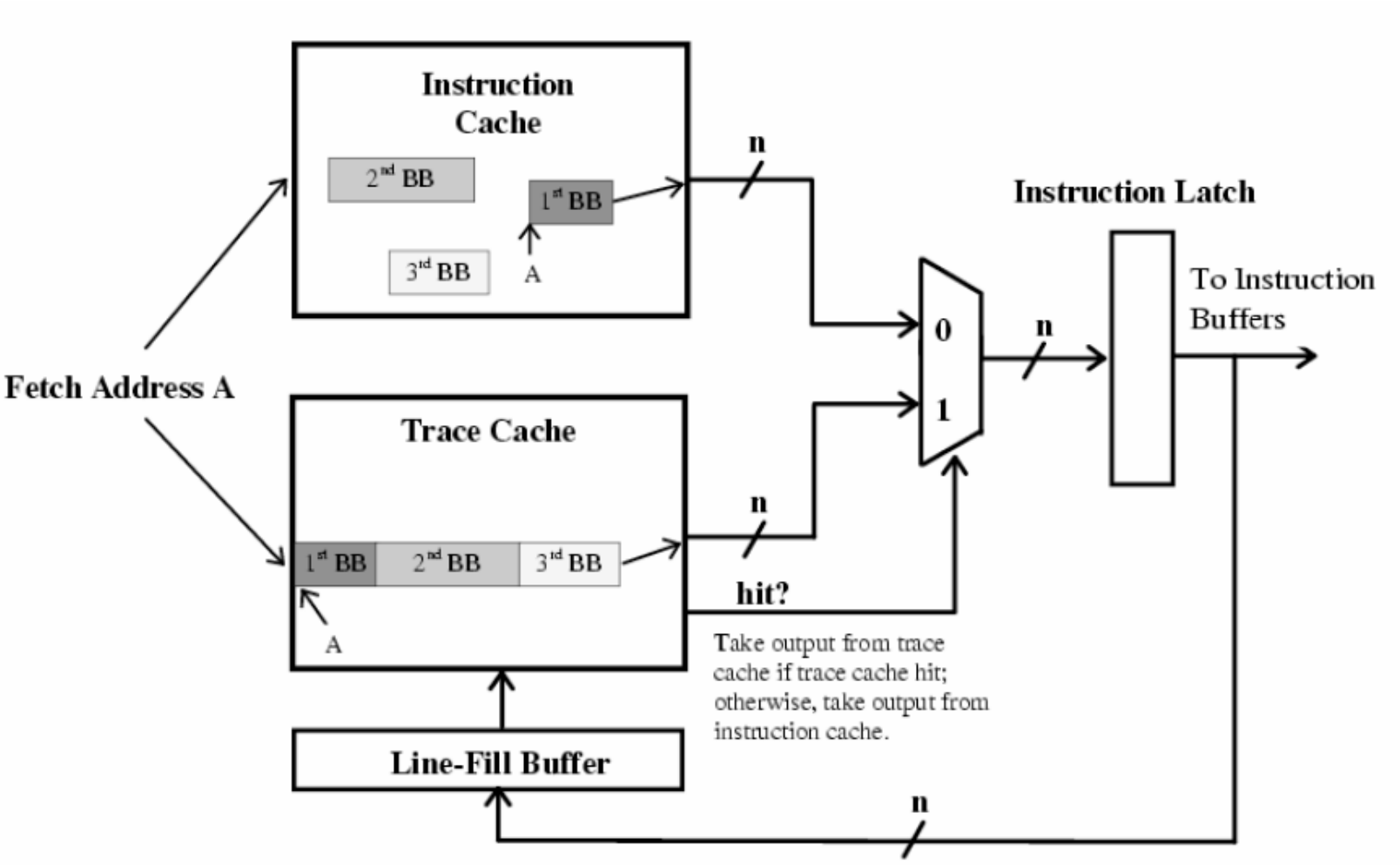


Dynamic Instruction Stream

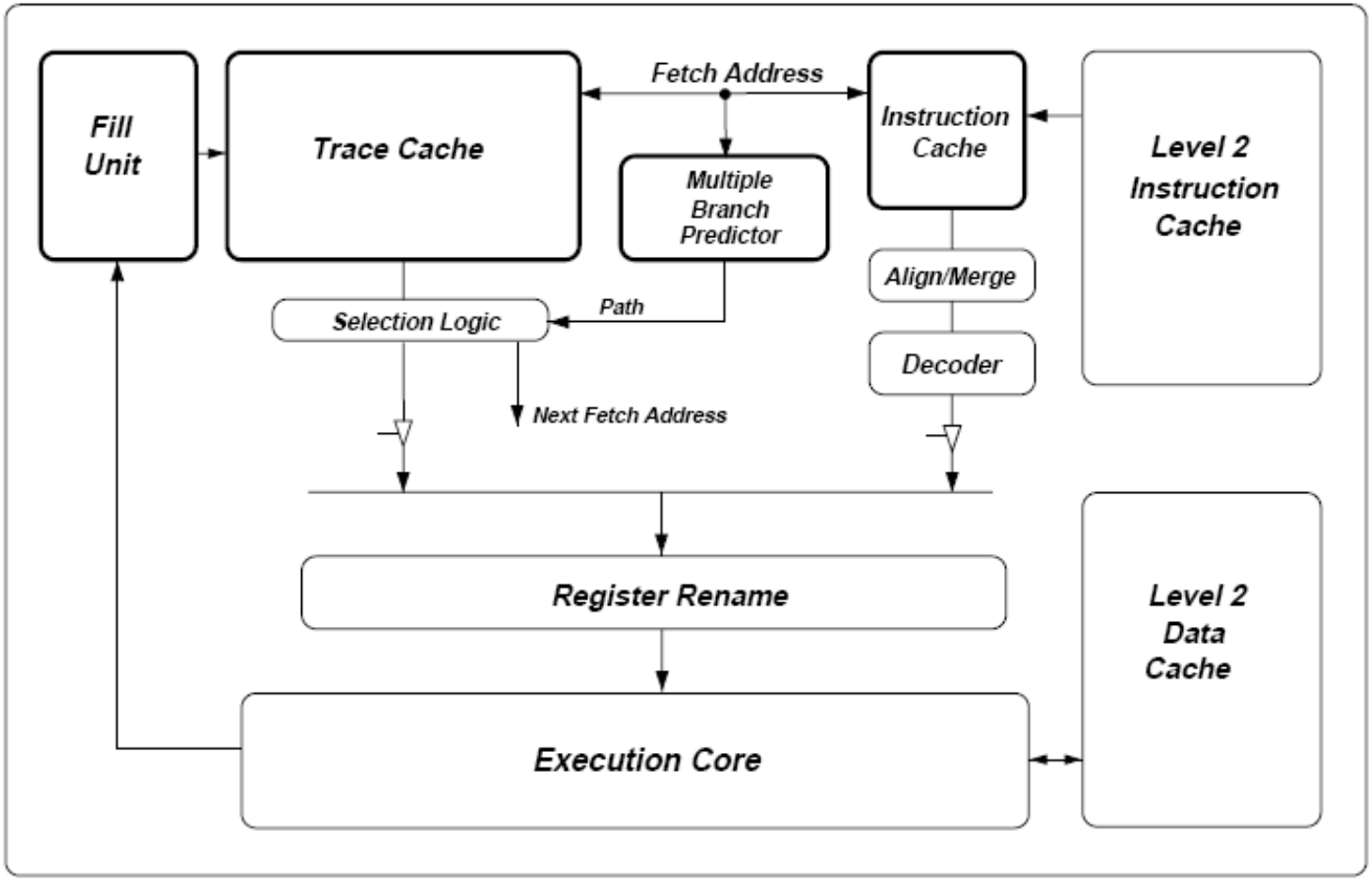
- Basic trace cache operation:
  - Fetch from consecutively-stored basic blocks (predict next trace or branches)
  - Verify the executed branch directions with the stored ones
  - If mismatch, flush the remaining portion of the trace

- 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



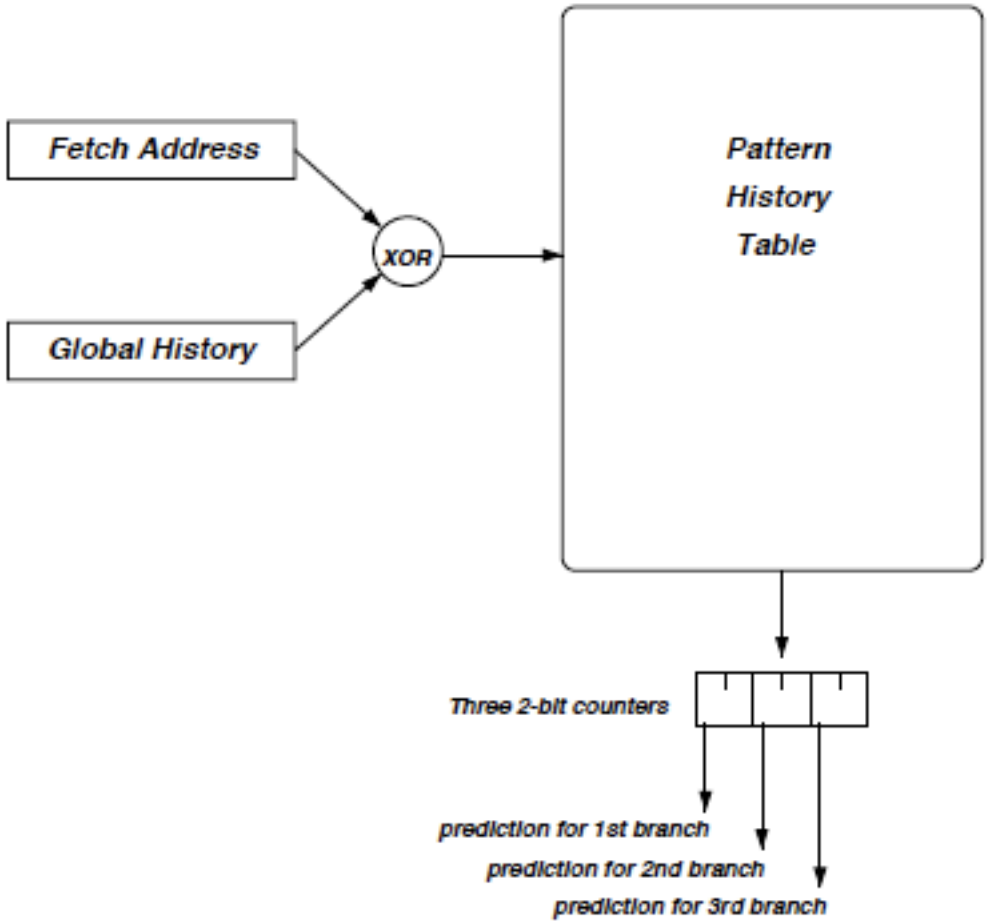
An Example Trace Cache Based Processor



- 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.

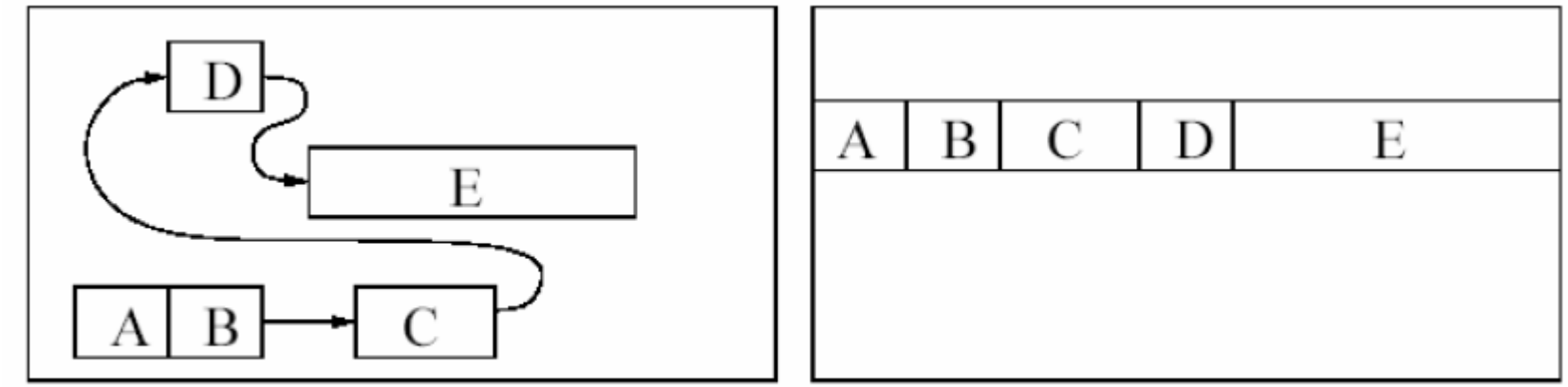


What Does A Trace Cache Line Store?

- 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

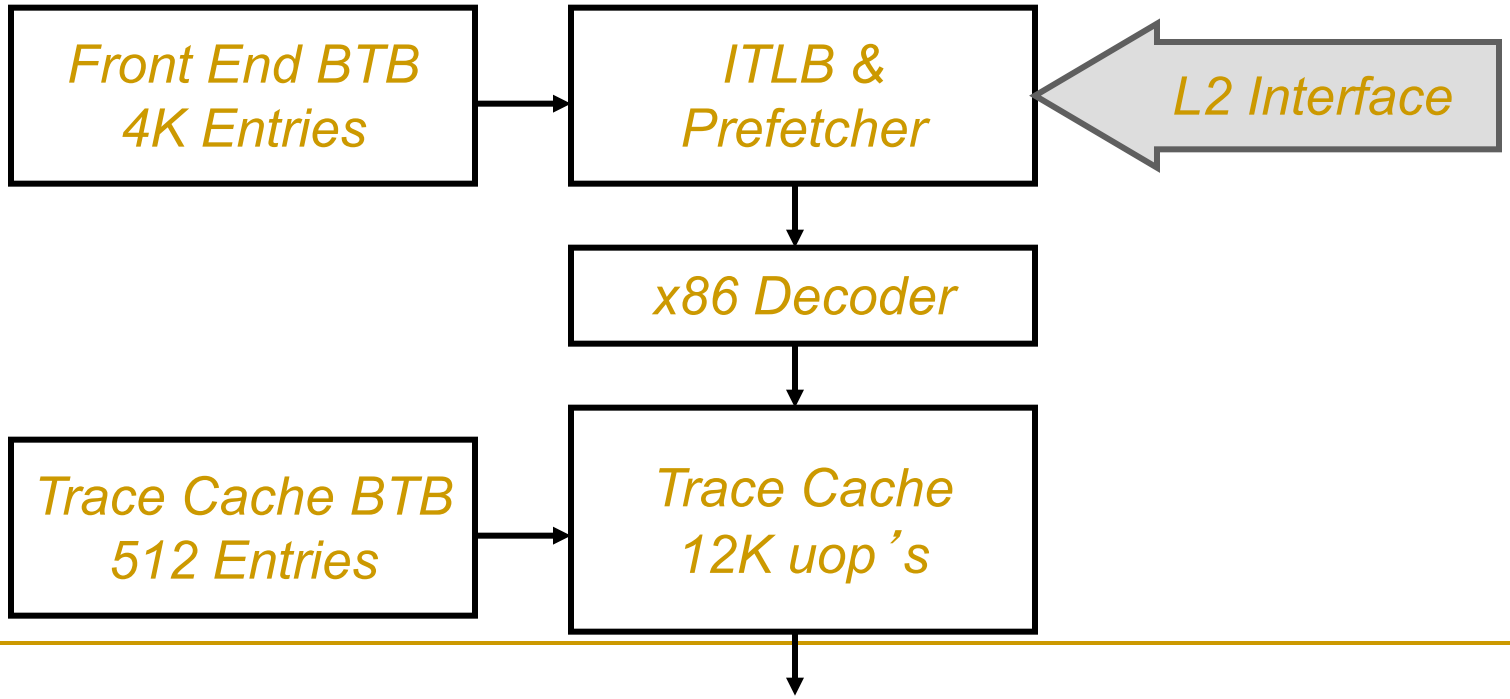


(a) Instruction cache. (b) Trace cache.

+ 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



Delayed Branching (I)

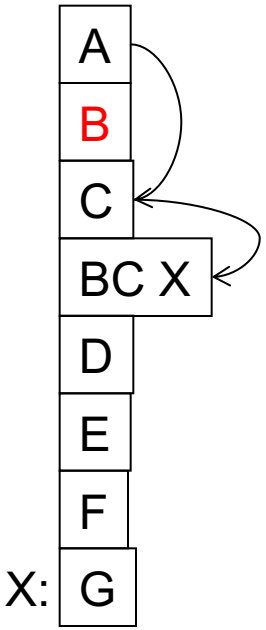
- Change the semantics of a branch instruction
  - Branch after N instructions
  - Branch after N cycles
- Idea: Delay the execution of a branch. N instructions (delay slots) that come after the branch are **always** executed regardless of branch direction.

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

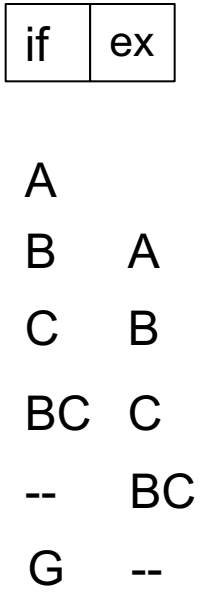
- Unconditional branch: Easier to find instructions to fill the delay slot
- Conditional branch: Condition computation should not depend on instructions in delay slots → difficult to fill the delay slot

Delayed Branching (II)

Normal code:

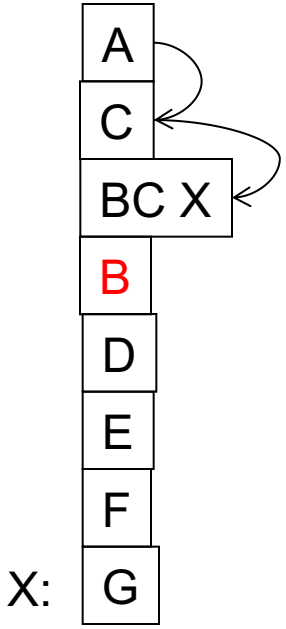


Timeline:

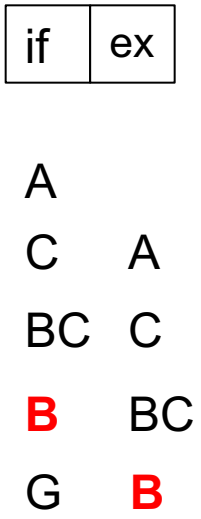


6 cycles

Delayed branch code:



Timeline:

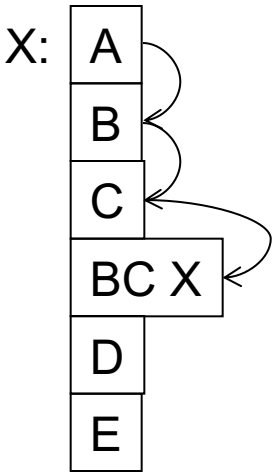


5 cycles

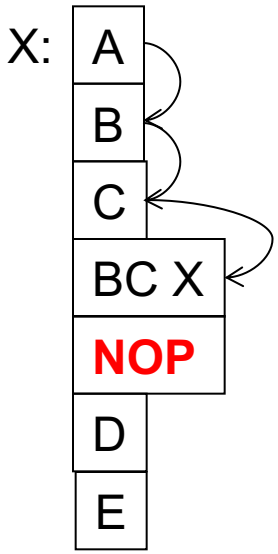
Fancy Delayed Branching (III)

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

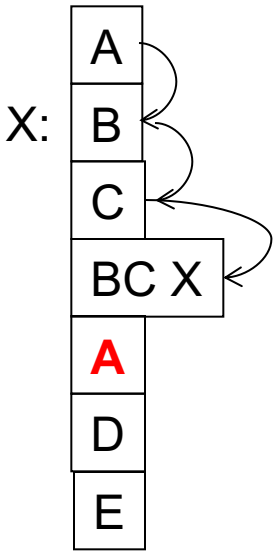
Normal code:



Delayed branch code:



Delayed branch w/ squashing:



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

    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

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

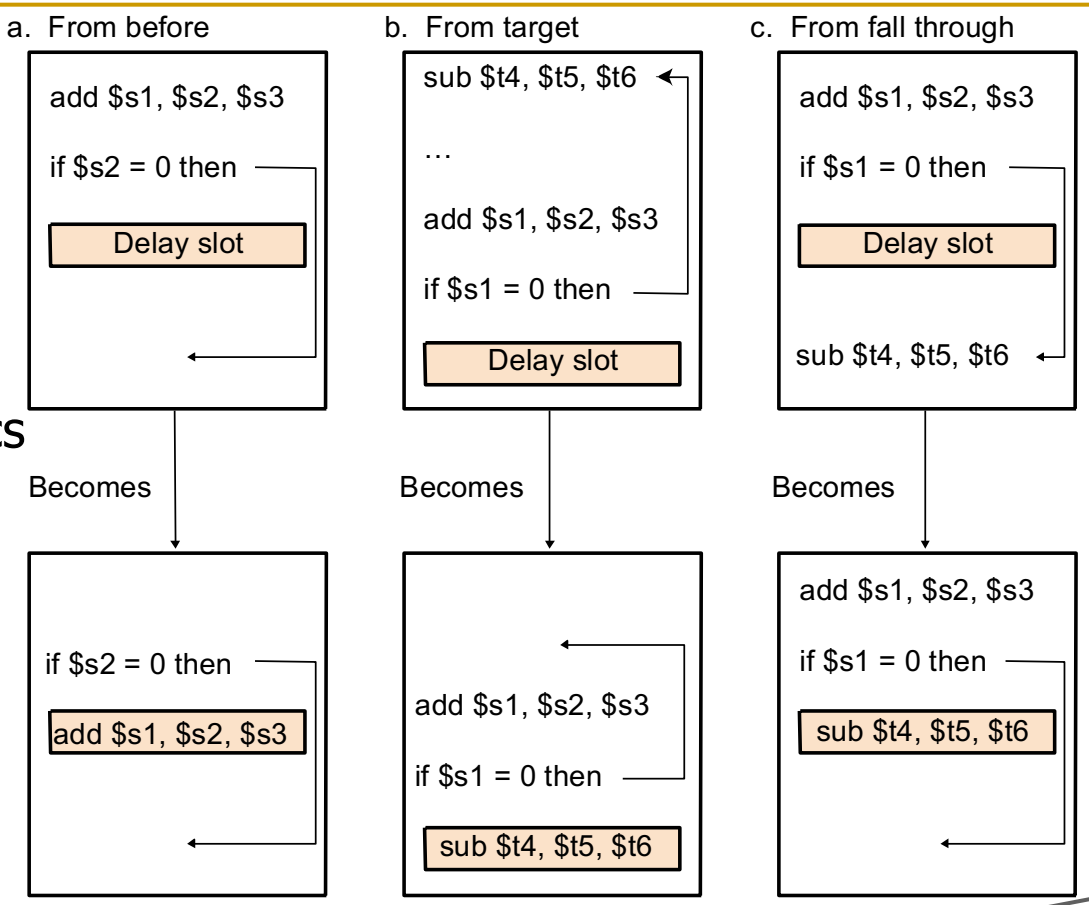
  -- Ties ISA semantics to hardware implementation

    -- SPARC, MIPS, HP-PA: 1 delay slot

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

An Aside: Filling the Delay Slot

reordering independent instructions does not change program semantics



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?

Safe?