18-447
Computer Architecture
Lecture 16: Systolic Arrays & Static Scheduling

Prof. Onur Mutlu

Carnegie Mellon University

Spring 2015, 2/23/2015

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, …

- Alternative Approaches to Instruction Level Parallelism

Approaches to (Instruction-Level) Concurrency

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

- Static Instruction Scheduling

Homework 3.1: Feedback Form

- Due Today!

- I would like your feedback on the course
- Easy to fill in
- Can submit anonymously, if you wish
- Worth 0.25% of your grade (extra credit)
- Need to get checked off after submitting to get your grade points
  - Can email
  - If anonymous, show that you are turning in and have a TA check you off

A Few Things

- **Midterm I Date**
  - March 18 or 20

- **Collaboration on Labs**
  - All labs individual – no collaboration permitted

- **Collaboration on homeworks**
  - You can collaborate
  - But need to submit individual writeups on your own

Wednesday Seminar – Noon-1pm GHC 6115

- ## Data Center Computers: Modern Challenges in CPU Design
  - Richard Sites, Google
  - Feb 25, noon-1pm, GHC 6115
- Computers used as datacenter servers have usage patterns that differ substantially from those of desktop or laptop computers. We discuss four key differences in usage and their first-order implications for designing computers that are particularly well-suited as servers: data movement, thousands of transactions per second, program isolation, and measurement underpinnings.
- Maintaining high-bandwidth data movement requires coordinated design decisions throughout the memory system, instruction-issue system, and even instruction set. Serving thousands of transactions per second requires continuous attention to all sources of delay -- causes of long-latency transactions. Unrelated programs running on shared hardware produce delay through undesired interference; isolating programs from one another needs further hardware help. And finally, when running datacenter servers as a business it is vital to be able to observe and hence decrease inefficiencies across dozens of layers of software and thousands of interacting servers. There are myriad open research problems related to these issues.

Isolating Programs from One Another

- Remember matlab vs. gcc?
- We will get back to this again

- In the meantime, if you are curious, take a look at:
  - Subramanian et al., "MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems," HPCA 2013.
  - Moscibroda and Mutlu, "Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems," USENIX Security 2007.

Recap of Last Lecture

- **GPUs**
  - Programming Model vs. Execution Model Separation
  - GPUs: SPMD programming on SIMD/SIMT hardware
  - SIMT Advantages vs. Traditional SIMD
  - Warps, Fine-grained Multithreading of Warps
  - SIMT Memory Access
  - Branch Divergence Problem in SIMT
  - Dynamic Warp Formation/Merging

- **VLIW**
  - Philosophy: RISC and VLIW
  - VLIW vs. SIMD vs. Superscalar
  - Tradeoffs and Advantages

- **DAE (Decoupled Access/Execute)**
  - Dynamic and Static Scheduling

Systolic Arrays

Systolic Arrays: Motivation

- Goal: design an accelerator that has
  - Simple, regular design (keep # unique parts small and regular)
  - High concurrency → high performance
  - Balanced computation and I/O (memory) bandwidth

- Idea: Replace a single processing element (PE) with a regular array of PEs and carefully orchestrate flow of data between the PEs
  - such that they collectively transform a piece of input data before outputting it to memory

- Benefit: Maximizes computation done on a single piece of data element brought from memory
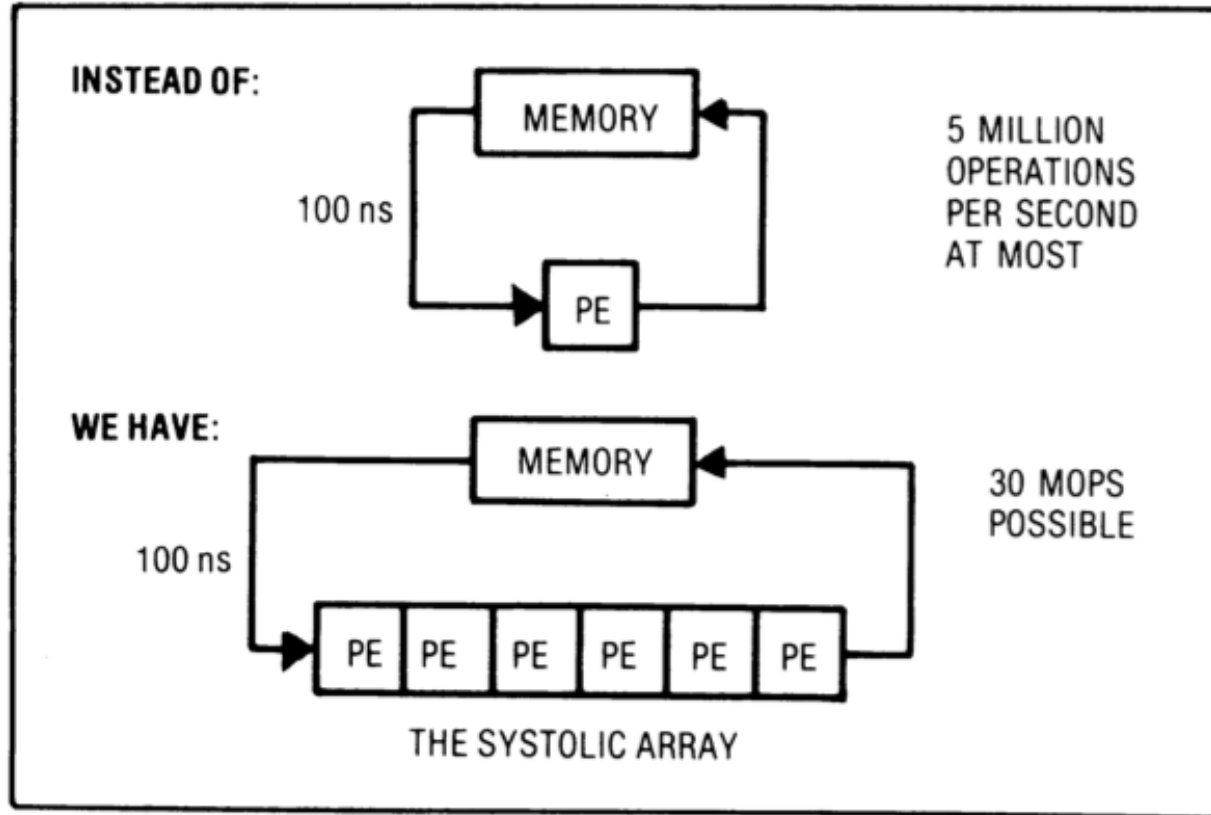
Systolic Arrays

INSTEAD OF:

MEMORY

100 ns

PE

5 MILLION OPERATIONS PER SECOND AT MOST

WE HAVE:

MEMORY

100 ns

PE PE PE PE PE PE

THE SYSTOLIC ARRAY

30 MOPS POSSIBLE

Memory: heart
PEs: cells

Memory pulses data through cells

Figure 1. Basic principle of a systolic system.

- H. T. Kung, "Why Systolic Architectures?," IEEE Computer 1982.

Why Systolic Architectures?

- Idea: Data flows from the computer memory in a rhythmic fashion, passing through many processing elements before it returns to memory

- Similar to an assembly line of processing elements
  - Different people work on the same car
  - Many cars are assembled simultaneously
  - Can be two-dimensional

- Why? Special purpose accelerators/architectures need
  - Simple, regular design (keep # unique parts small and regular)
  - High concurrency → high performance
  - Balanced computation and I/O (memory) bandwidth

Systolic Architectures

- Basic principle: Replace a single PE with a regular array of PEs and carefully orchestrate flow of data between the PEs
  - Balance computation and memory bandwidth

Figure 1. Basic principle of a systolic system.

- Differences from pipelining:
  - These are individual PEs
  - Array structure can be non-linear and multi-dimensional
  - PE connections can be multidirectional (and different speed)
  - PEs can have local memory and execute kernels (rather than a piece of the instruction)
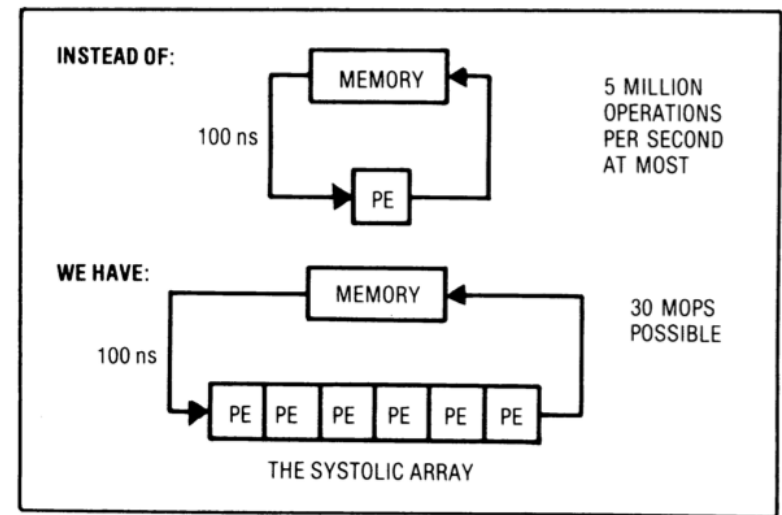
Systolic Computation Example

- Convolution
  - Used in filtering, pattern matching, correlation, polynomial evaluation, etc …
  - Many image processing tasks

Given the sequence of weights $\{w_1, w_2, \ldots, w_k\}$ and the input sequence $\{x_1, x_2, \ldots, x_n\}$,

compute the result sequence $\{y_1, y_2, \ldots, y_{n+1-k}\}$ defined by

$$y_i = w_1 x_i + w_2 x_{i+1} + \ldots + w_k x_{i+k-1}$$

Systolic Computation Example: Convolution

- $y1 = w1x1 + w2x2 + w3x3$
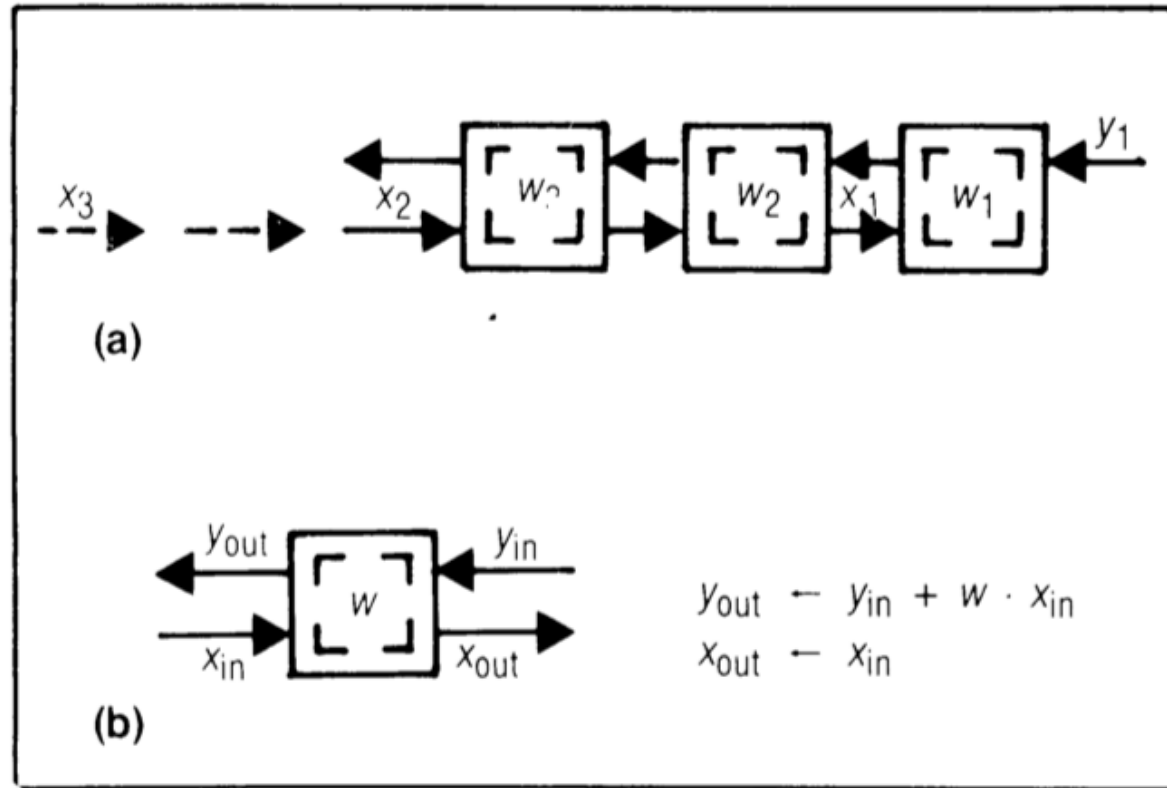
- $y2 = w1x2 + w2x3 + w3x4$

- $y3 = w1x3 + w2x4 + w3x5$

Figure 8. Design W1: systolic convolution array (a) and cell (b) where w_i's stay and x_i's and y_i's move systolically in opposite directions.
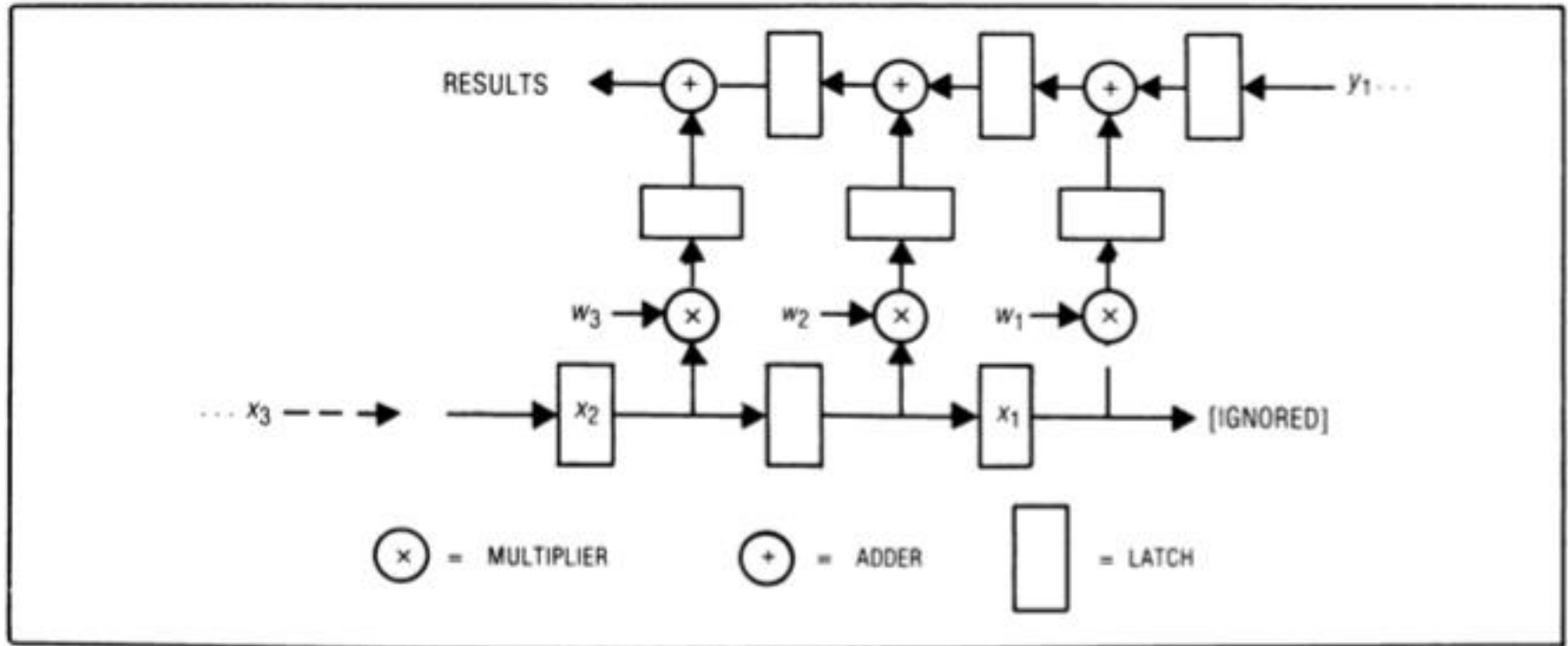
Systolic Computation Example: Convolution

Figure 10. Overlapping the executions of multiply and add in design W1.

- Worthwhile to implement adder and multiplier separately to allow overlapping of add/mul executions

Systolic Computation Example: Convolution

- One needs to carefully orchestrate when data elements are input to the array

- And when output is buffered

- This gets more involved when
  - Array dimensionality increases
  - PEs are less predictable in terms of latency

Systolic Arrays: Pros and Cons

- **Advantage:**
  - Specialized (computation needs to fit PE organization/functions)
    - → improved efficiency, simple design, high concurrency/ performance
    - → good to do more with less memory bandwidth requirement

- **Downside:**
  - Specialized
    - → not generally applicable because computation needs to fit the PE functions/organization

More Programmability

- Each PE in a systolic array
    - Can store multiple "weights"
    - Weights can be selected on the fly
    - Eases implementation of, e.g., adaptive filtering

- Taken further
    - Each PE can have its own data and instruction memory
    - Data memory → to store partial/temporary results, constants
    - Leads to stream processing, pipeline parallelism
        - More generally, staged execution

Pipeline Parallelism
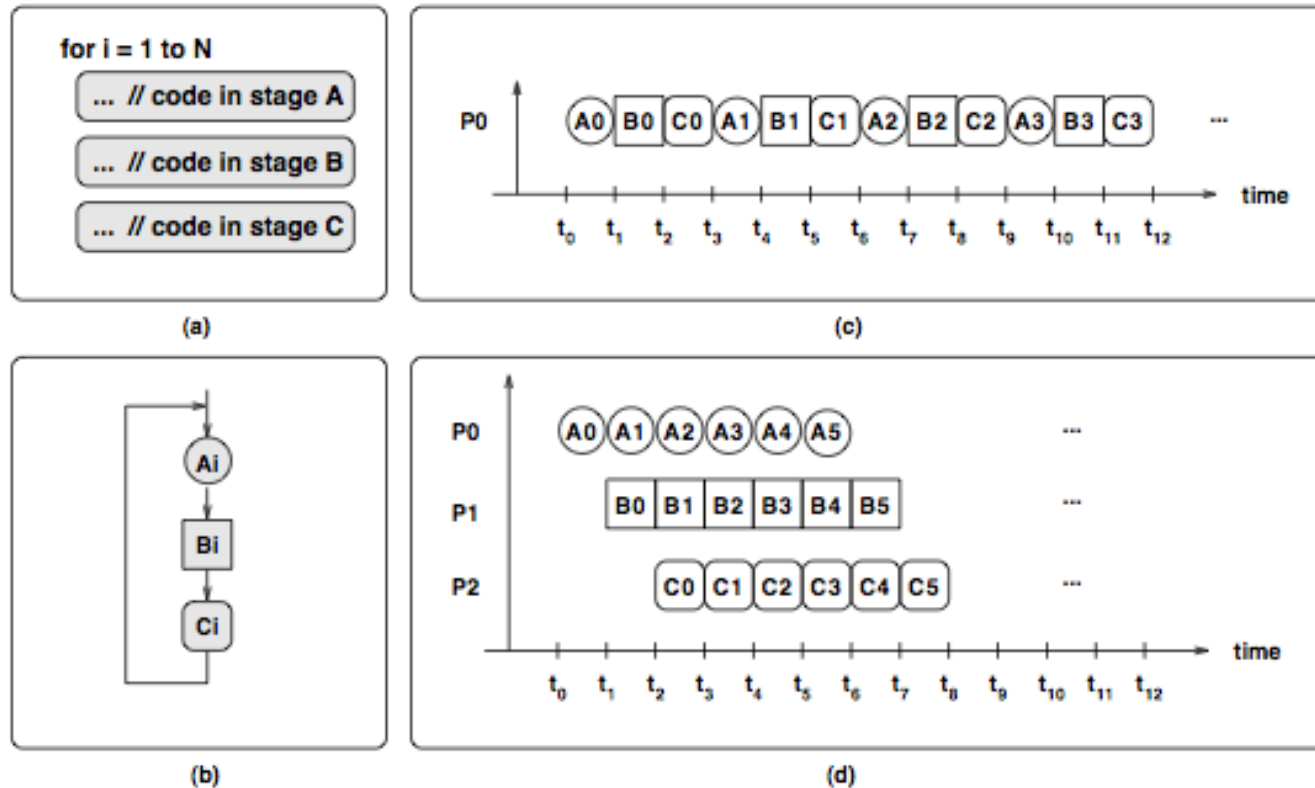

Figure 1. (a) The code of a loop, (b) Each iteration is split into 3 pipeline stages: A, B, and C. Iteration i comprises Ai, Bi, Ci. (c) Sequential execution of 4 iterations. (d) Parallel execution of 6 iterations using pipeline parallelism on a three-core machine. Each stage executes on one core.

Stages of Pipelined Programs

- Loop iterations are divided into code segments called *stages*
- Threads execute stages on different cores

loop {
  Compute1 **A**

  Compute2 **B**
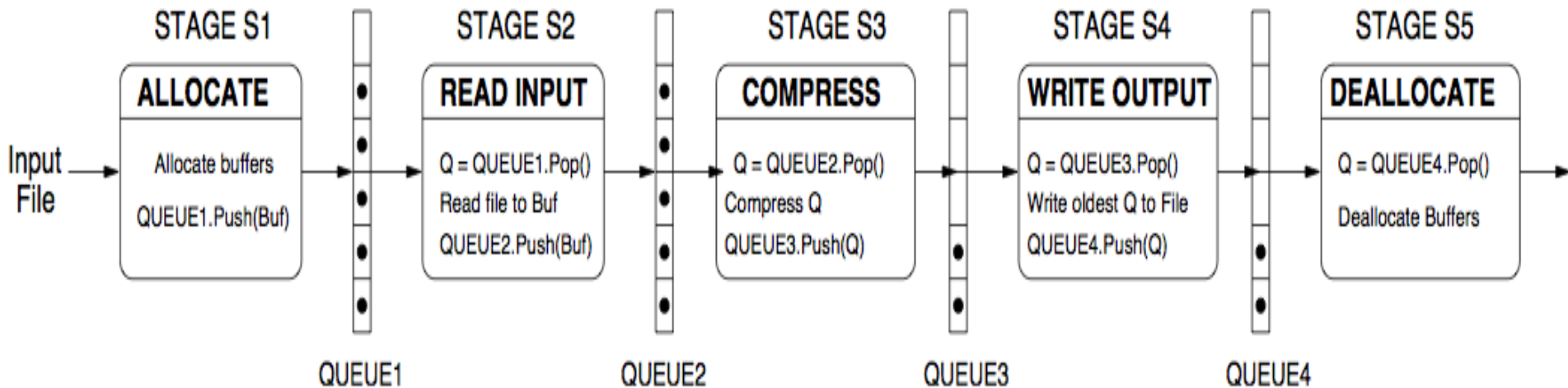
  Compute3 **C**
}
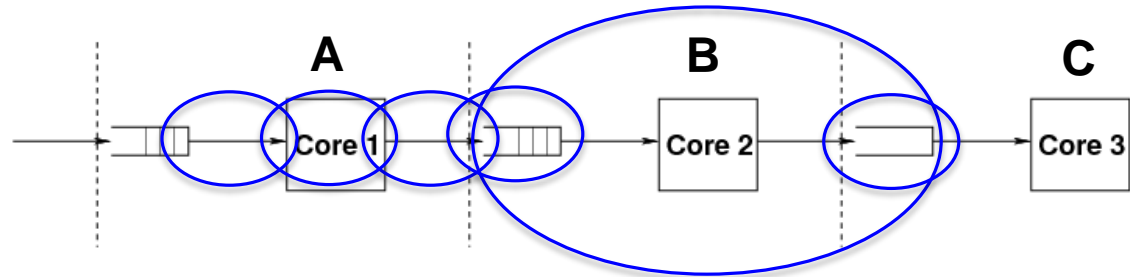
Pipelined File Compression Example

Figure 3. File compression algorithm executed using pipeline parallelism

Systolic Array

- Advantages
  - Makes multiple uses of each data item → reduced need for fetching/refetching
  - High concurrency
  - Regular design (both data and control flow)

- Disadvantages
  - Not good at exploiting irregular parallelism
  - Relatively special purpose → need software, programmer support to be a general purpose model

Example Systolic Array: The WARP Computer

- HT Kung, CMU, 1984-1988

- Linear array of 10 cells, each cell a 10 Mflop programmable processor
- Attached to a general purpose host machine
- HLL and optimizing compiler to program the systolic array
- Used extensively to accelerate vision and robotics tasks

- Annaratone et al., "Warp Architecture and Implementation," ISCA 1986.
- Annaratone et al., "The Warp Computer: Architecture, Implementation, and Performance," IEEE TC 1987.
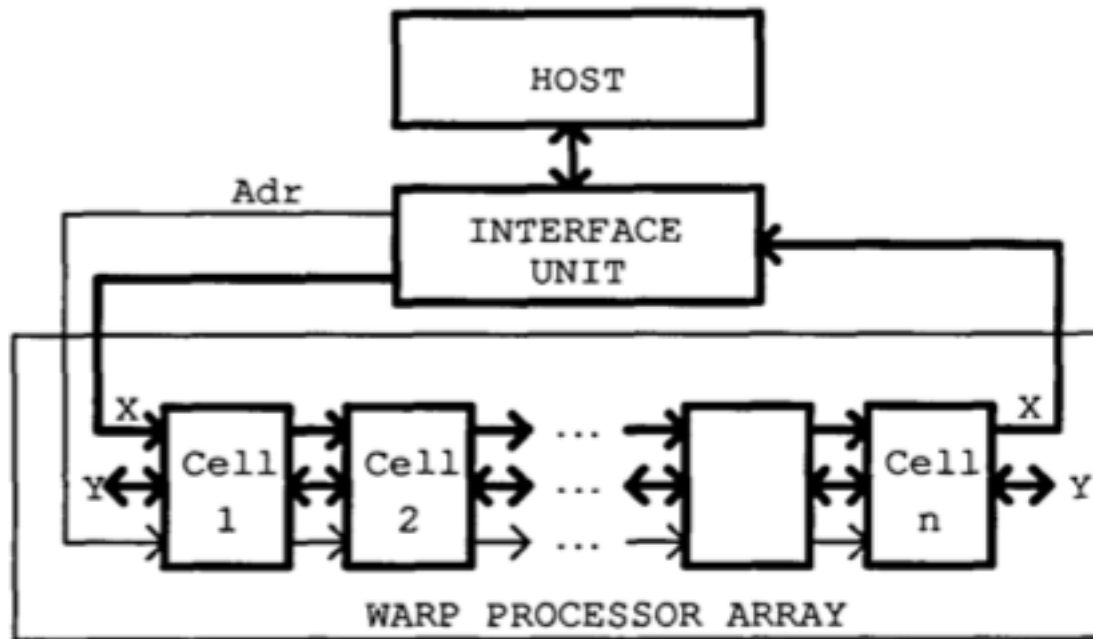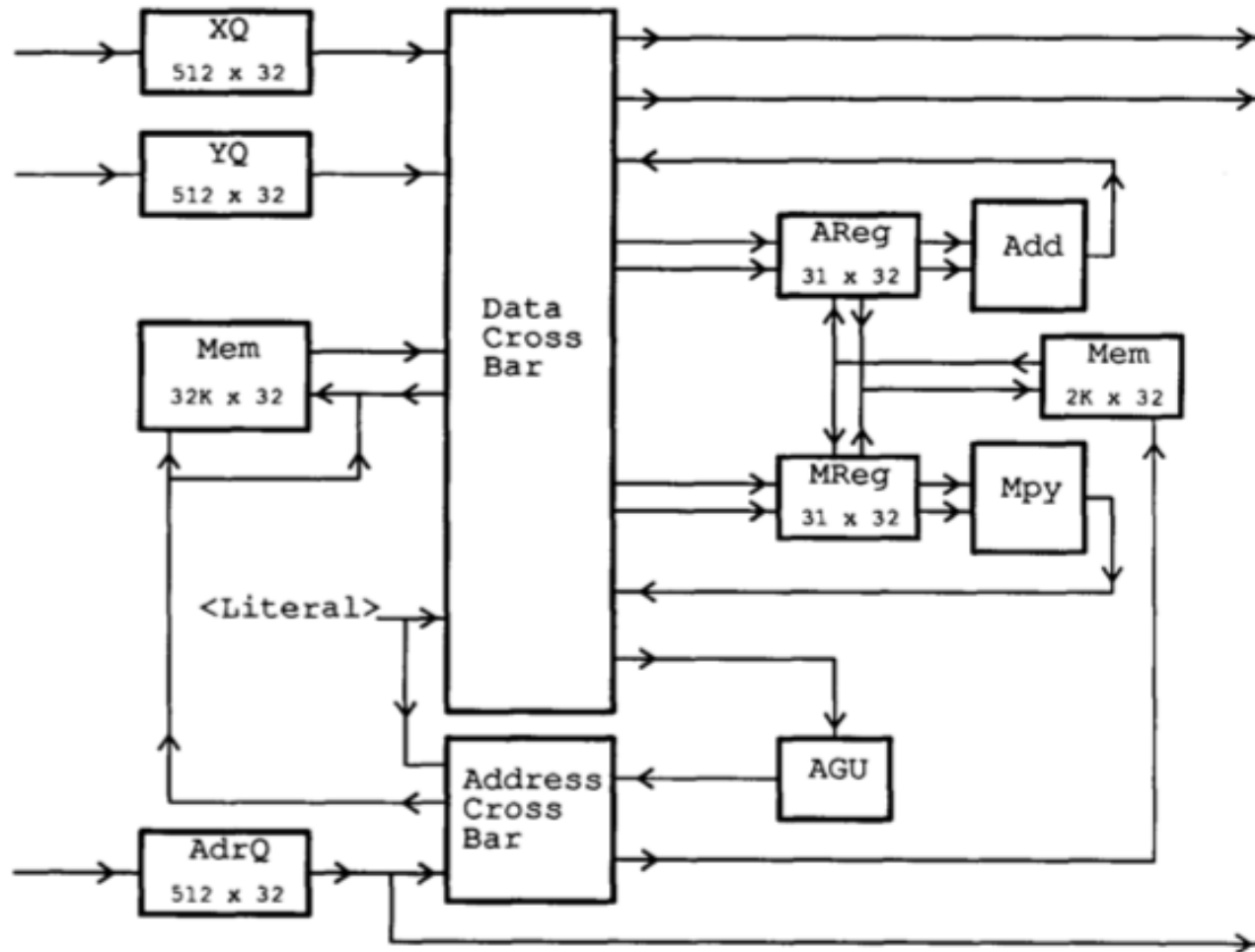
The WARP Computer

Figure 1: Warp system overview

The WARP Cell

Figure 2: Warp cell data path

Systolic Arrays vs. SIMD

- Food for thought…

Agenda Status

- 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, …

- Alternative Approaches to Instruction Level Parallelism

Approaches to (Instruction-Level) Concurrency

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

- Static Instruction Scheduling

Some More Recommended Readings

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

- Smith, "Decoupled Access/Execute Compute Architectures," ISCA 1982, ACM TOCS 1984.

- H. T. Kung, "Why Systolic Architectures?," IEEE Computer 1982.

- Huck et al., "Introducing the IA-64 Architecture," IEEE Micro 2000.

- Rau and Fisher, "Instruction-level parallel processing: history, overview, and perspective," Journal of Supercomputing, 1993.

- Faraboschi et al., "Instruction Scheduling for Instruction Level Parallel Processors," Proc. IEEE, Nov. 2001.

Static Instruction Scheduling (with a Slight Focus on VLIW)

Agenda

- **Static Scheduling**
  - Key Questions and Fundamentals

- **Enabler of Better Static Scheduling: Block Enlargement**
  - Predicated Execution
  - Loop Unrolling
  - Trace
  - Superblock
  - Hyperblock
  - Block-structured ISA

Key Questions

Q1. How do we find independent instructions to fetch/execute?

Q2. How do we enable more compiler optimizations?
    e.g., common subexpression elimination, constant propagation, dead code elimination, redundancy elimination, …

Q3. How do we increase the instruction fetch rate?
    i.e., have the ability to fetch more instructions per cycle

A: Enabling the compiler to optimize across a larger number of instructions that will be executed straight line (without branches getting in the way) eases all of the above

How Do We Enable Straight-Line Code?

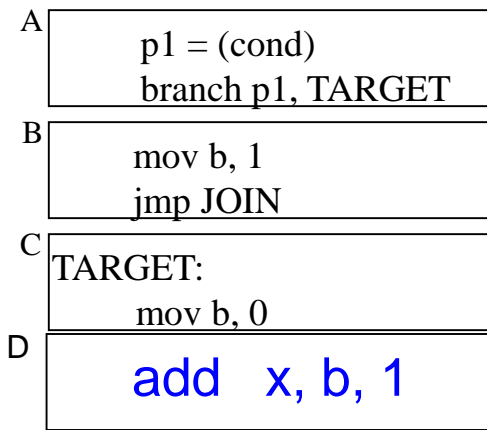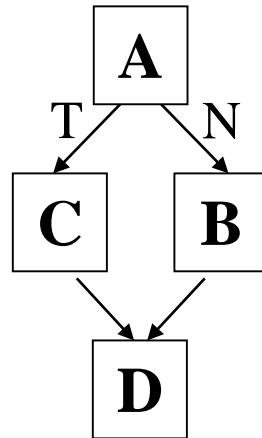- **Get rid of control flow**
  - Predicated Execution
  - Loop Unrolling
  - …

- **Optimize frequently executed control flow paths**
  - Trace
  - Superblock
  - Hyperblock
  - Block-structured ISA
  - …

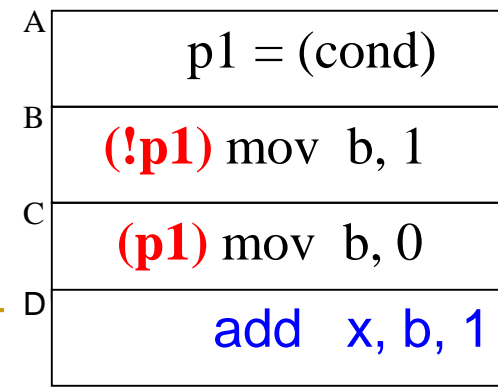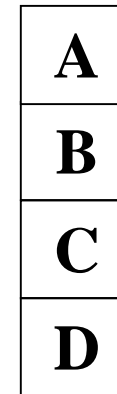# Review: Predication (Predicated Execution)

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

(normal branch code)   (predicated code)

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

A
T / \ N
C   B
 \   /
   D

A	p1 = (cond) branch p1, TARGET
B	mov b, 1 jmp JOIN
C	TARGET: mov b, 0
D	add x, b, 1

A	p1 = (cond)
B	**(!p1)** mov b, 1
C	**(p1)** mov b, 0
D	add x, b, 1

A
B
C
D

Review: Loop Unrolling

```
i = 1;
while ( i < 100 ) {
      a[i] = b[i+1] + (i+1)/m
      b[i] = a[i-1] - i/m
      i = i + 1
}
```

```
i = 1;
while ( i < 100 ) {
      a[i] = b[i+1] + (i+1)/m
      b[i] = a[i-1] - i/m

      a[i+1] = b[i+2] + (i+2)/m
      b[i+1] = a[i] - (i+1)/m
      i = i + 2
}
```

- **Idea:** Replicate loop body multiple times within an iteration

+ Reduces loop maintenance overhead
  - Induction variable increment or loop condition test

+ Enlarges basic block (and analysis scope)
  - Enables code optimization and scheduling opportunities

-- What if iteration count not a multiple of unroll factor? (need extra code to detect this)

-- Increases code size

Some Terminology: Basic vs. Atomic Block

- Basic block: A sequence (block) of instructions with a single control flow entry point and a single control flow exit point
  - A basic block executes uninterrupted (if no exceptions/interrupts)

- Atomic block: A block of instructions where either all instructions complete or none complete
  - In most modern ISAs, the atomic unit of execution is at the granularity of an instruction
  - A basic block can be considered atomic (if there are no exceptions/interrupts and side effects observable in the middle of execution)
  - One can reorder instructions freely within an atomic block, subject only to true data dependences

VLIW: Finding Independent Operations

- Within a basic block, there is limited instruction-level parallelism (if the basic block is small)

- To find multiple instructions to be executed in parallel, the compiler needs to consider multiple basic blocks

- Problem: Moving an instruction above a branch is unsafe because instruction is not guaranteed to be executed

- Idea: Enlarge blocks at compile time by finding the frequently-executed paths
  - Trace scheduling
  - Superblock scheduling
  - Hyperblock scheduling
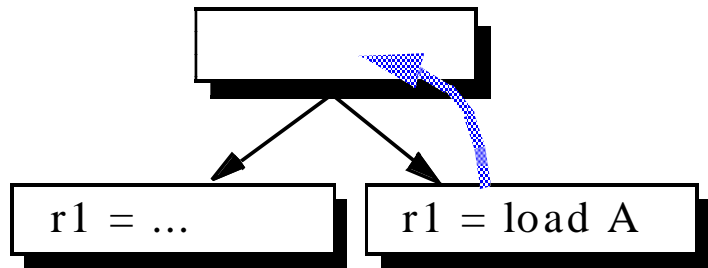
Safety and Legality in Code Motion

- Two characteristics of speculative code motion:
  - Safety: whether or not spurious exceptions may occur
  - Legality: whether or not result will be always correct
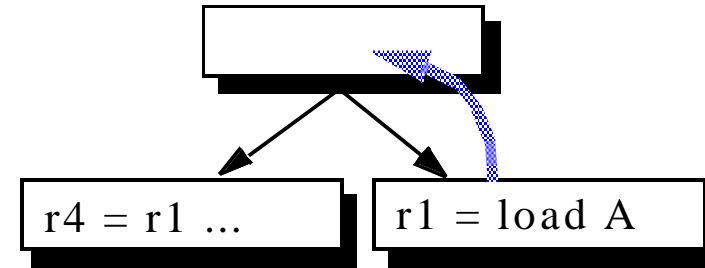- Four possible types of code motion:
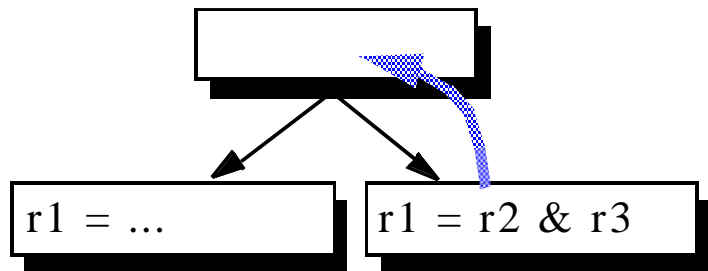
(a) safe and legal
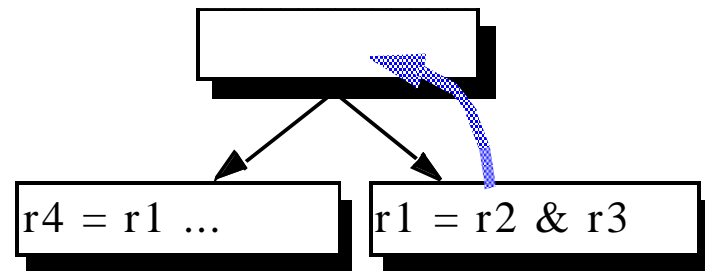
(b) illegal

(c) unsafe

(d) unsafe and illegal

Code Movement Constraints

- Downward
    - When moving an operation from a BB to one of its dest BB's,
        - all the other dest basic blocks should still be able to use the result of the operation
        - the other source BB's of the dest BB should not be disturbed

- Upward
    - When moving an operation from a BB to its source BB's
        - register values required by the other dest BB's must not be destroyed
        - the movement must not cause new exceptions

Trace Scheduling

- Trace: A frequently executed path in the control-flow graph (has multiple side entrances and multiple side exits)

- Idea: Find independent operations within a trace to pack into VLIW instructions.
  - Traces determined via profiling
  - Compiler adds fix-up code for correctness (if a side entrance or side exit of a trace is exercised at runtime, corresponding fix-up code is executed)

Trace Scheduling (II)

- There may be conditional branches from the middle of the trace (side exits) and transitions from other traces into the middle of the trace (side entrances).

- These control-flow transitions are ignored during trace scheduling.

- After scheduling, fix-up/bookkeeping code is inserted to ensure the correct execution of off-trace code.

- Fisher, "Trace scheduling: A technique for global microcode compaction," IEEE TC 1981.

Trace Scheduling Idea
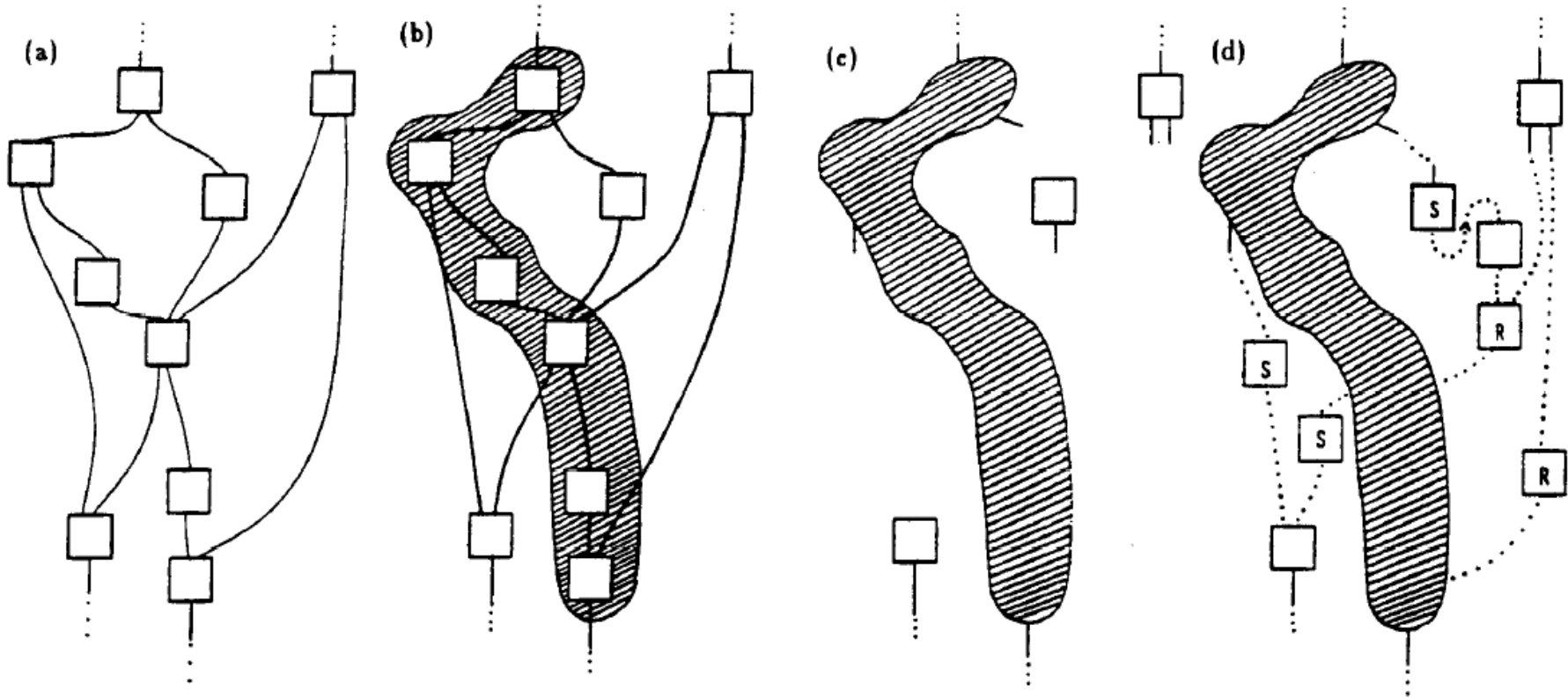
TRACE SCHEDULING LOOP-FREE CODE

Trace Scheduling (III)
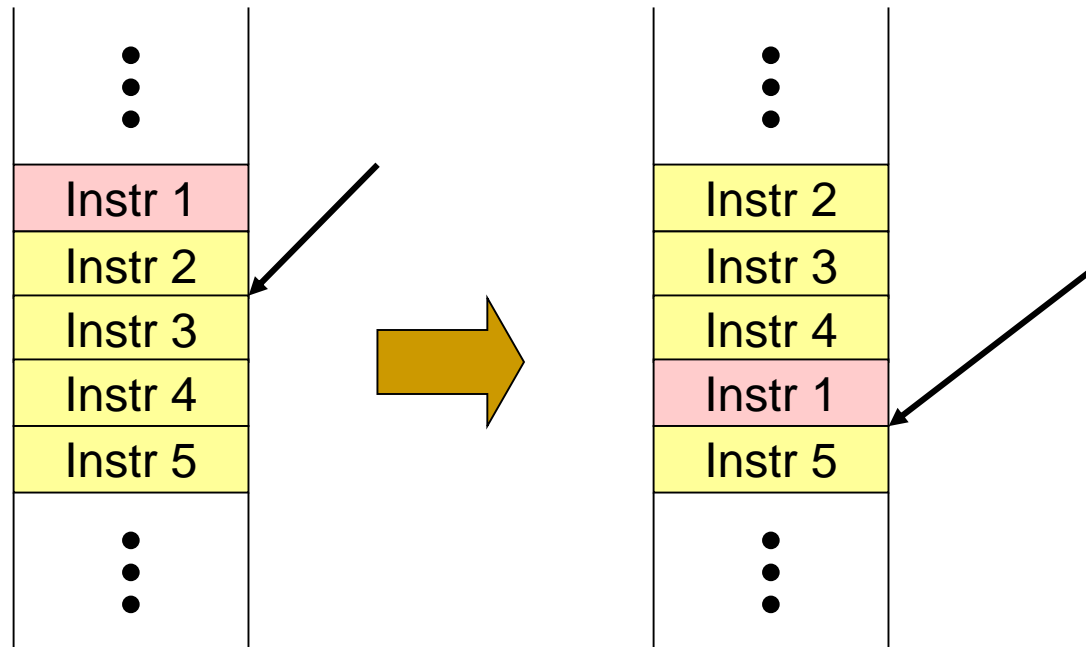
What bookeeping is required when Instr 1
is moved below the side entrance in the trace?
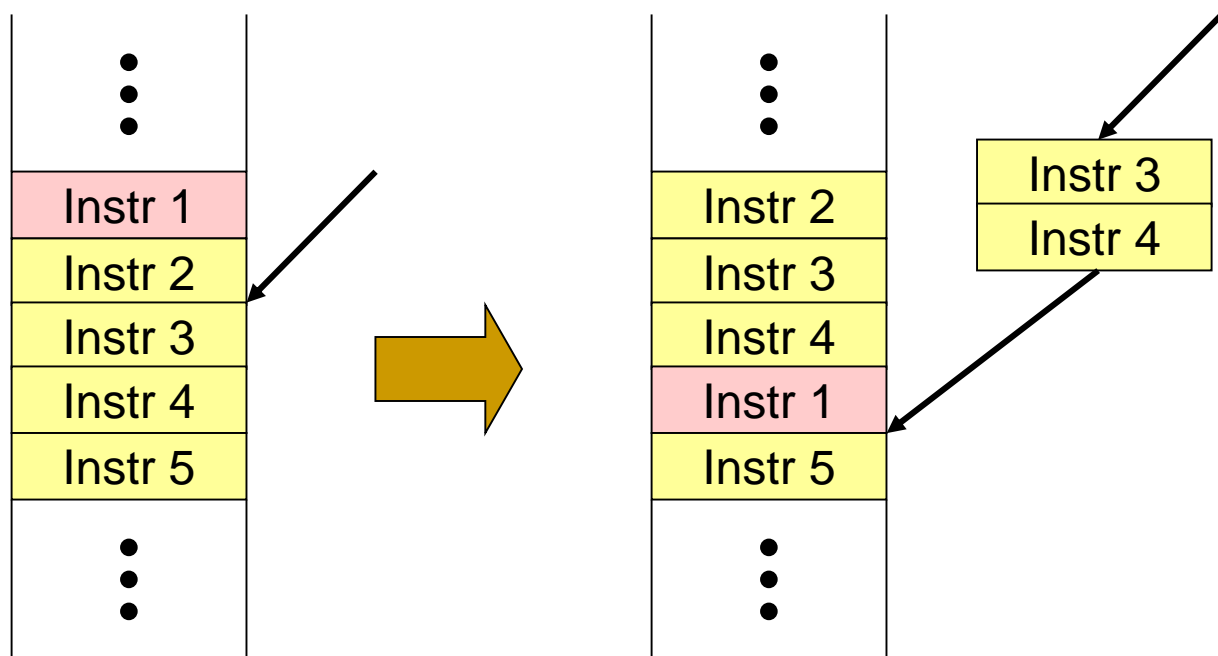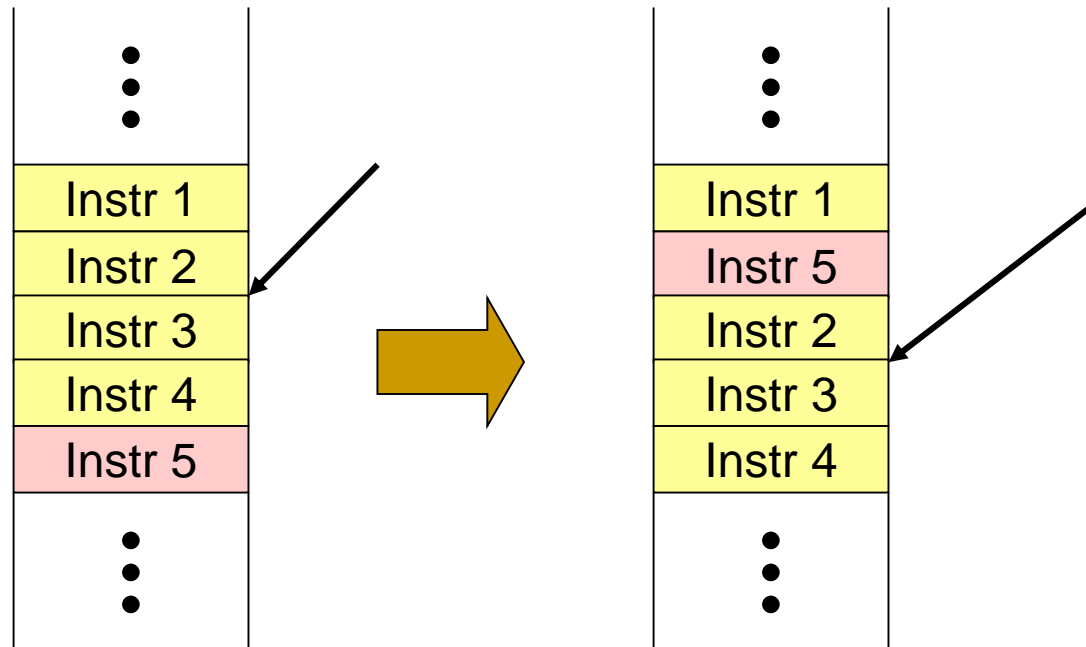
Trace Scheduling (IV)

Trace Scheduling (V)

What bookeeping is required when Instr 5 moves above the side entrance in the trace?

Trace Scheduling (VI)
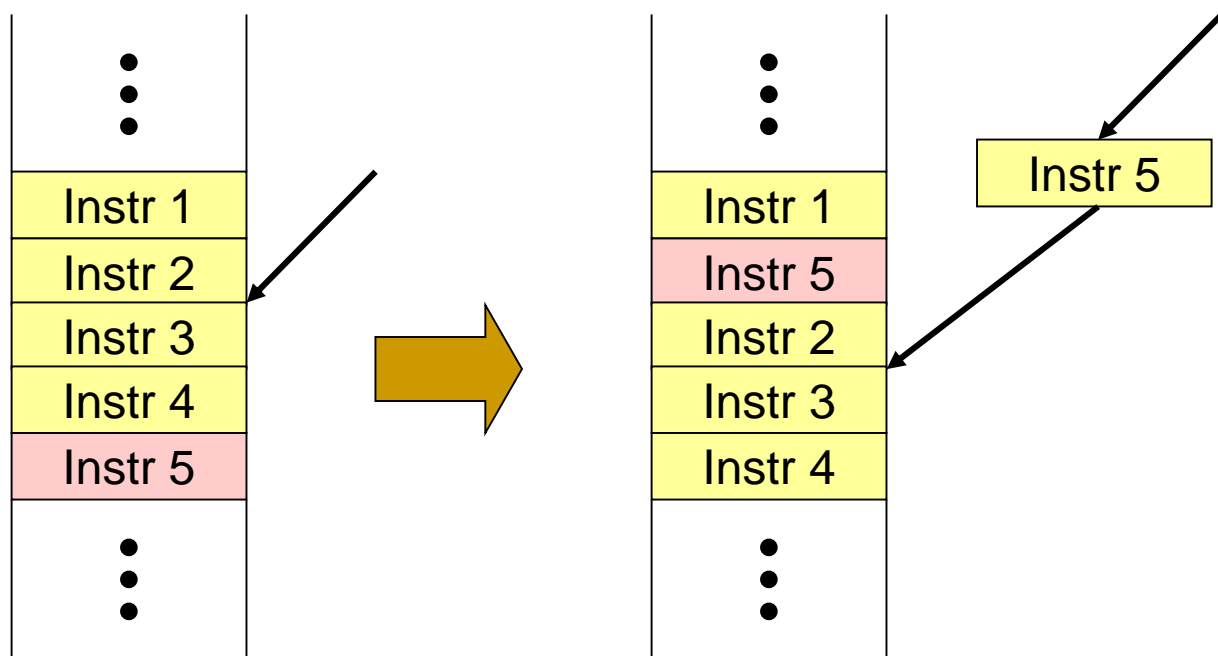
Trace Scheduling Fixup Code Issues

■ Sometimes need to copy instructions more than once to ensure correctness on all paths (see C below)
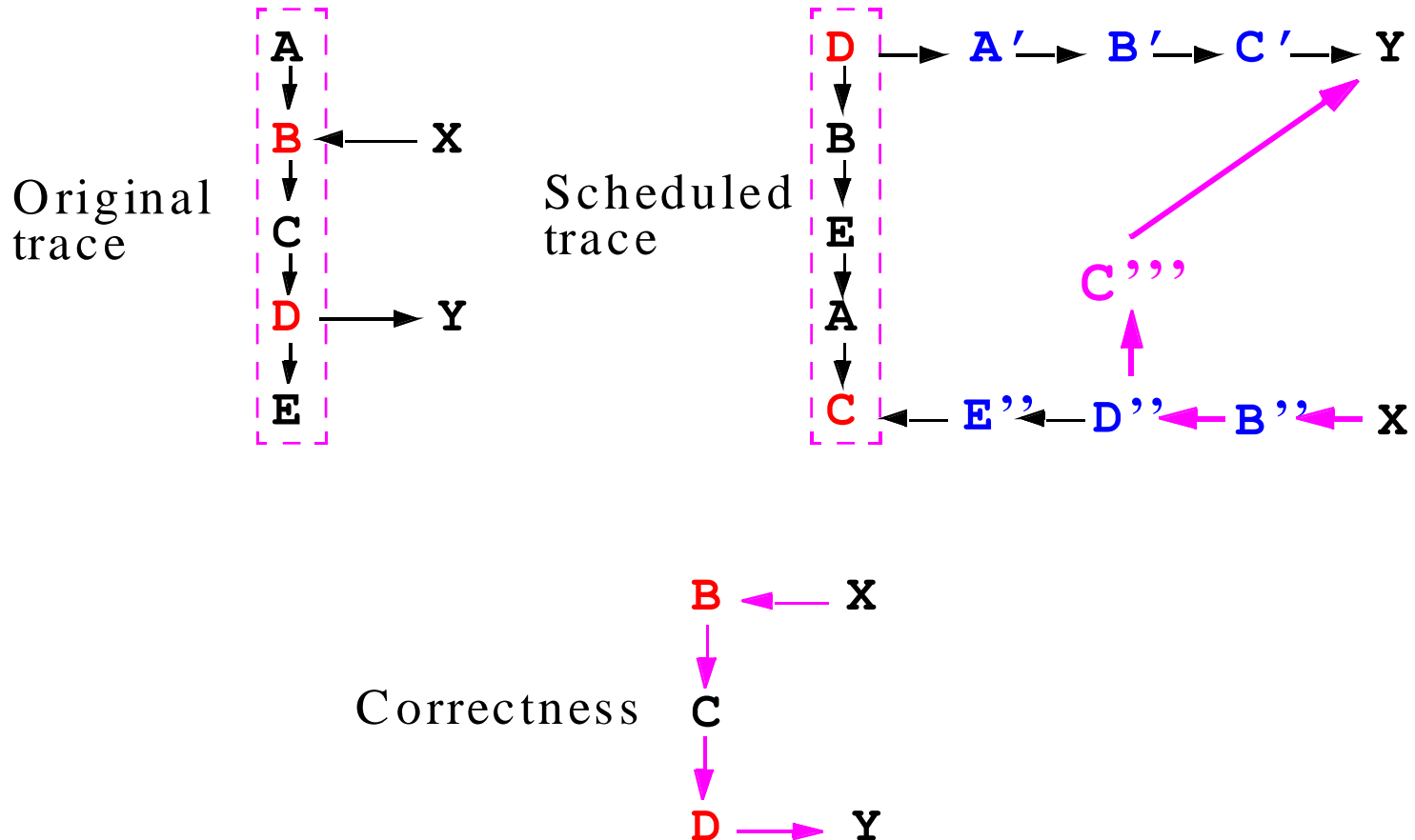
Trace Scheduling Overview

- **Trace Selection**
  - select seed block (the highest frequency basic block)
  - extend trace (along the highest frequency edges)
    forward (successor of the last block of the trace)
    backward (predecessor of the first block of the trace)
  - don't cross loop back edge
  - bound max_trace_length heuristically

- **Trace Scheduling**
  - build data precedence graph for a whole trace
  - perform list scheduling and allocate registers
  - add compensation code to maintain semantic correctness

- **Speculative Code Motion (upward)**
  - move an instruction above a branch if safe

Data Precedence Graph
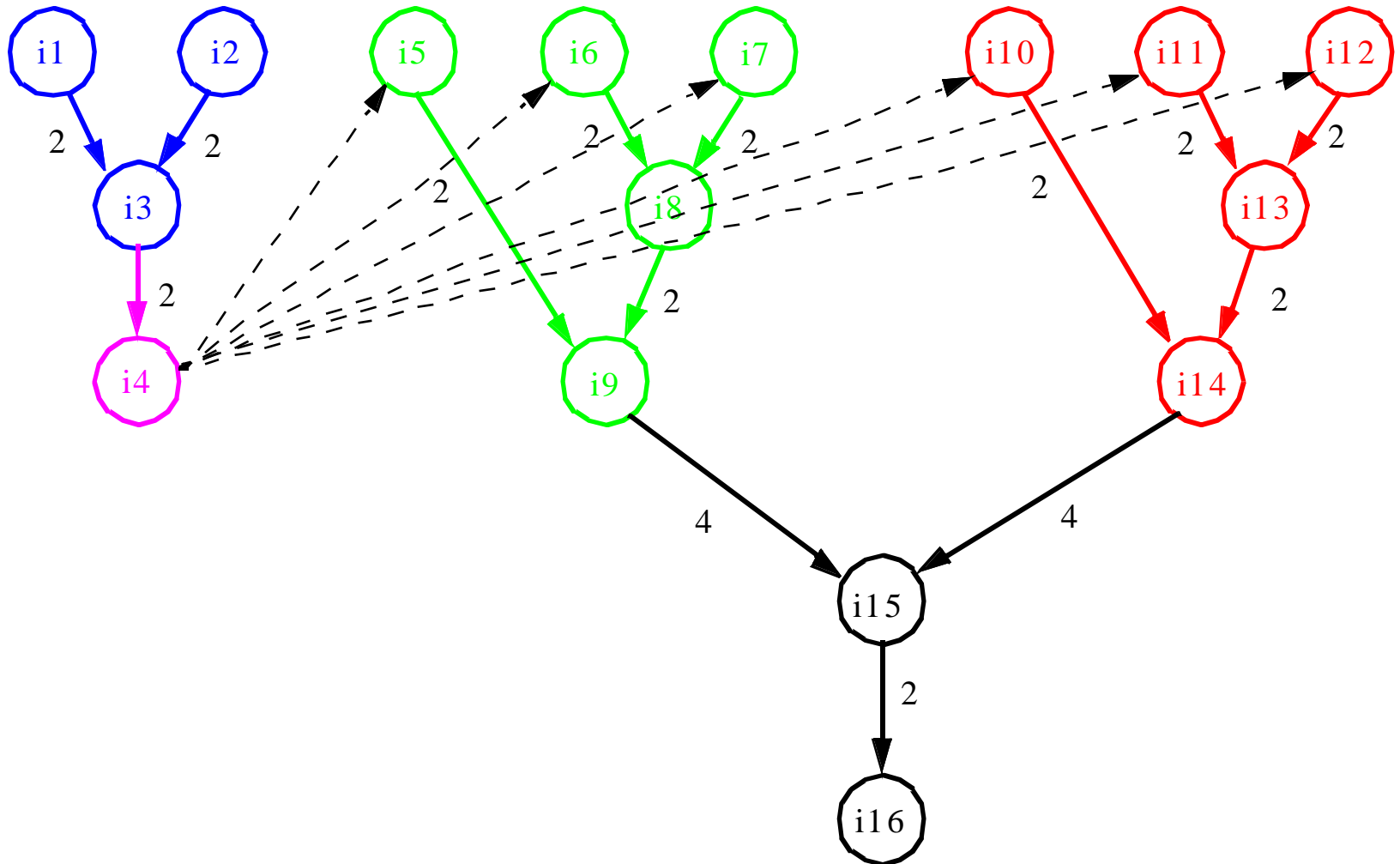
List Scheduling

- Assign priority to each instruction
- Initialize ready list that holds all ready instructions
    - Ready = data ready and can be scheduled
- Choose one ready instruction *I* from ready list with the highest priority
    - Possibly using tie-breaking heuristics
- Insert *I* into schedule
    - Making sure resource constraints are satisfied
- Add those instructions whose precedence constraints are now satisfied into the ready list

Instruction Prioritization Heuristics

- Number of descendants in precedence graph
- Maximum latency from root node of precedence graph
- Length of operation latency
- Ranking of paths based on importance
- Combination of above

VLIW List Scheduling

- Assign Priorities
- Compute Data Ready List (DRL) - all operations whose predecessors have been scheduled.
- Select from DRL in priority order while checking resource constraints
- Add newly ready operations to DRL and repeat for next instruction
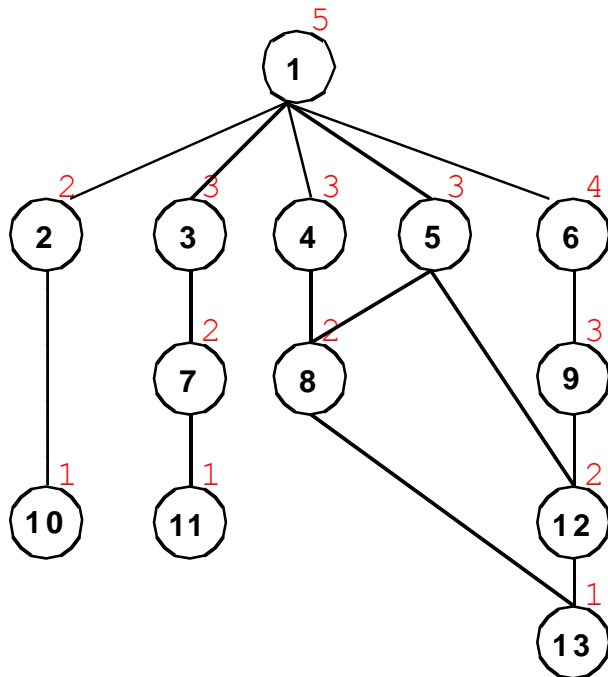
4-wide VLIW				Data Ready List
1				{1}
6	3	4	5	{2,3,4,5,6}
9	2	7	8	{2,7,8,9}
12	10	11		{10,11,12}
13				{13}

Trace Scheduling Example (I)

fdiv f1, f2, f3
fadd f4, f1, f5
beq r1, $0   B 1

990   10

B 2   ld r2, 0(r3)   ld r2, 4(r3)   B 3

990   10

add r2, r2, 4   B 4
beq r2, $0

800   200

B 5   fsub f2, f2, f6   fsub f2, f3, f7   B 6
st.d f2, 0(r8)

800   200

add r3, r3, 4   B 7
add r8, r8, 4

---

9 stalls —   **fdiv f1, f2, f3**
fadd f4, f1, f5
beq r1, $0

**r2 and f2
not live
out**

ld r2, 0(r3)

B 3

1 stall —   **add r2, r2, 4**
beq r2, $0

**f2 not
live out**

fsub f2, f2, f6
st.d f2, 0(r8)

1 stall —   **B 6**

add r3, r3, 4
add r8, r8, 4

Trace Scheduling Example (II)

```
fdiv  f1, f2, f3
beq  r1, $0


ld  r2, 0(r3)
fsub  f2, f2, f6
add  r2, r2, 4
beq  r2, $0


st.d  f2, 0(r8)



add  r3, r3, 4
add  r8, r8, 4
fadd  f4, f1, f5
```

0 stall

0 stall

1 stall

B 3

B 6

```
fdiv  f1, f2, f3
beq  r1, $0


ld  r2, 0(r3)
fsub  f2, f2, f6
add  r2, r2, 4
beq  r2, $0


st.d  f2, 0(r8)



add  r3, r3, 4
add  r8, r8, 4
fadd  f4, f1, f5
```

fadd f4, f1, f5

**Split
comp. code**

fadd f4, f1, f5

B 3

B 6

Trace Scheduling Example (III)

```
fdiv  f1, f2, f3
beq   r1, $0


ld  r2, 0(r3)
fsub  f2, f2, f6
add  r2, r2, 4
beq  r2, $0


st.d  f2, 0(r8)



add  r3, r3, 4
add  r8, r8, 4
fadd  f4, f1, f5
```

```
fadd  f4, f1, f5
```

**Split
comp. code**

```
fadd  f4, f1, f5
```

B3 **B6**

```
add  r3, r3, 4
add  r8, r8, 4
```

Join comp. code

Trace Scheduling Example (IV)

```
fdiv  f1, f2, f3
beq  r1, $0
```

```
fadd   f4, f1, f5
```

Split comp. code

```
ld  r2, 0(r3)
fsub f2, f2, f6
add  r2, r2, 4
beq  r2, $0
```

```
fadd   f4, f1, f5
```

```
st.d  f2, 0(r8)
```

B 3

```
add  r2, r2, 4
beq  r2, $0

fsub f2, f2, f6
st.d  f2, 0(r8)

add  r3, r3, 4
add  r8, r8, 4
```

Copied split instructions

```
add  r3, r3, 4
add  r8, r8, 4
fadd   f4, f1, f5
```

B 6

```
add  r3, r3, 4
add  r8, r8, 4
```

Join comp. code

Trace Scheduling Example (V)

Trace Scheduling Tradeoffs

- **Advantages**

  + Enables the finding of more independent instructions → fewer NOPs in a VLIW instruction
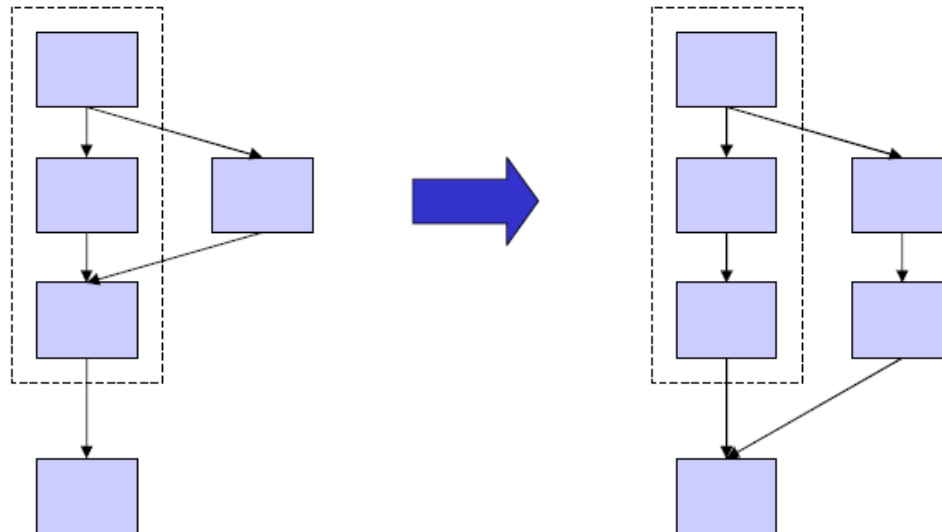
- **Disadvantages**

  -- Profile dependent

      -- What if dynamic path deviates from trace?

  -- Code bloat and additional fix-up code executed

      -- Due to side entrances and side exits

      -- Infrequent paths interfere with the frequent path

  -- Effectiveness depends on the bias of branches

      -- Unbiased branches → smaller traces → less opportunity for finding independent instructions
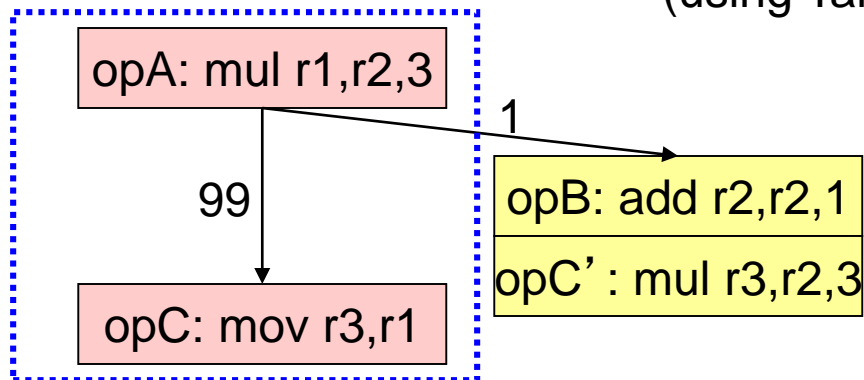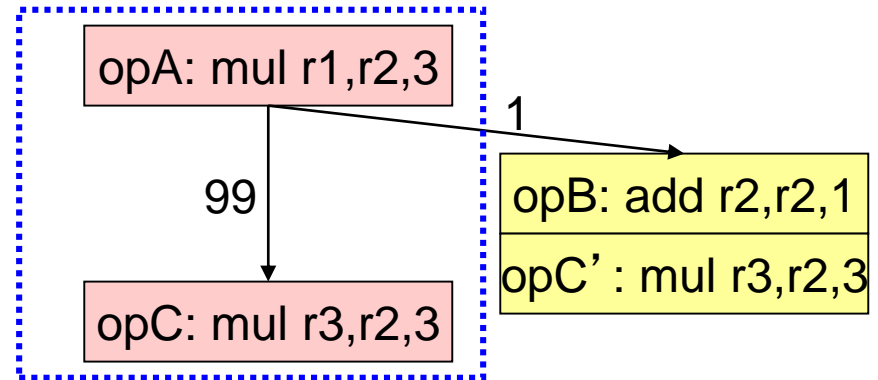
Superblock Scheduling

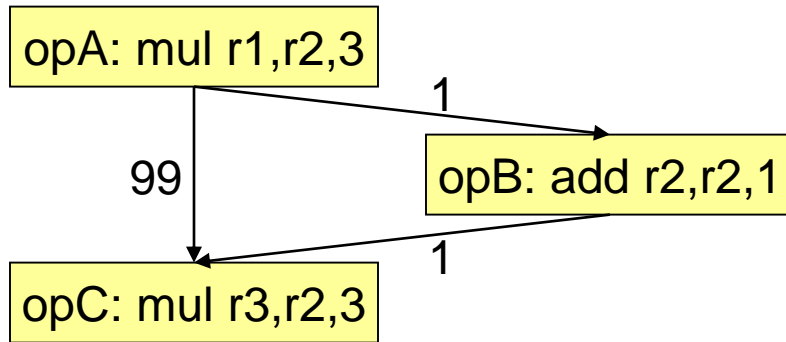- Trace: multiple entry, multiple exit block
- Superblock: single-entry, multiple exit block
  - A trace with side entrances are eliminated
  - Infrequent paths do not interfere with the frequent path
+ More optimization/scheduling opportunity than traces
+ Eliminates "difficult" bookkeeping due to side entrances

Hwu+, "The Superblock: An Effective Technique for VLIW and superscalar compilation," J of SC 1991.

Superblock Example

Could you have done this with a trace?

opA: mul r1,r2,3

99

1

opB: add r2,r2,1

1

opC: mul r3,r2,3

Original Code

---

opA: mul r1,r2,3

99

1

opB: add r2,r2,1

opC': mul r3,r2,3

opC: mul r3,r2,3

Code After Superblock Formation
(using Tail Duplication)

---

opA: mul r1,r2,3

99

1

opB: add r2,r2,1

opC': mul r3,r2,3

opC: mov r3,r1

Code After Common
Subexpression Elimination

Superblock Scheduling Shortcomings

-- Still profile-dependent

-- No single frequently executed path if there is an unbiased branch

    -- Reduces the size of superblocks

-- Code bloat and additional fix-up code executed

    -- Due to side exits

Hyperblock Scheduling

- **Idea:** Use predication support to eliminate unbiased branches and increase the size of superblocks

- **Hyperblock:** A single-entry, multiple-exit block with internal control flow eliminated using predication (if-conversion)

- **Advantages**

  + Reduces the effect of unbiased branches on scheduling block size

- **Disadvantages**

  -- Requires predicated execution support

  -- All disadvantages of predicated execution

Hyperblock Formation (I)

- Hyperblock formation
  1. Block selection
  2. Tail duplication
  3. If-conversion

- Block selection
  - Select subset of BBs for inclusion in HB
  - Difficult problem
  - Weighted cost/benefit function
    - Height overhead
    - Resource overhead
    - Dependency overhead
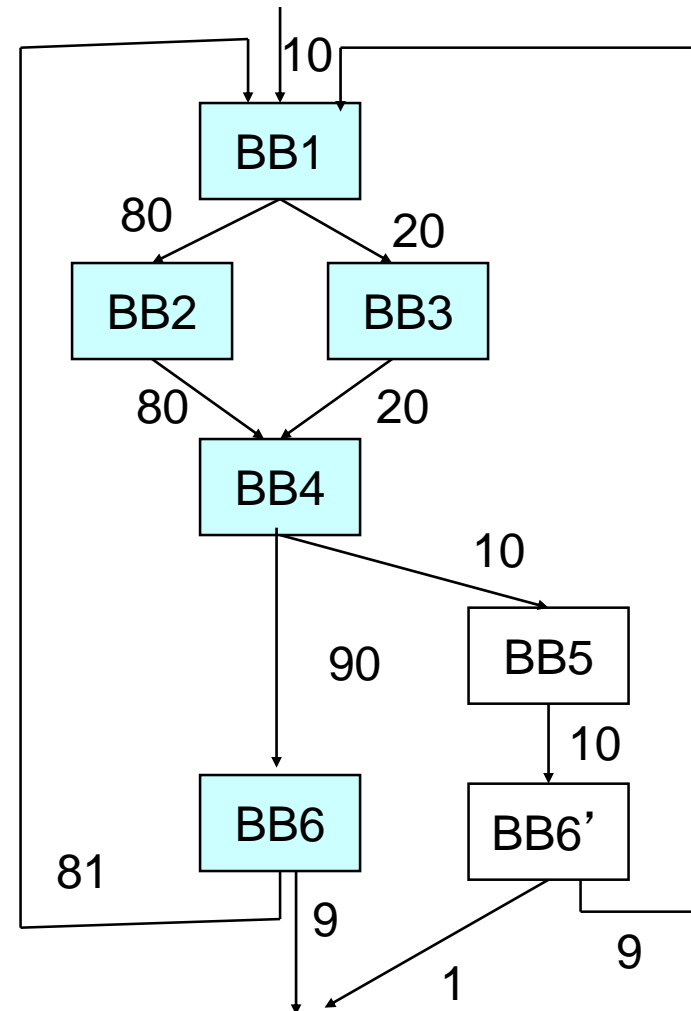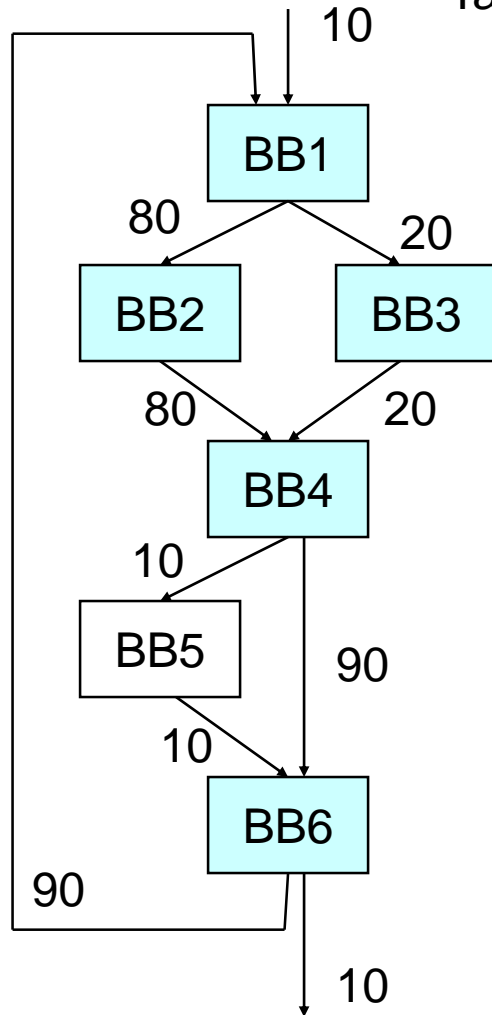   - Branch elimination benefit
    - Weighted by frequency
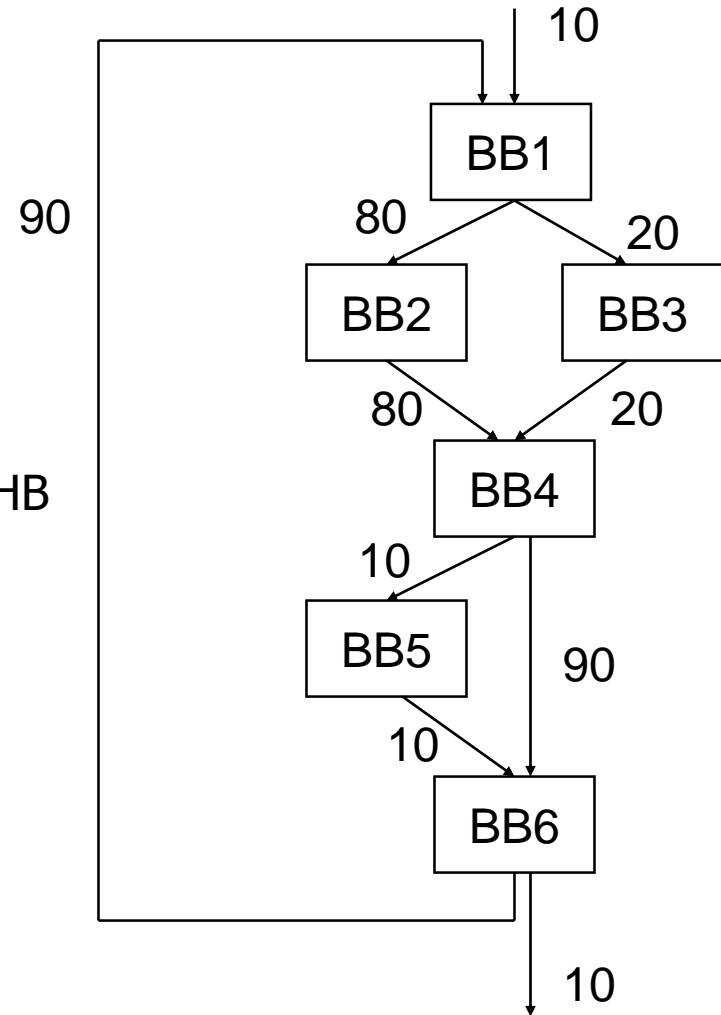
- Mahlke et al., "Effective Compiler Support for Predicated Execution Using the Hyperblock," MICRO 1992.
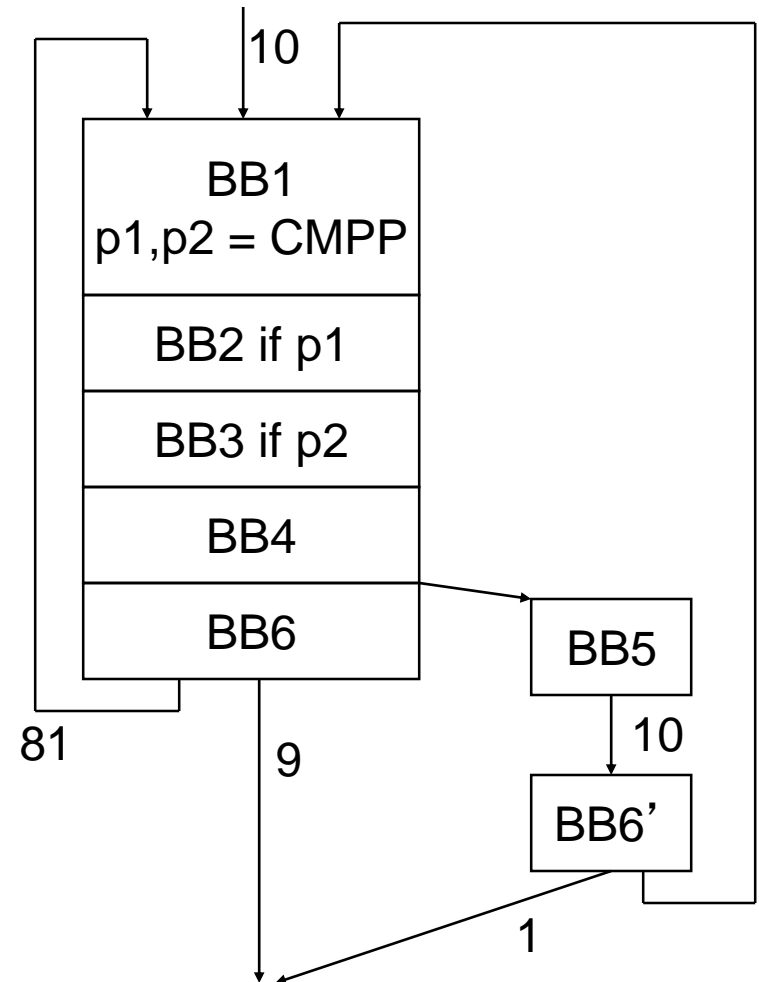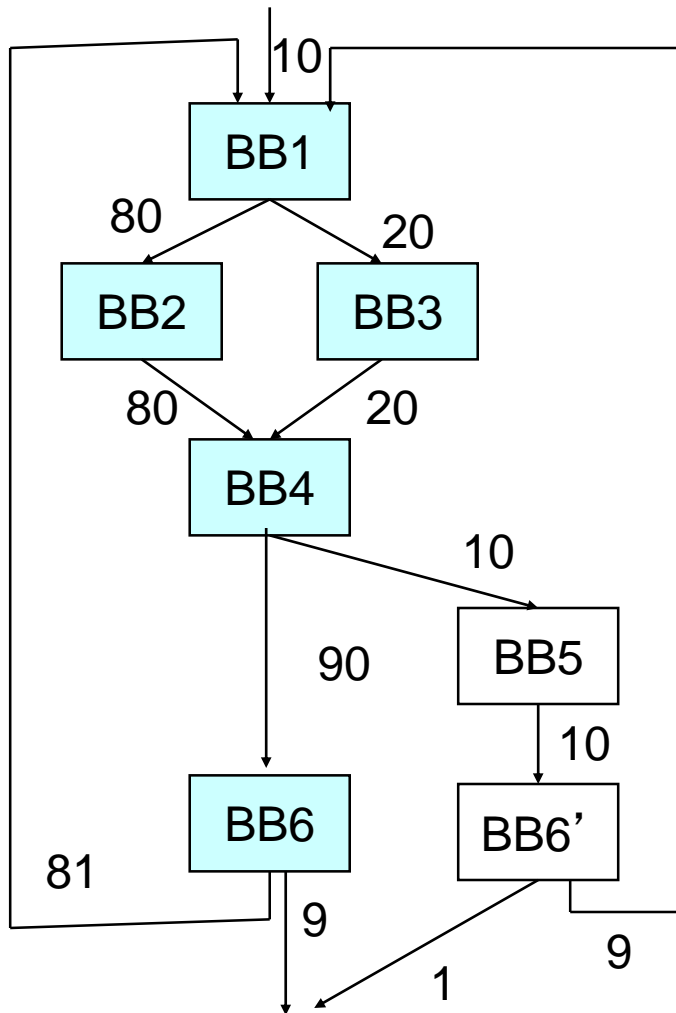
Hyperblock Formation (II)

Tail duplication same as with Superblock formation

Hyperblock Formation (III)

If-convert (predicate) intra-hyperblock branches

Aside: Test of Time

- Mahlke et al., "Effective Compiler Support for Predicated Execution Using the Hyperblock," MICRO 1992.

- MICRO Test of Time Award
    - http://www.cs.cmu.edu/~yixinluo/new_home/2014-Micro-ToT.html

Can We Do Better?

- Hyperblock still has disadvantages
  - Profile dependent (Optimizes for a single path)
  - Requires fix-up code
  - And, it requires predication support

- Can we do even better?

- Solution: Single-entry, single-exit enlarged blocks
  - Block-structured ISA: atomic enlarged blocks

Block Structured ISA

- Blocks (> instructions) are atomic (all-or-none) operations
  - Either all of the block is committed or none of it
- Compiler enlarges blocks by combining basic blocks with their control flow successors
  - Branches within the enlarged block converted to "fault" operations → if the fault operation evaluates to true, the block is discarded and the target of fault is fetched

Melvin and Patt, "Enhancing Instruction Scheduling with a Block-Structured ISA," IJPP 1995.

69
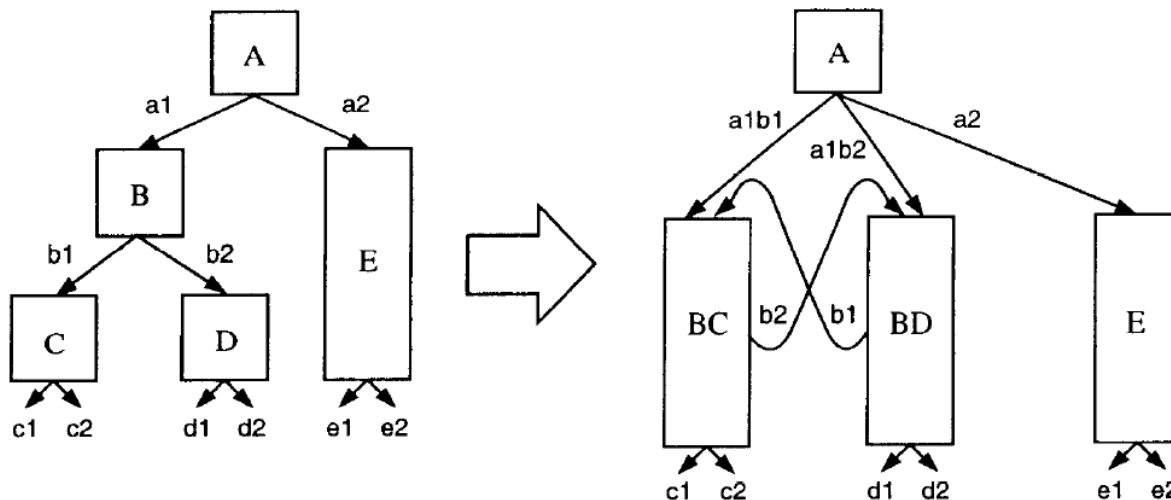
Block Structured ISA (II)

- Advantages
    - + Large atomic blocks
        - → Aggressive compiler optimizations (e.g. reordering) can be enabled within atomic blocks (no side entries or exits)
        - → Larger units can be fetched from I-cache → wide fetch
    - + Can dynamically predict which optimized atomic block is executed using a "branch predictor"
        - → can optimize multiple "hot" paths
    - + No compensation (fix-up) code

- Disadvantages
    - -- "Fault operations" can lead wasted work (atomicity)
    - -- Code bloat (multiple copies of the same basic block exists in the binary and possibly in I-cache)
    - -- Need to predict which enlarged block comes next

Block Structured ISA (III)

- Hao et al., "Increasing the instruction fetch rate via block-structured instruction set architectures," MICRO 1996.
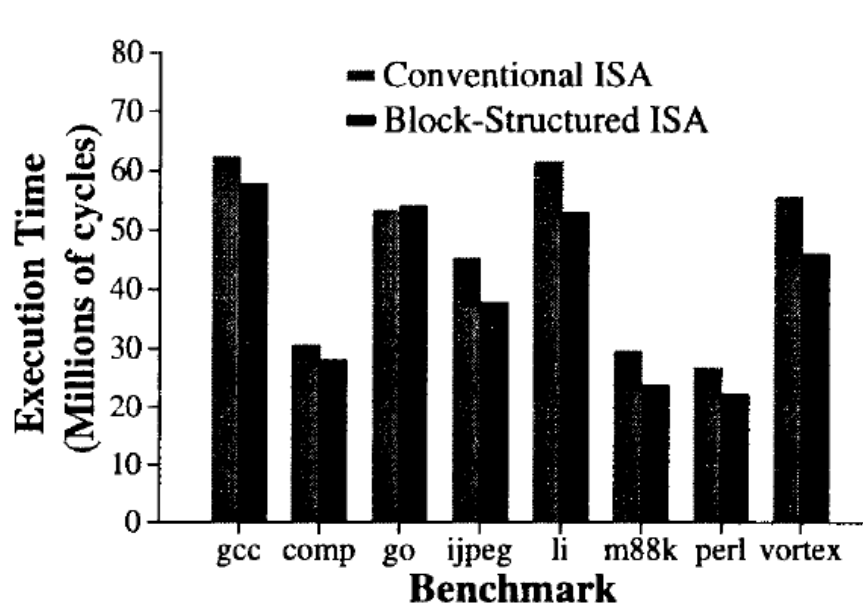

Figure 3. Performance comparison of block-structured ISA executables and conventional ISA executables.
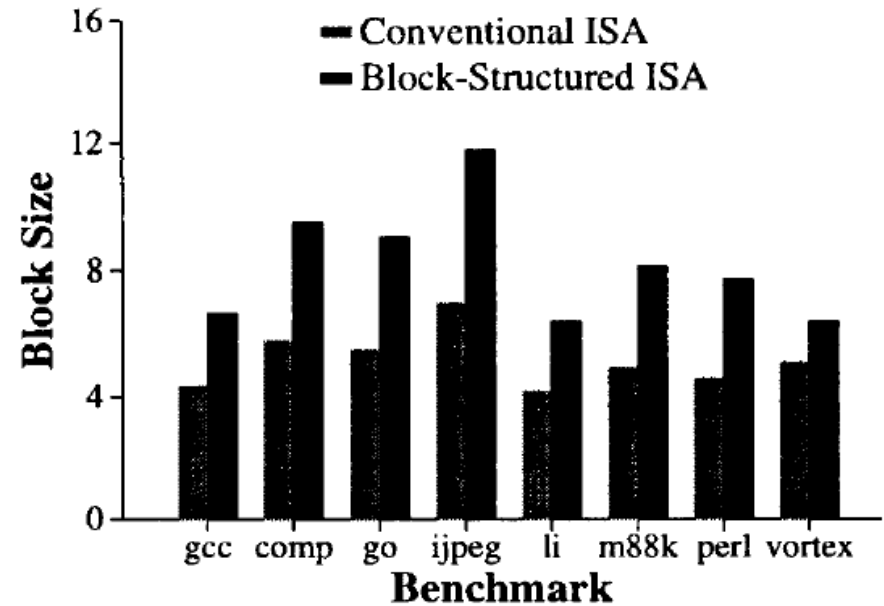
Figure 5. Average block sizes for block-structured and conventional ISA executables.

Superblock vs. BS-ISA

- Superblock
  - Single-entry, multiple exit code block
  - Not atomic
  - Compiler inserts fix-up code on superblock side exit
  - Only one path optimized (hardware has no choice to pick dynamically)

- BS-ISA blocks
  - Single-entry, single exit
  - Atomic
  - Need to roll back to the beginning of the block on fault
  - Multiple paths optimized (hardware has a choice to pick)

Superblock vs. BS-ISA

- **Superblock**
  - \+ No ISA support needed
  - -- Optimizes for only 1 frequently executed path
    - -- Not good if dynamic path deviates from profiled path → missed opportunity to optimize another path

- **Block Structured ISA**
  - \+ Enables optimization of multiple paths and their dynamic selection.
  - \+ Dynamic prediction to choose the next enlarged block. Can dynamically adapt to changes in frequently executed paths at run-time
  - \+ Atomicity can enable more aggressive code optimization
  - -- Code bloat becomes severe as more blocks are combined
  - -- Requires "next enlarged block" prediction, ISA+HW support
  - -- More wasted work on "fault" due to atomicity requirement

Summary: Larger Code Blocks

Summary and Questions

- Trace, superblock, hyperblock, block-structured ISA

- How many entries, how many exits does each of them have?
  - What are the corresponding benefits and downsides?

- What are the common benefits?
  - Enable and enlarge the scope of code optimizations
  - Reduce fetch breaks; increase fetch rate

- What are the common downsides?
  - Code bloat (code size increase)
  - Wasted work if control flow deviates from enlarged block's path

We did not cover the following slides in lecture. These are for your preparation for the next lecture.

IA-64: A "Complicated" VLIW ISA

Recommended reading:

Huck et al., "Introducing the IA-64 Architecture," IEEE Micro 2000.

EPIC – Intel IA-64 Architecture

- Gets rid of lock-step execution of instructions within a VLIW instruction

- Idea: More ISA support for static scheduling and parallelization
  - Specify dependencies within and between VLIW instructions (explicitly parallel)

+ No lock-step execution

+ Static reordering of stores and loads + dynamic checking

-- Hardware needs to perform dependency checking (albeit aided by software)

-- Other disadvantages of VLIW still exist

- Huck et al., "Introducing the IA-64 Architecture," IEEE Micro, Sep/Oct 2000.

IA-64 Instructions

- ## IA-64 "Bundle" (~EPIC Instruction)
  - ❑ Total of 128 bits
  - ❑ Contains three IA-64 instructions
  - ❑ Template bits in each bundle specify dependencies within a bundle

Instruction 2	Instruction 1	Instruction 0	Template
41 bits	41 bits	41 bits	5 bits

Op	Register 1	Register 2	Register 3	Predicate
14 bits	7 bits	7 bits	7 bits	6 bits
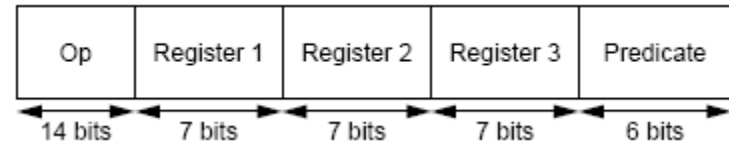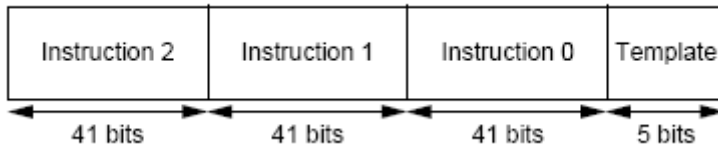
- ## IA-64 Instruction
  - ❑ Fixed-length 41 bits long
  - ❑ Contains three 7-bit register specifiers
  - ❑ Contains a 6-bit field for specifying one of the 64 one-bit predicate registers

IA-64 Instruction Bundles and Groups

```
{ .mii
    add r1 = r2, r3
    sub r4 = r4, r5 ;;
    shr r7 = r4, r12 ;;
}
{ .mmi
    ld8 r2 = [r1] ;;
    st8 [r1] = r23
    tbit p1,p2=r4,5
}
{ .mbb
    ld8 r45 = [r55]
(p3)br.call b1=func1
(p4)br.cond Label1
}
{ .mfi
    st4 [r45]=r6
    fmac f1=f2,f3
    add  r3=r3,8 ;;
}
```

- **Groups of instructions can be executed safely in parallel**
  - Marked by "stop bits"

- **Bundles are for packaging**
  - Groups can span multiple bundles
   - Alleviates recompilation need somewhat

Template Bits

- **Specify two things**
  - Stop information: Boundary of independent instructions
  - Functional unit information: Where should each instruction be routed

Template	Slot 0	Slot 1	Slot 2
00	M-unit	I-unit	I-unit
01	M-unit	I-unit	I-unit
02	M-unit	I-unit	I-unit
03	M-unit	I-unit	I-unit
04	M-unit	L-unit	X-unit[a]
05	M-unit	L-unit	X-unit[a]
06			
07			
08	M-unit	M-unit	I-unit
09	M-unit	M-unit	I-unit
0A	M-unit	M-unit	I-unit
0B	M-unit	M-unit	I-unit
0C	M-unit	F-unit	I-unit
0D	M-unit	F-unit	I-unit
0E	M-unit	M-unit	F-unit
0F	M-unit	M-unit	F-unit
10	M-unit	I-unit	B-unit
11	M-unit	I-unit	B-unit
12	M-unit	B-unit	B-unit

Template	Slot 0	Slot 1	Slot 2
13	M-unit	B-unit	B-unit
14			
15			
16	B-unit	B-unit	B-unit
17	B-unit	B-unit	B-unit
18	M-unit	M-unit	B-unit
19	M-unit	M-unit	B-unit
1A			
1B			
1C	M-unit	F-unit	B-unit
1D	M-unit	F-unit	B-unit
1E			
1F			

Non-Faulting Loads and Exception Propagation
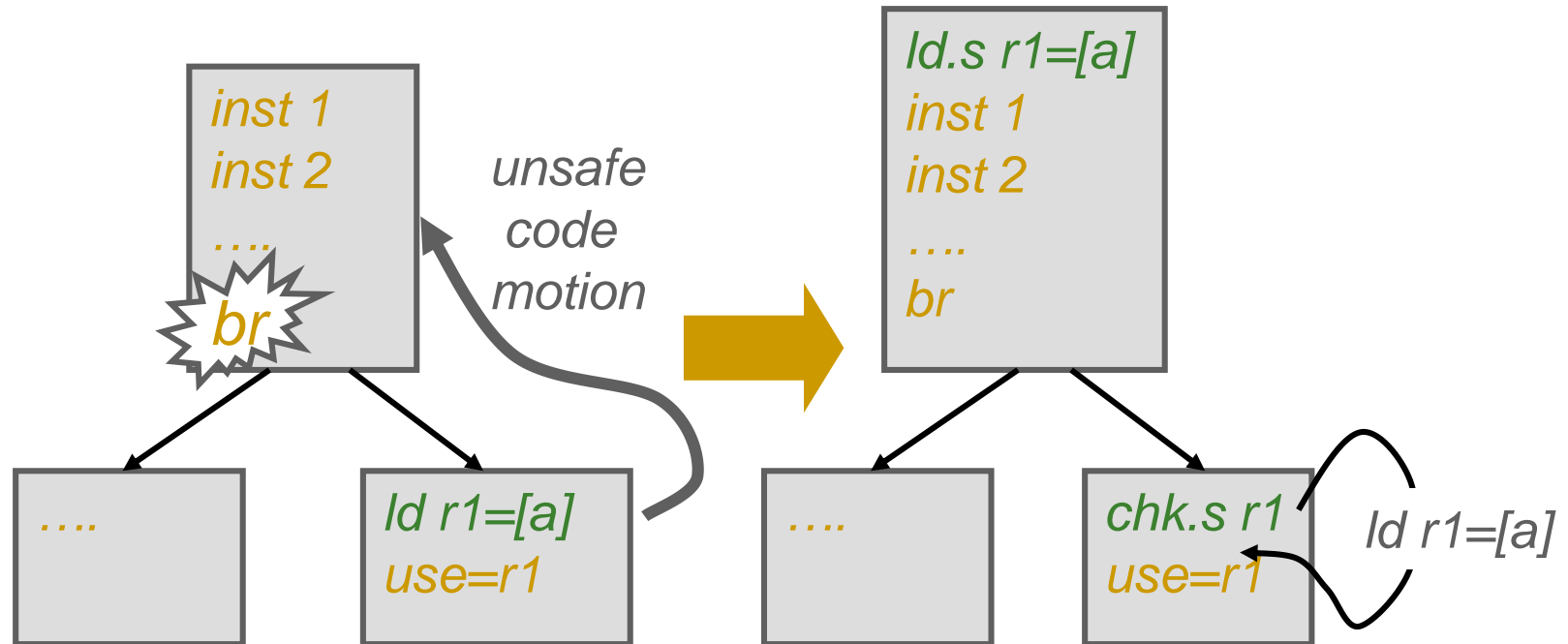
- *ld.s* (speculative load) fetches *speculatively* from memory

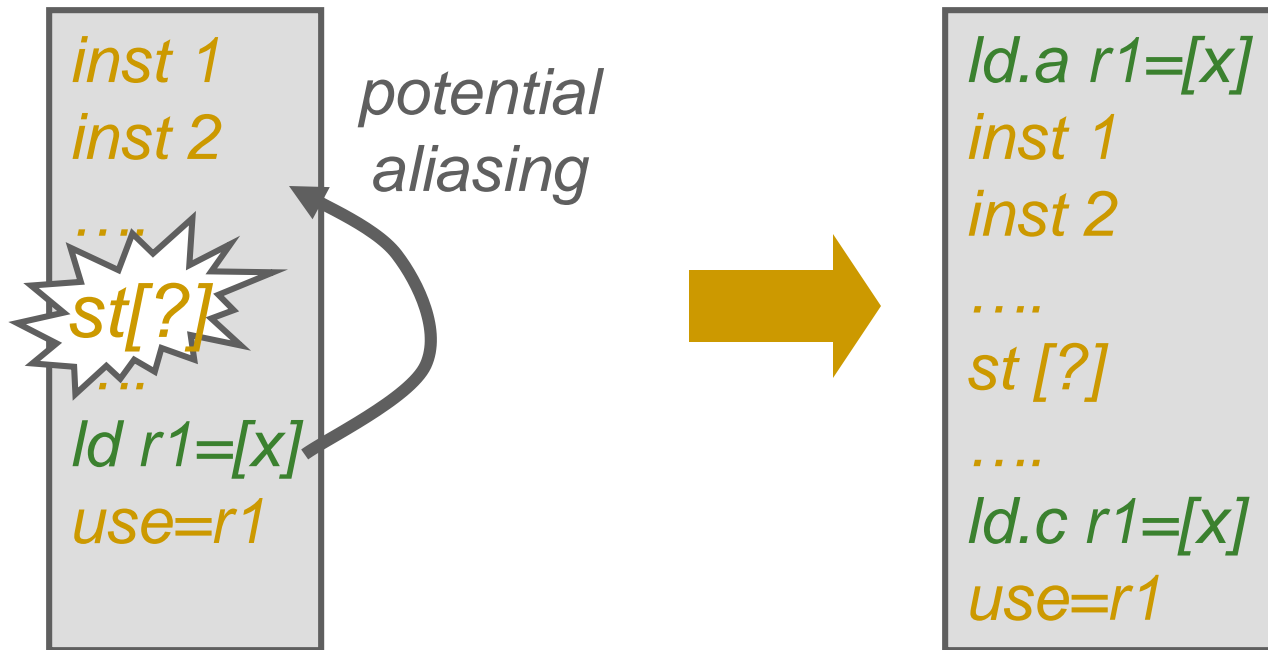   i.e. any exception due to *ld.s* is suppressed

- If *ld.s r1* did not cause an exception then *chk.s r1* is a NOP, else a branch is taken (to execute some compensation code)

Non-Faulting Loads and Exception Propagation in IA-64


```
inst 1
inst 2
….
br
```

```
….
```

```
ld r1=[a]
use=r1
```

*unsafe
code
motion*

```
ld.s r1=[a]
inst 1
inst 2
use=r1
….
br
```

```
….
```

```
chk.s use
```

```
ld r1=[a]
use=r1
```
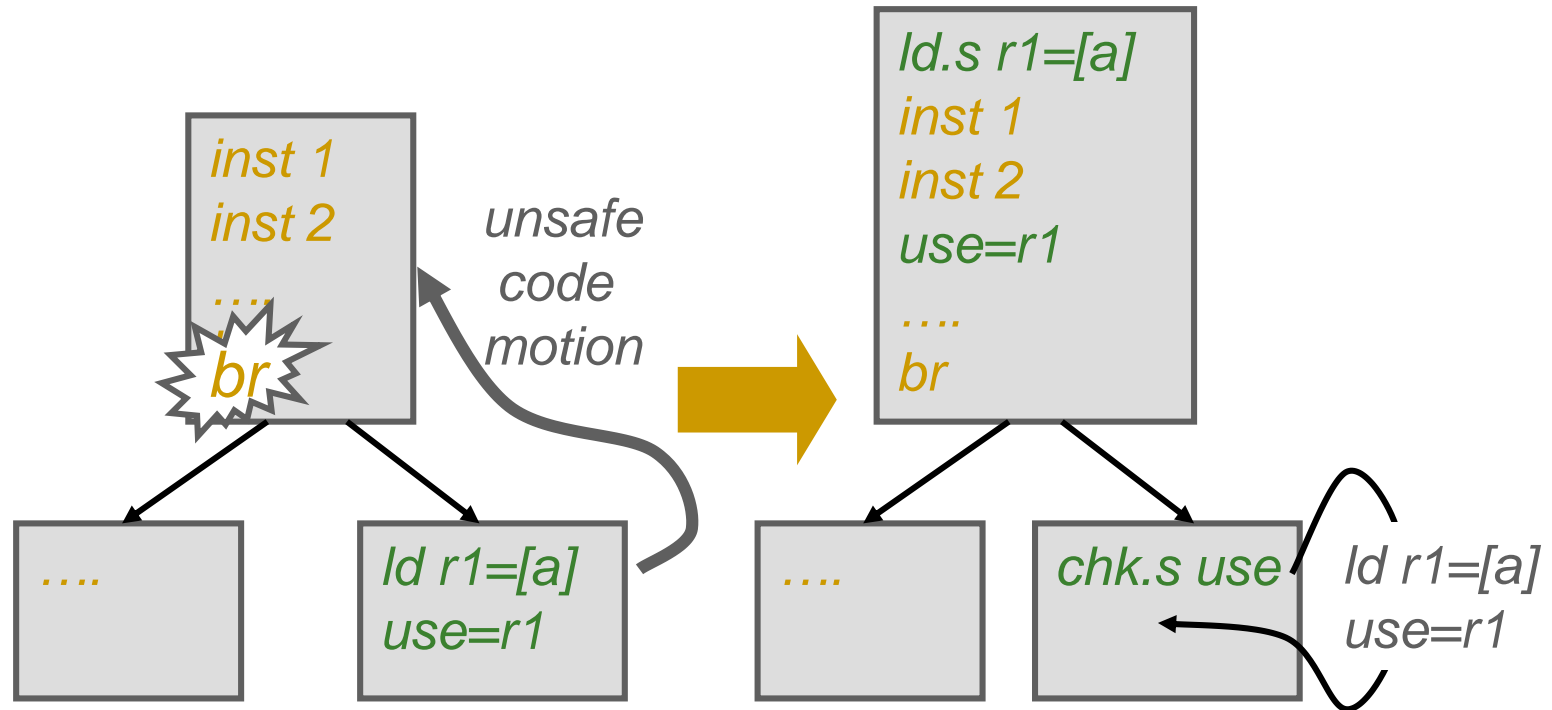
- Load data can be speculatively consumed prior to check
- "speculation" status is propagated with speculated data
- Any instruction that uses a speculative result also becomes speculative itself (i.e. suppressed exceptions)
- *chk.s* checks the entire dataflow sequence for exceptions

Aggressive ST-LD Reordering in IA-64

inst 1
inst 2
….
st[?]
…
ld r1=[x]
use=r1

potential aliasing

➡️

ld.a r1=[x]
inst 1
inst 2
….
st [?]
….
ld.c r1=[x]
use=r1

- *ld.a* (advanced load) starts the monitoring of any store to the same address as the advanced load
- If no aliasing has occurred since *ld.a*, *ld.c* is a NOP
- If aliasing has occurred, *ld.c* re-loads from memory

Aggressive ST-LD Reordering in IA-64

inst 1
inst 2
....
st[?]
...
ld r1=[x]
use=r1

potential aliasing

ld.a r1=[x]
inst 1
inst 2
use=r1

....

st [?]

....

chk.a X

....

ld r1=[a]
use=r1