

18-447

# Computer Architecture

## Lecture 18: Static Instruction Scheduling

Prof. Onur Mutlu

Carnegie Mellon University

Spring 2014, 2/28/2014

# A Note on This Lecture

---

- These slides are partly from 18-447 Spring 2013, Computer Architecture, Lecture 21: Static Instruction Scheduling
- Video of that lecture:
  - <http://www.youtube.com/watch?v=XdDUn2WtkRg>
  - Start from 16:26 and watch it until the end

# Last Lecture

---

- GPUs
- VLIW
- Decoupled Access Execute
- Systolic Arrays

# Review: Systolic Architectures

- Basic principle: Replace a single PE with a regular array of PEs and carefully orchestrate flow of data between the PEs → achieve high throughput w/o increasing memory bandwidth requirements

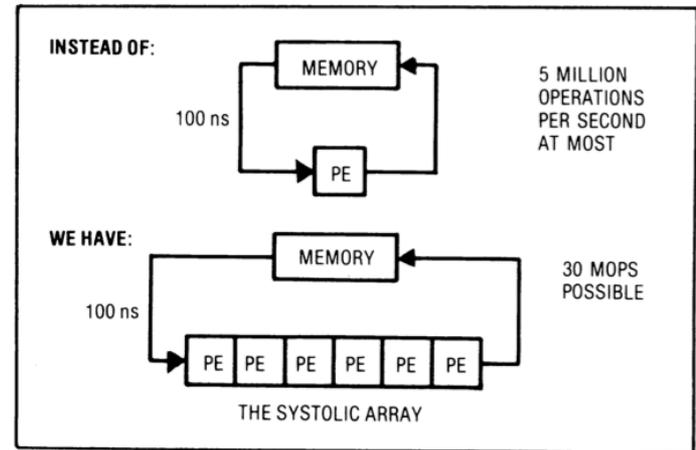
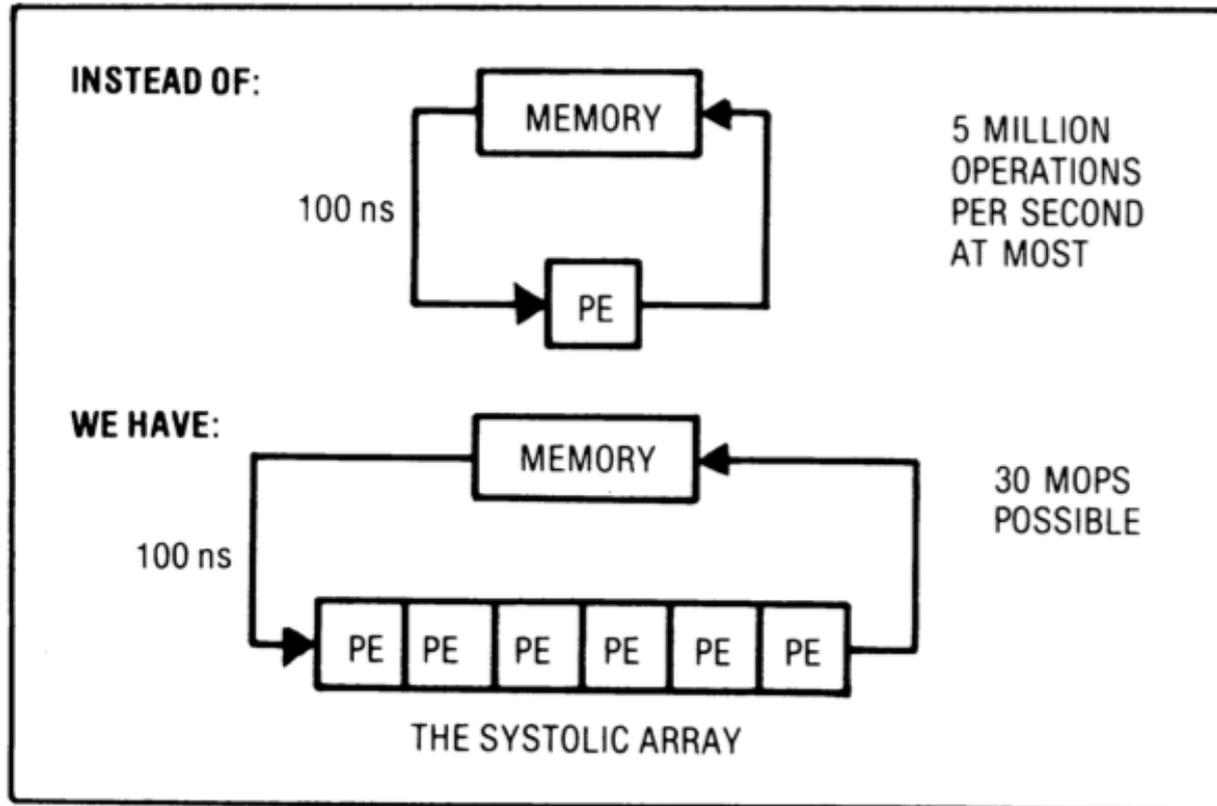


Figure 1. Basic principle of a systolic system.

- Differences from pipelining:
  - ❑ 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)

# Review: Systolic Architectures

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



Memory: heart  
PEs: cells

Memory pulses  
data through  
cells

Figure 1. Basic principle of a systolic system.

# Pipeline Parallel Programming Model

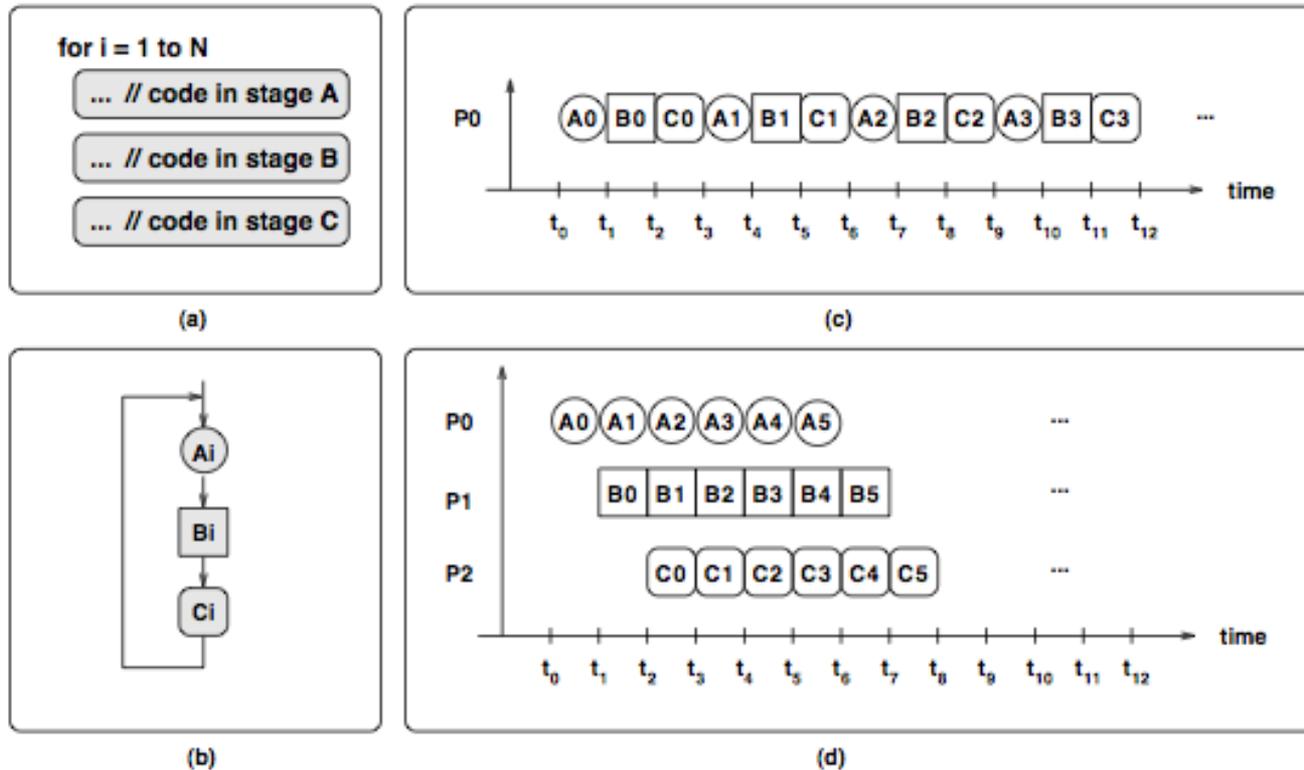
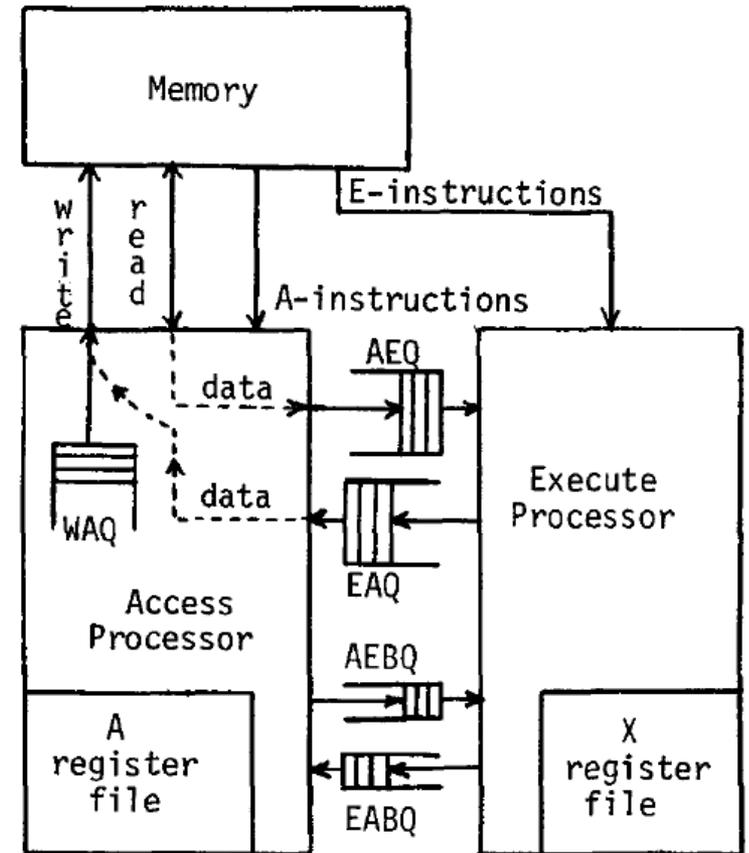


Figure 1. (a) The code of a loop, (b) Each iteration is split into 3 pipeline stages: A, B, and C. Iteration  $i$  comprises  $A_i$ ,  $B_i$ ,  $C_i$ . (c) Sequential execution of 4 iterations. (d) Parallel execution of 6 iterations using pipeline parallelism on a three-core machine. Each stage executes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

# Review: Decoupled Access/Execute

- Motivation: Tomasulo's algorithm too complex to implement
  - 1980s before HPS, Pentium Pro
- Idea: Decouple operand access and execution via two separate instruction streams that communicate via ISA-visible queues.
- Smith, "Decoupled Access/Execute Computer Architectures," ISCA 1982, ACM TOCS 1984.



# Review: Decoupled Access/Execute

---

## ■ Advantages:

- + Execute stream can run ahead of the access stream and vice versa
  - + If A takes a cache miss, E can perform useful work
  - + If A hits in cache, it supplies data to lagging E
  - + Queues reduce the number of required registers
- + Limited out-of-order execution without wakeup/select complexity

## ■ Disadvantages:

- Compiler support to partition the program and manage queues
  - Determines the amount of decoupling
- Branch instructions require synchronization between A and E
- Multiple instruction streams (can be done with a single one, though)

# Today

---

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

# Static Instruction Scheduling

(with a Slight Focus on VLIW)

# 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**

---

# 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

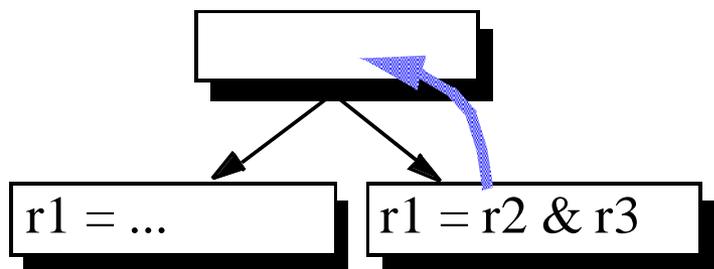
# VLIW: Finding Independent Operations

---

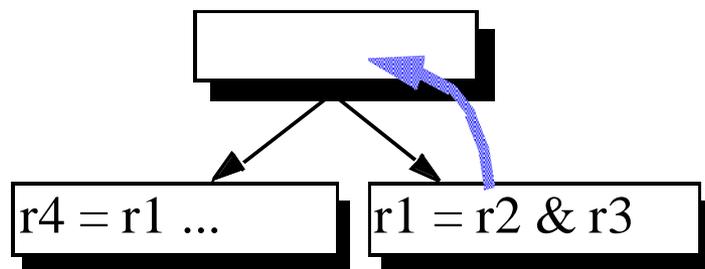
- Within a basic block, there is limited instruction-level parallelism
- To find multiple instructions to be executed in parallel, the compiler needs to consider multiple basic blocks
- Problem: Moving an instruction above a branch is unsafe because instruction is not guaranteed to be executed
- Idea: Enlarge blocks at compile time by finding the frequently-executed paths
  - Trace scheduling
  - Superblock scheduling
  - Hyperblock scheduling

# Safety and Legality in Code Motion

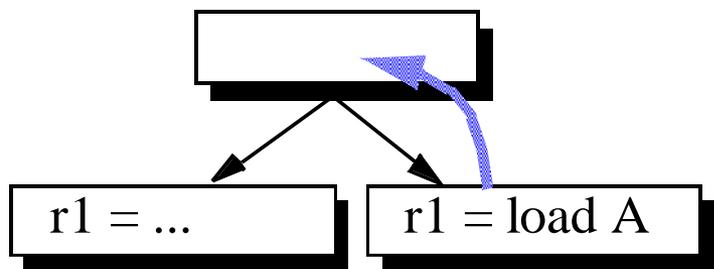
- Two characteristics of speculative code motion:
  - Safety: whether or not spurious exceptions may occur
  - Legality: whether or not result will be always correct
- Four possible types of code motion:



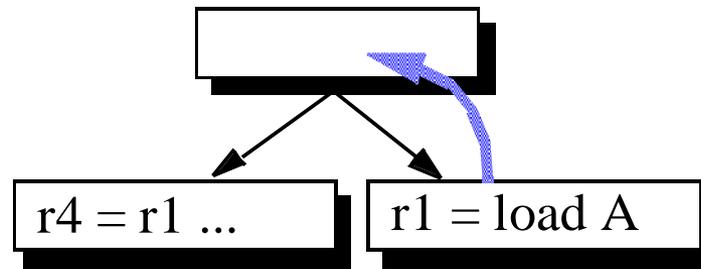
**(a) safe and legal**



**(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)

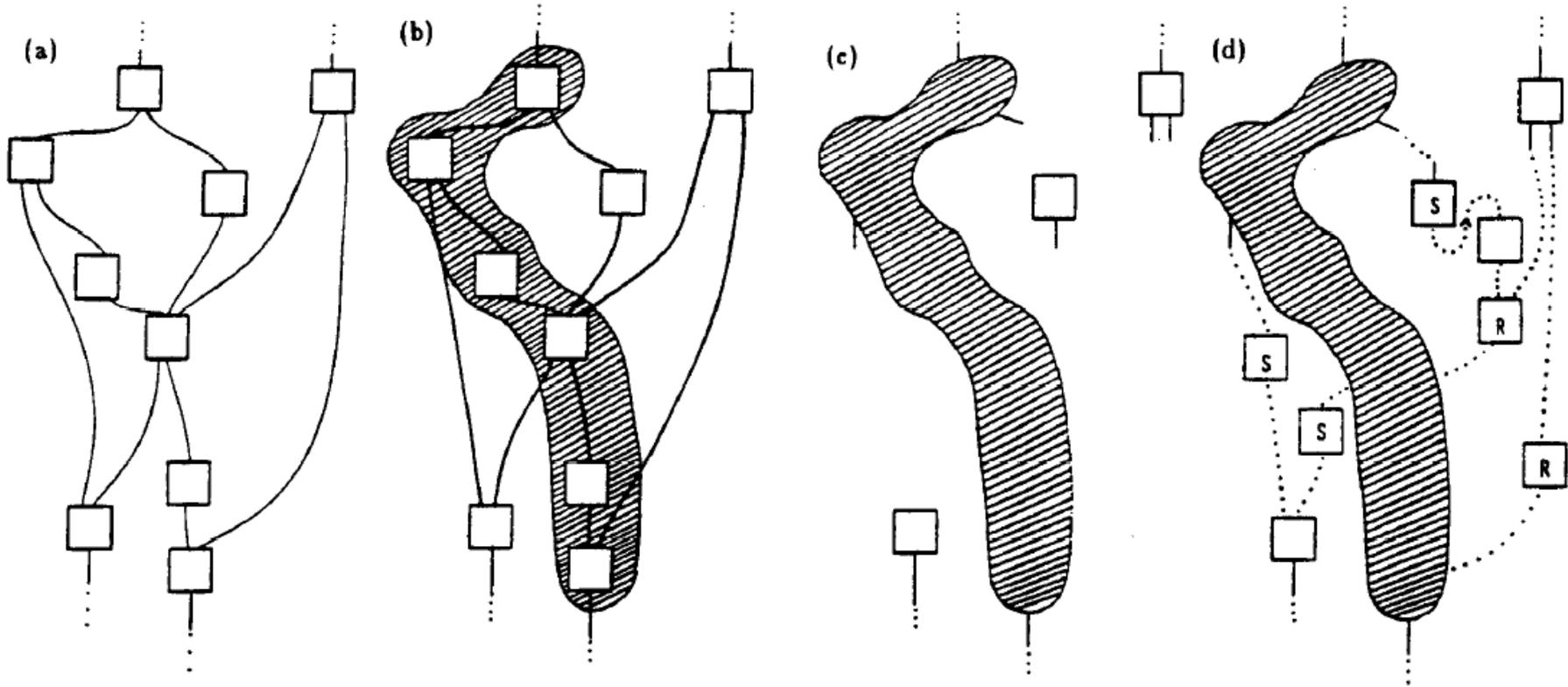
# Trace Scheduling (II)

---

- 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

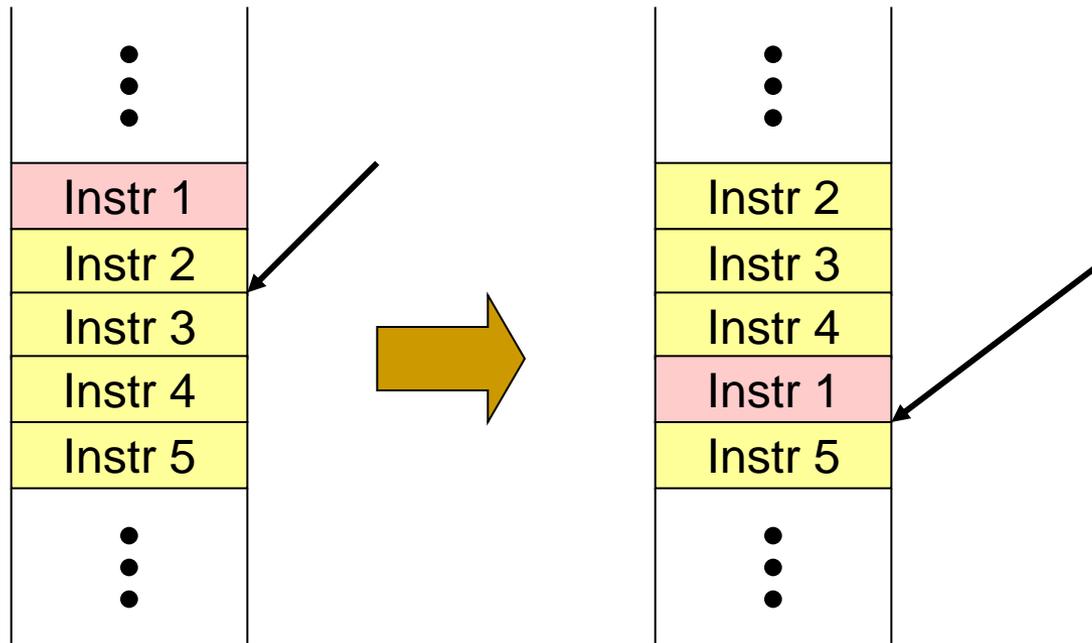
---



TRACE SCHEDULING LOOP-FREE CODE

# Trace Scheduling (III)

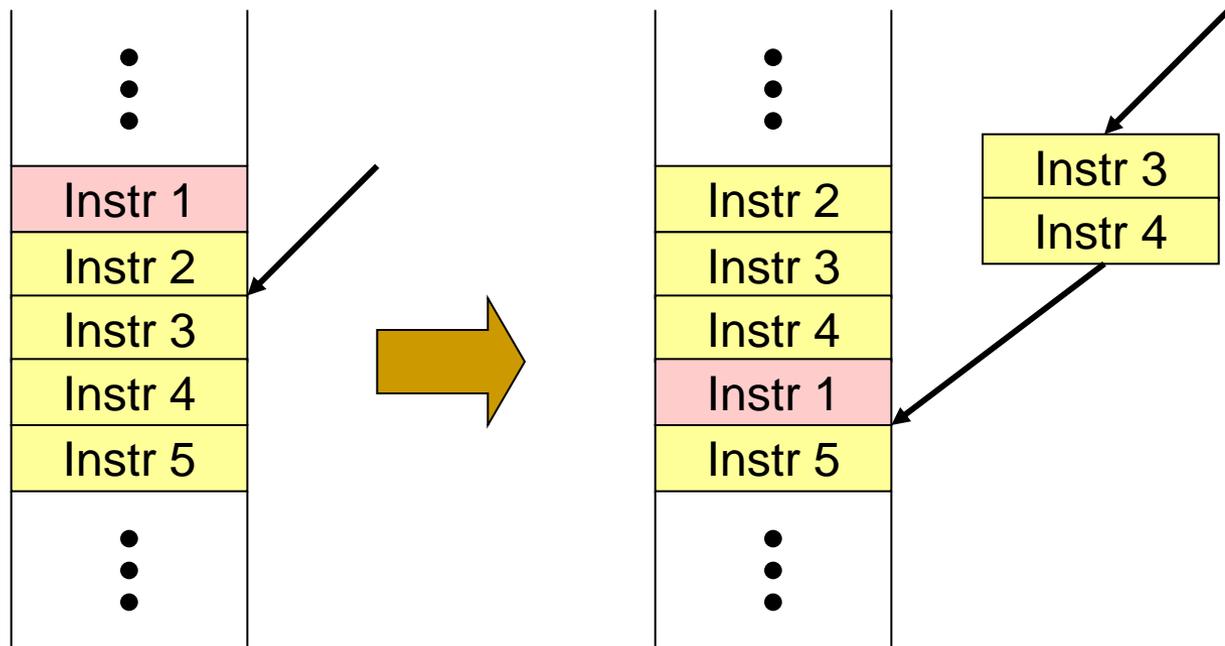
---



What bookkeeping is required when **Instr 1** is moved below the side entrance in the trace?

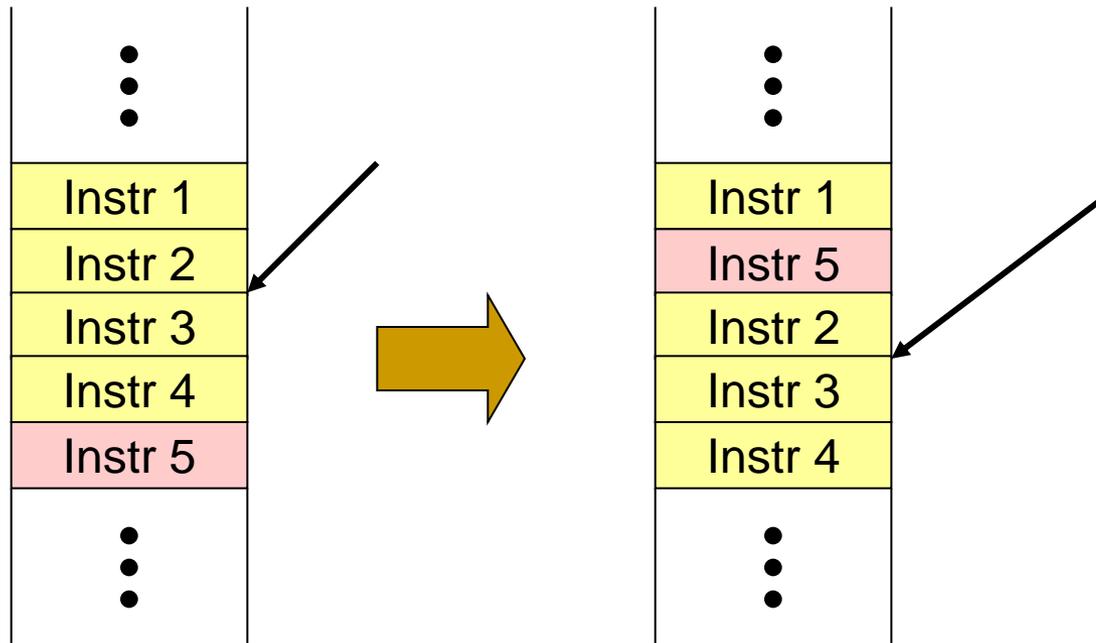
# Trace Scheduling (IV)

---



# Trace Scheduling (V)

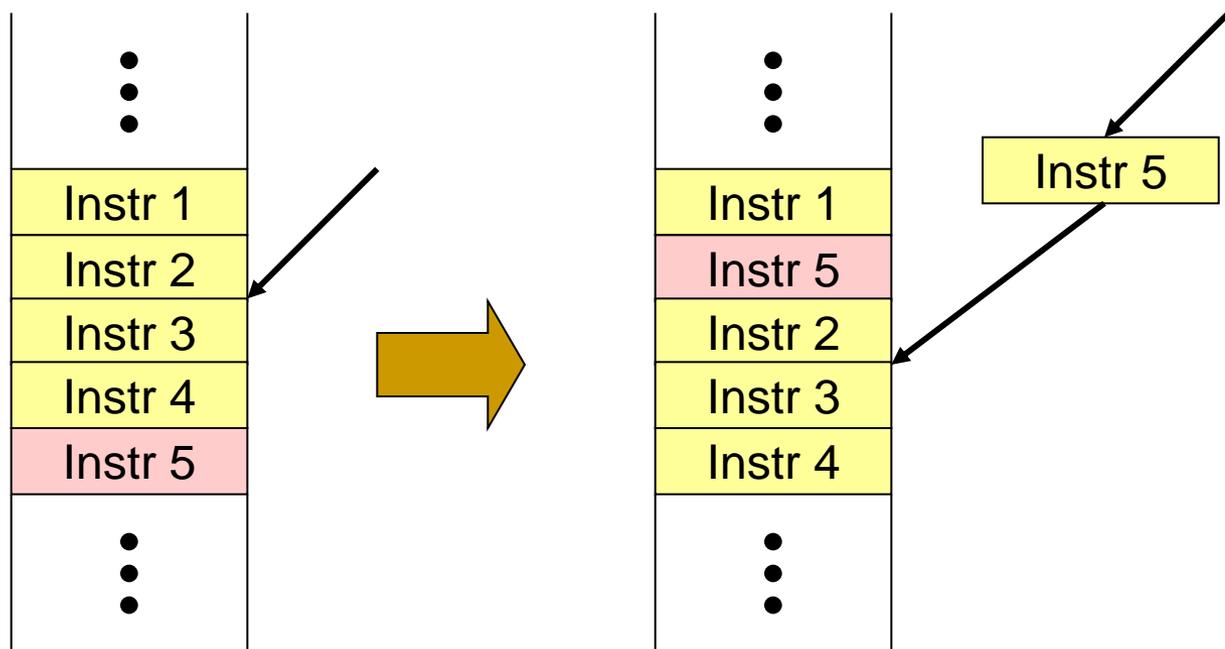
---



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

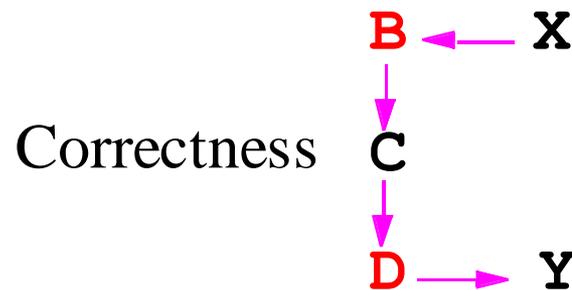
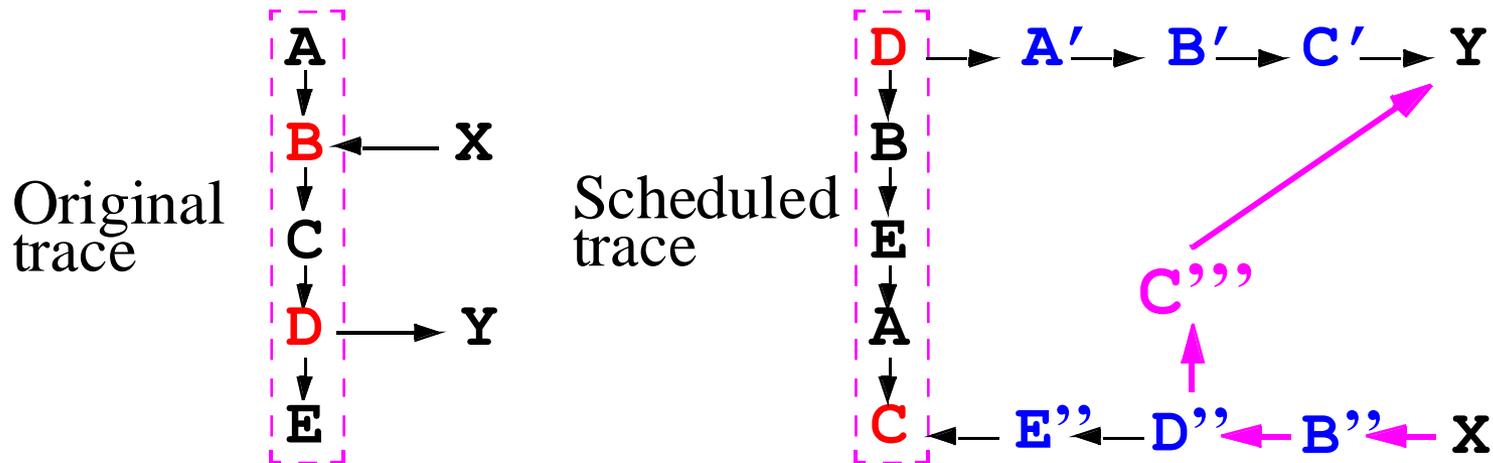
# Trace Scheduling (VI)

---



# Trace Scheduling Fixup Code Issues

- Sometimes need to copy instructions more than once to ensure correctness on all paths (see C below)



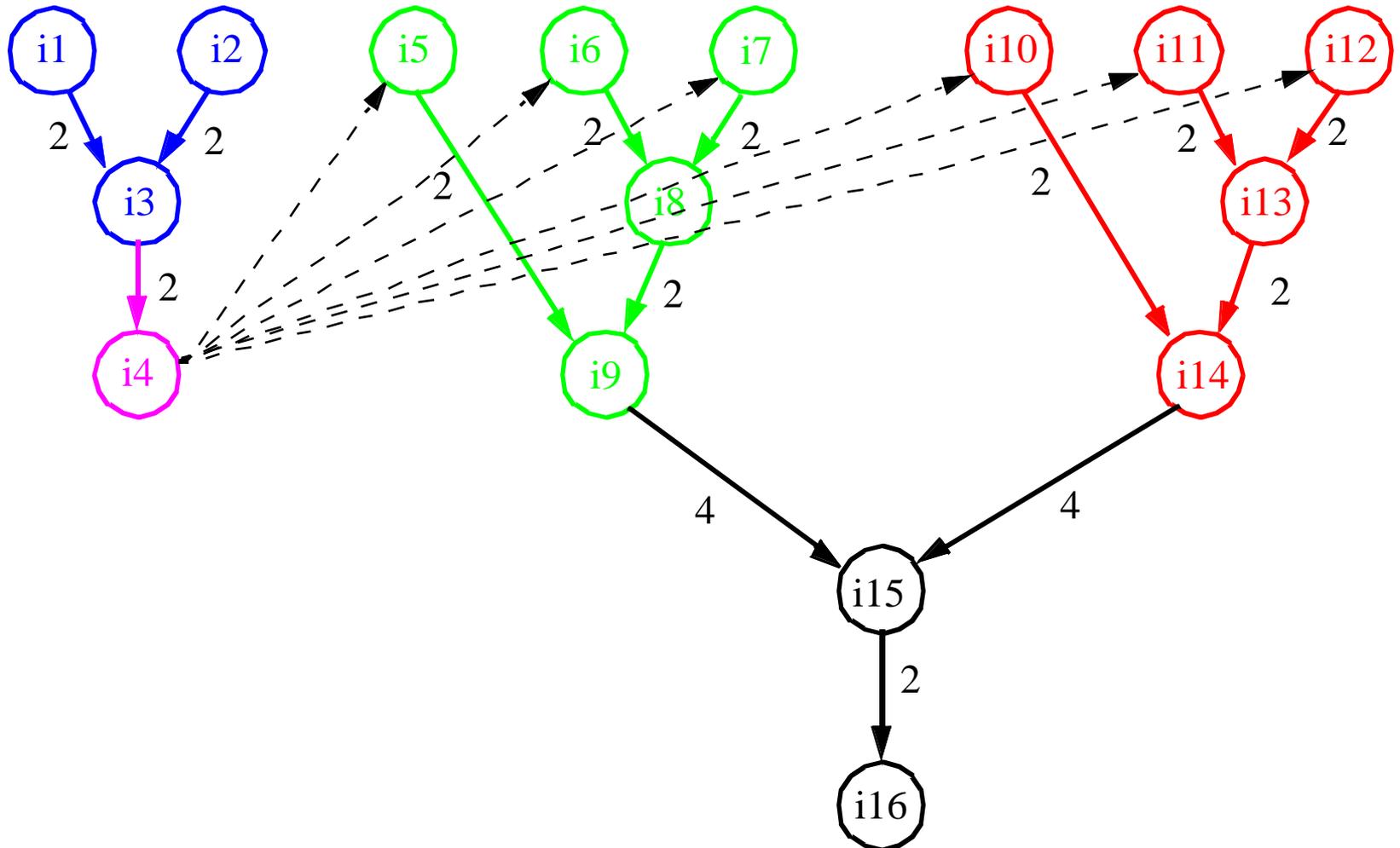
# Trace Scheduling Overview

---

- 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

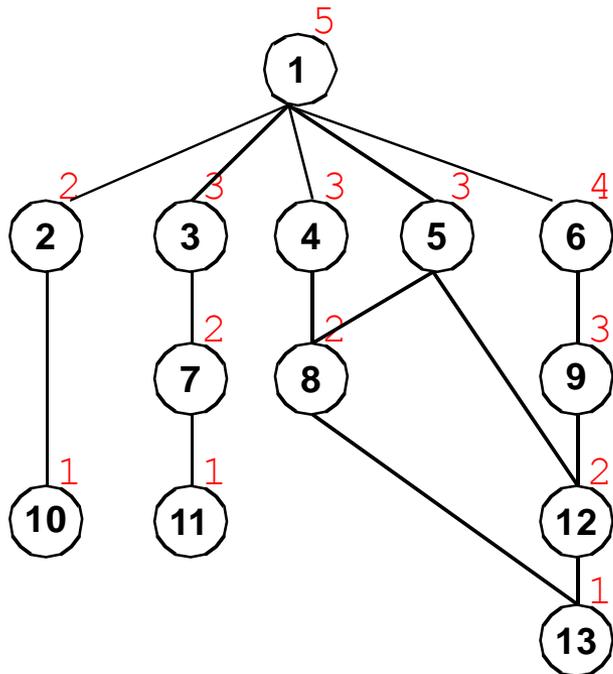
# Instruction Prioritization Heuristics

---

- 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

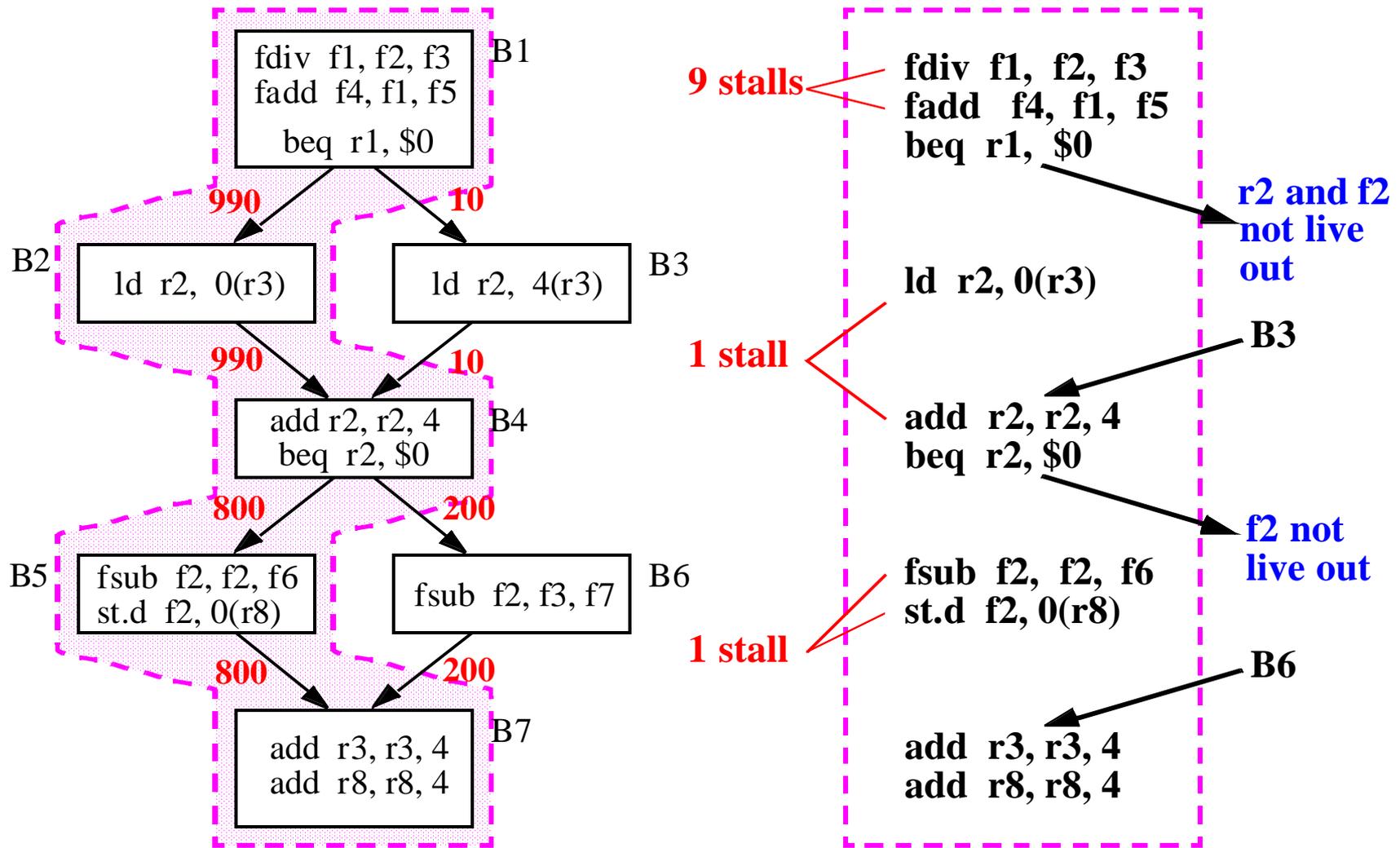
# VLIW List Scheduling

- Assign Priorities
- Compute Data Ready List - 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

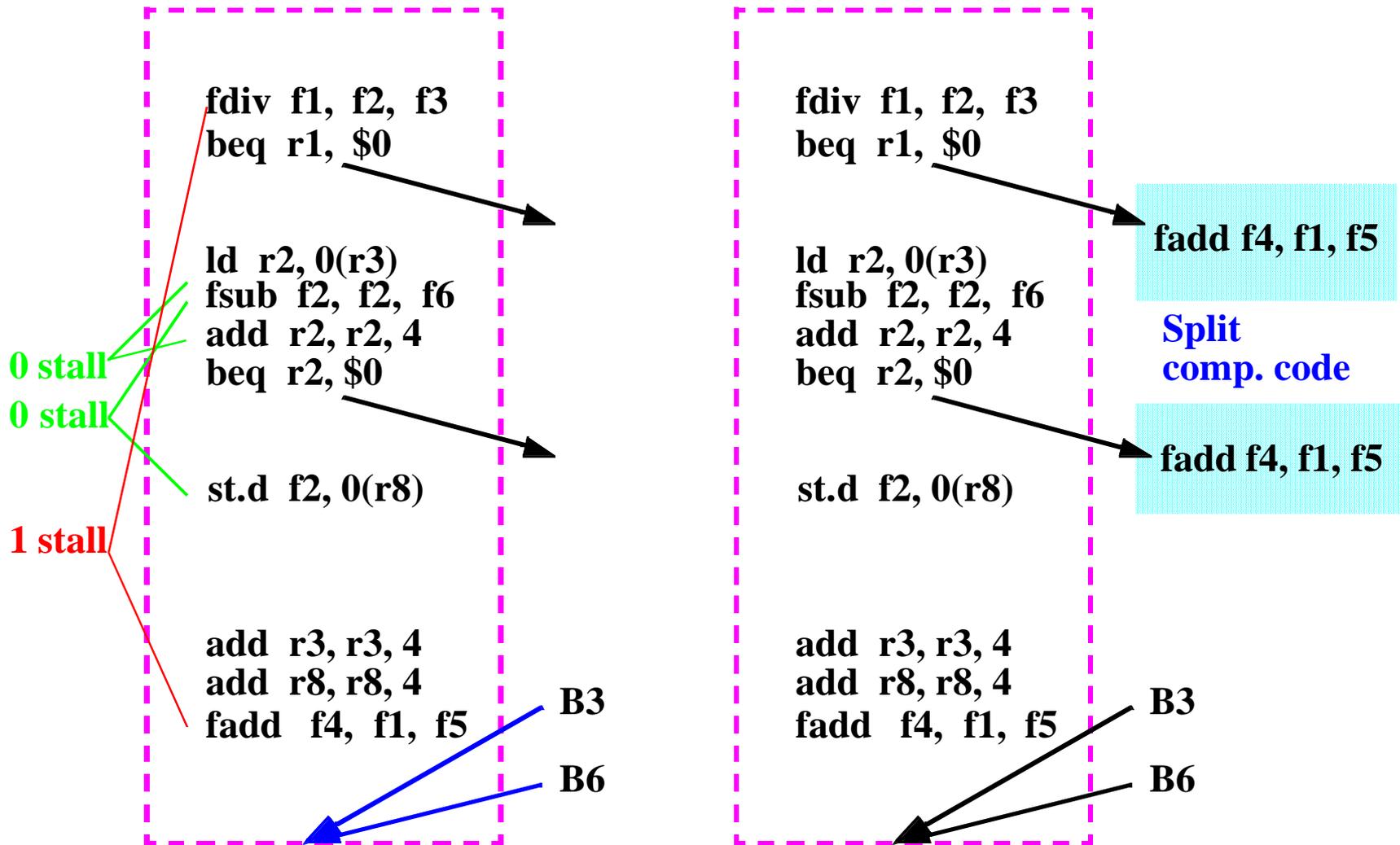


4-wide VLIW				Data Ready List
1				{1}
6	3	4	5	{2,3,4,5,6}
9	2	7	8	{2,7,8,9}
12	10	11		{10,11,12}
13				{13}

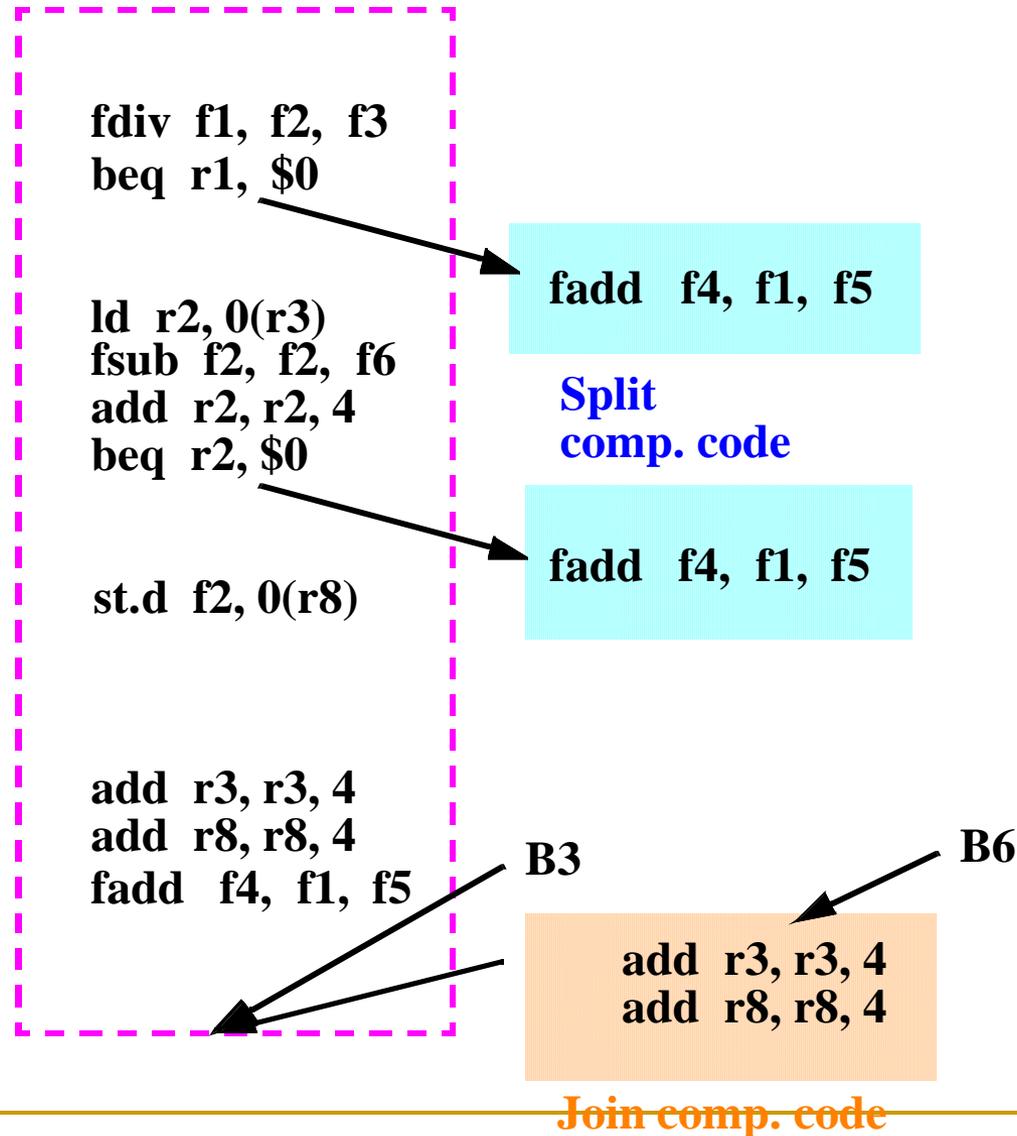
# Trace Scheduling Example (I)



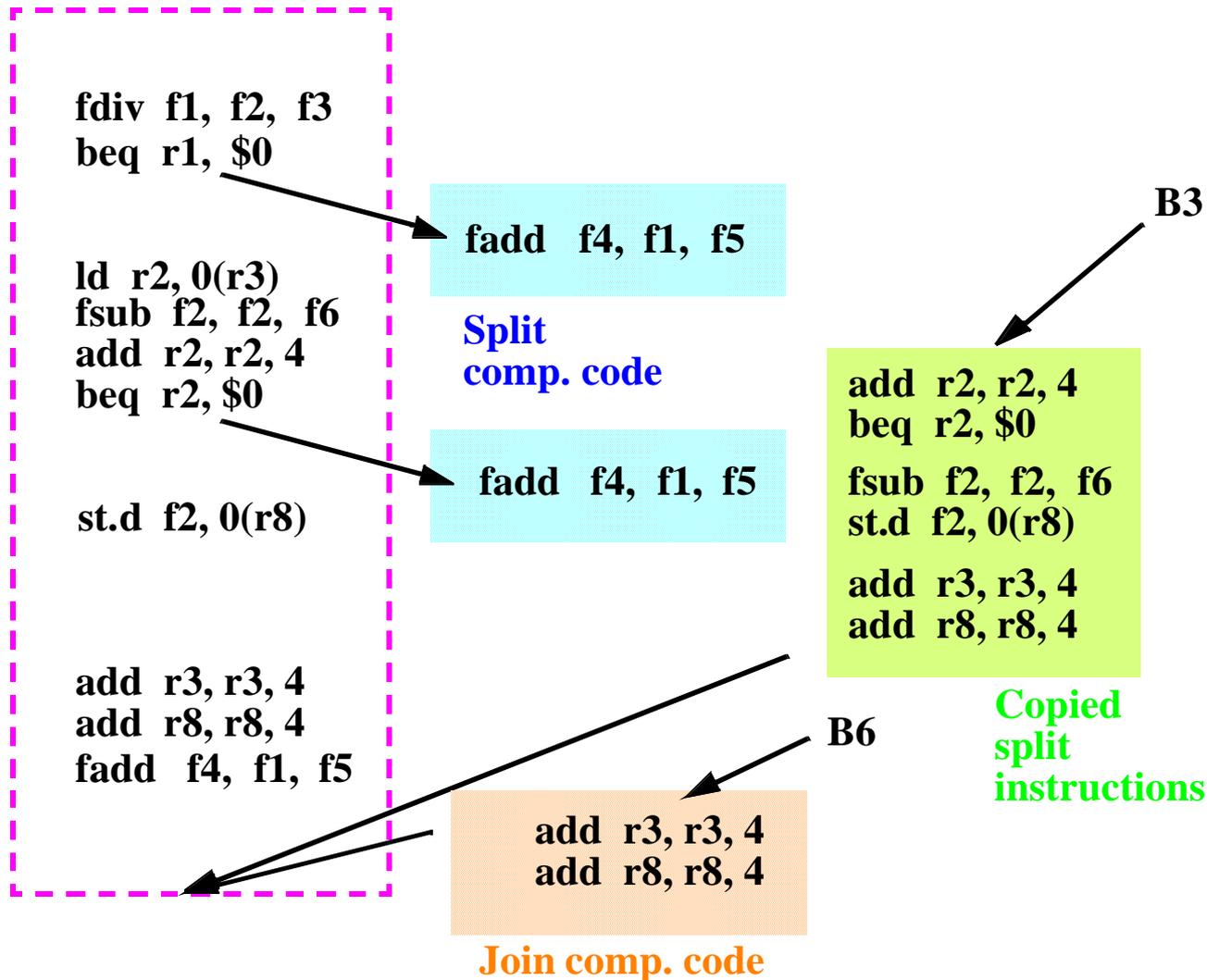
# Trace Scheduling Example (II)



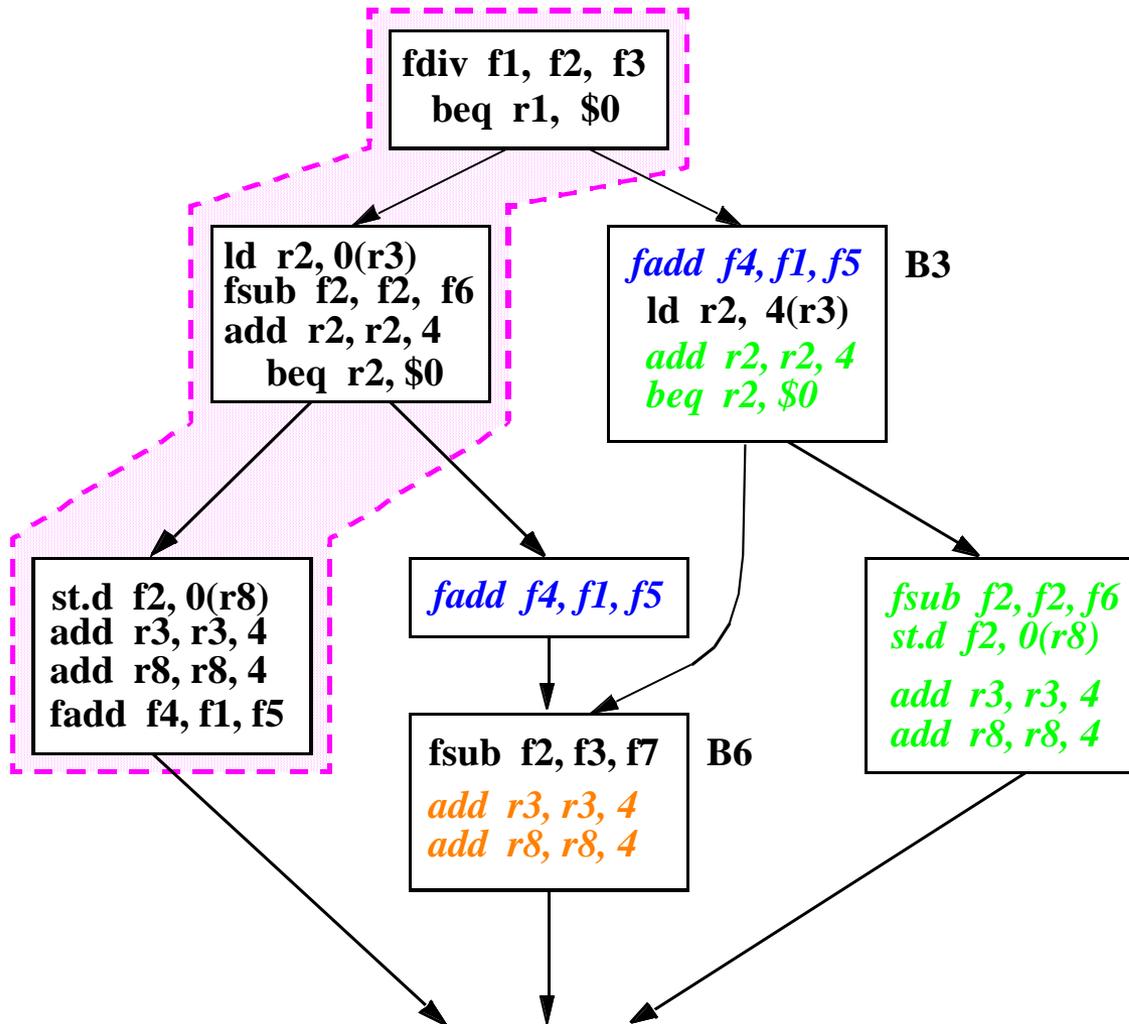
# Trace Scheduling Example (III)



# Trace Scheduling Example (IV)



# 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 → lots of NOPs in the VLIW instructions

- Code bloat and additional fix-up code executed

- Due to side entrances and side exits

- **Infrequent paths interfere with the frequent path**

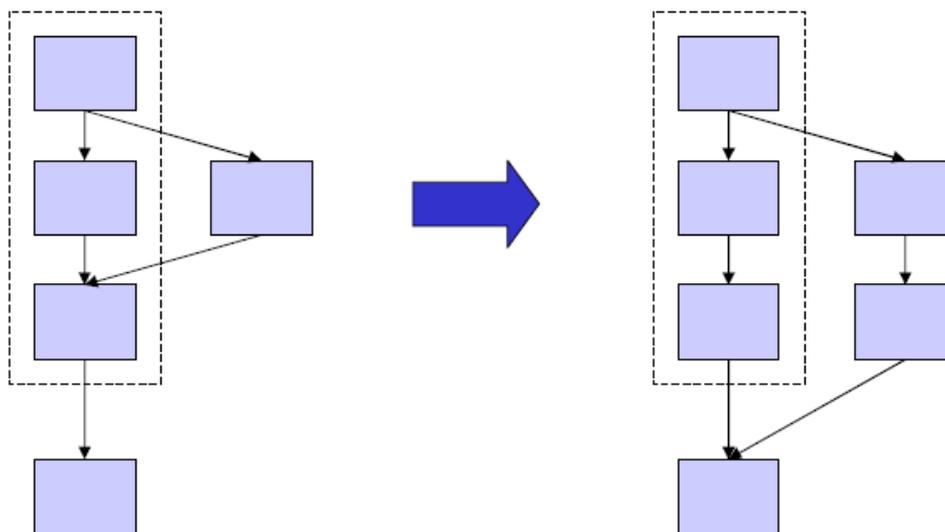
- Effectiveness depends on the bias of branches

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

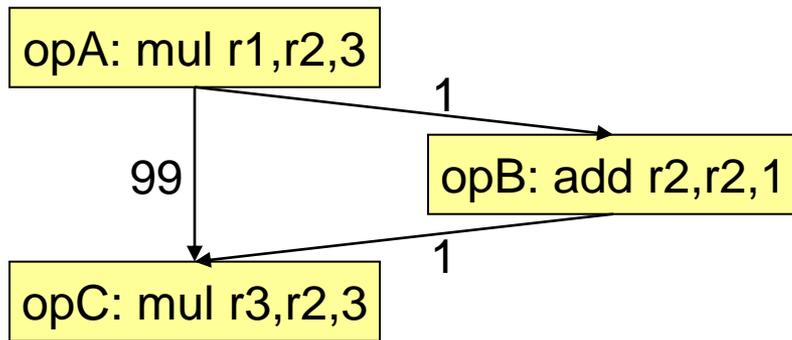
# Superblock Scheduling

---

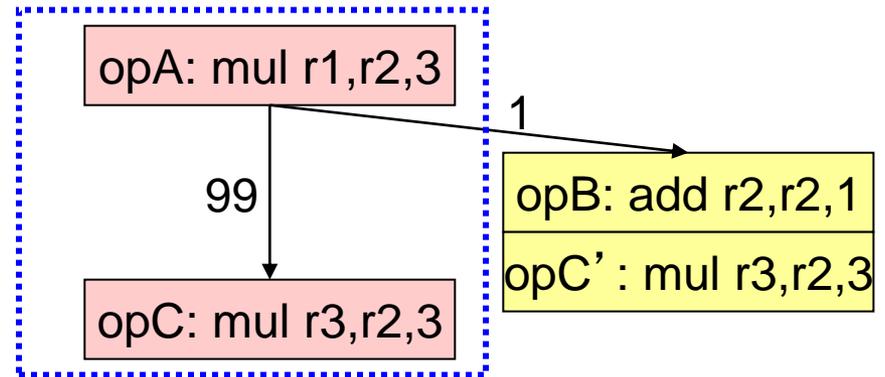
- Trace: multiple entry, multiple exit block
  - Superblock: single-entry, multiple exit block
    - A trace with side entrances are eliminated
    - Infrequent paths do not interfere with the frequent path
- + More optimization/scheduling opportunity than traces
- + Eliminates “difficult” bookkeeping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



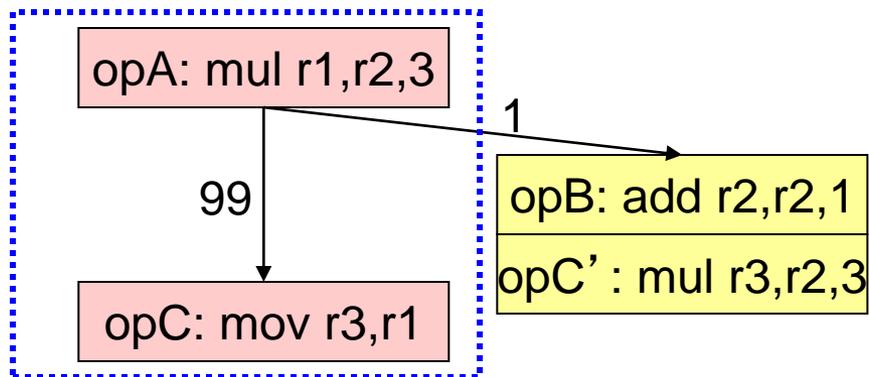
# Can You Do This with a Trace?



Original Code



Code After Superblock Formation



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

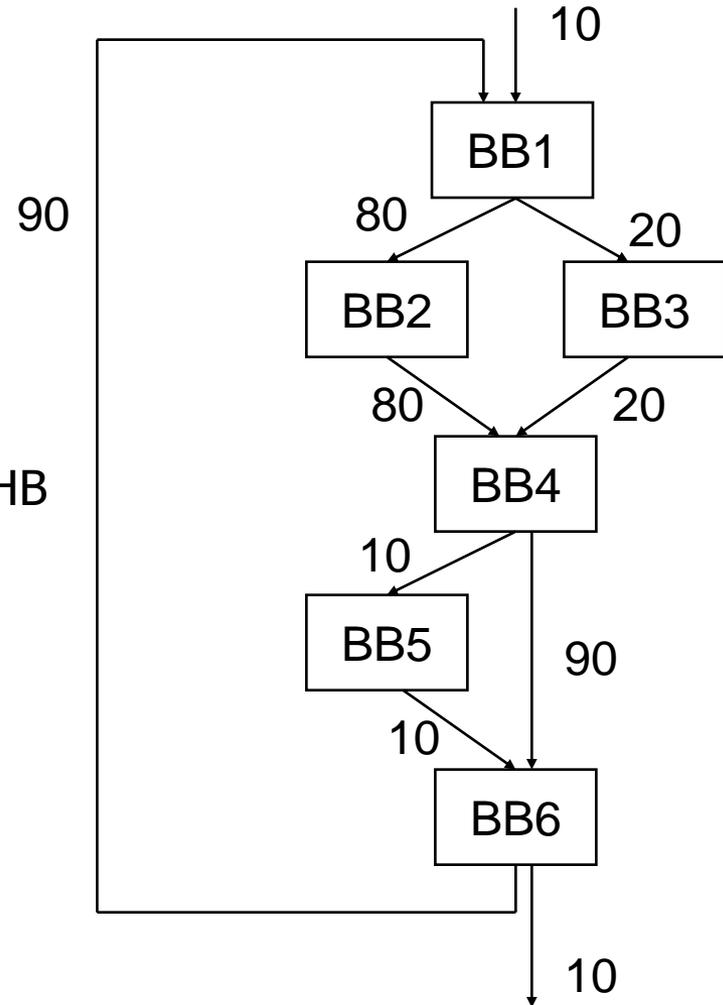
# Hyperblock Scheduling

---

- 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)

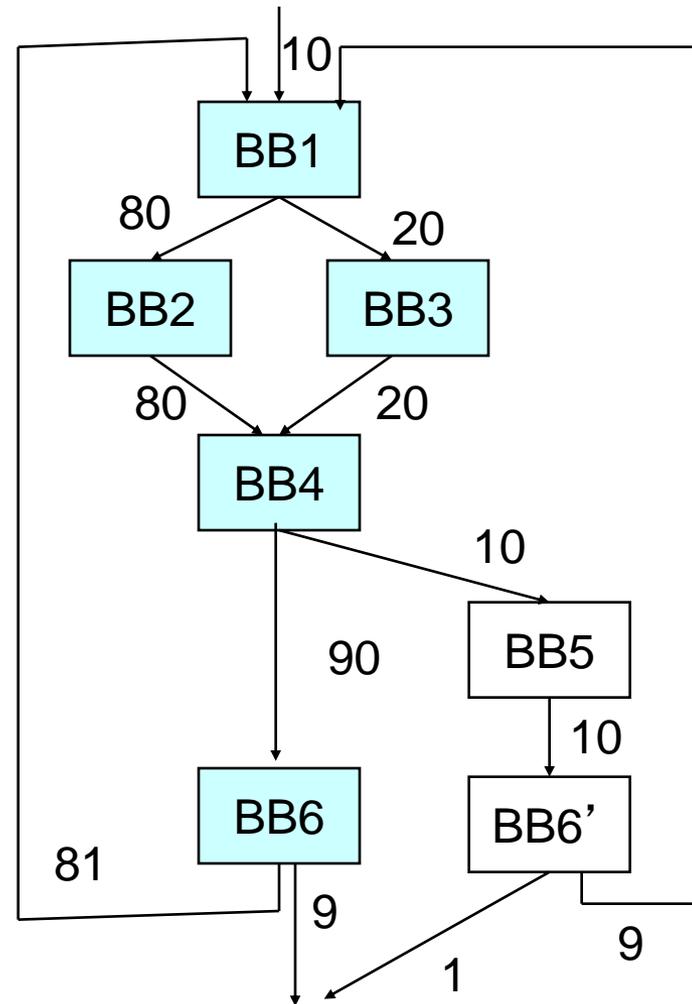
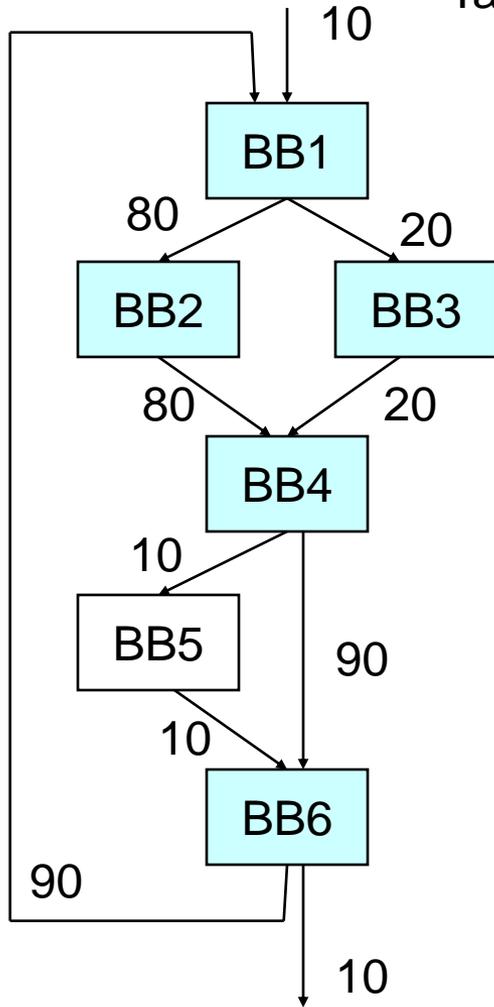
- Hyperblock formation
  1. Block selection
  2. Tail duplication
  3. If-conversion
- Block selection
  - Select subset of BBs for inclusion in HB
  - Difficult problem
  - Weighted cost/benefit function
    - Height overhead
    - Resource overhead
    - Dependency overhead
    - Branch elimination benefit
    - Weighted by frequency



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

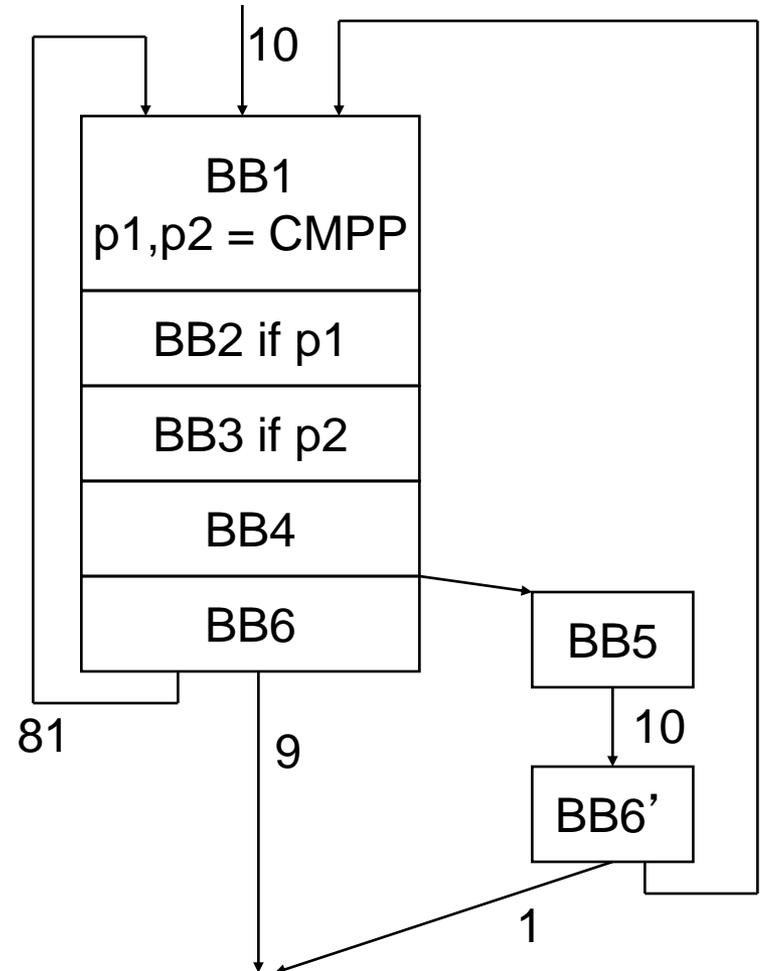
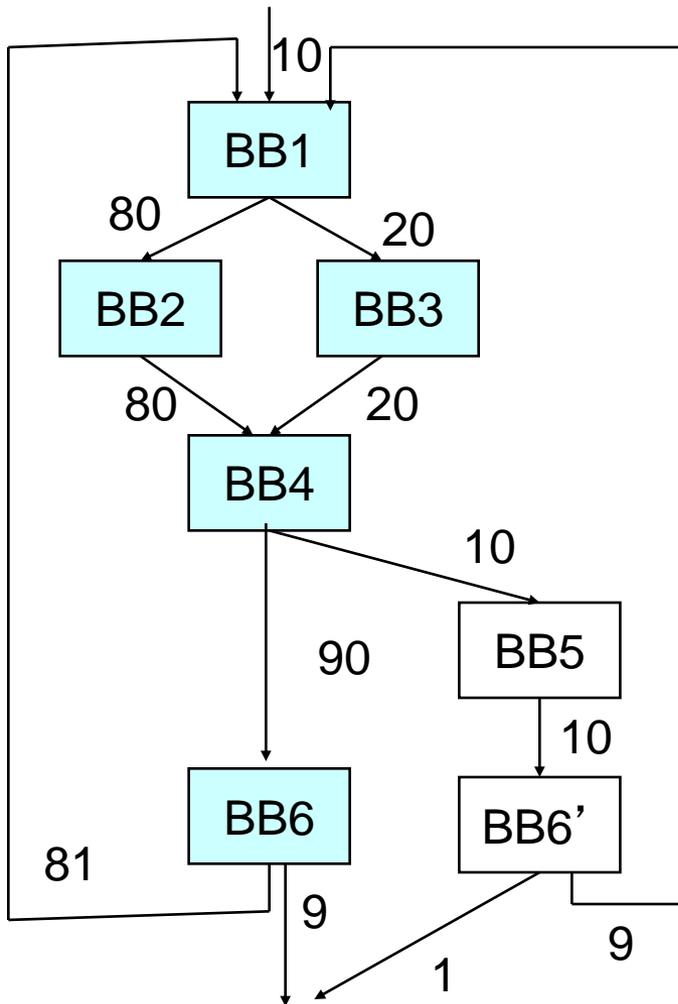
# Hyperblock Formation (II)

Tail duplication same as with Superblock formation



# Hyperblock Formation (III)

If-convert (predicate) intra-hyperblock branches



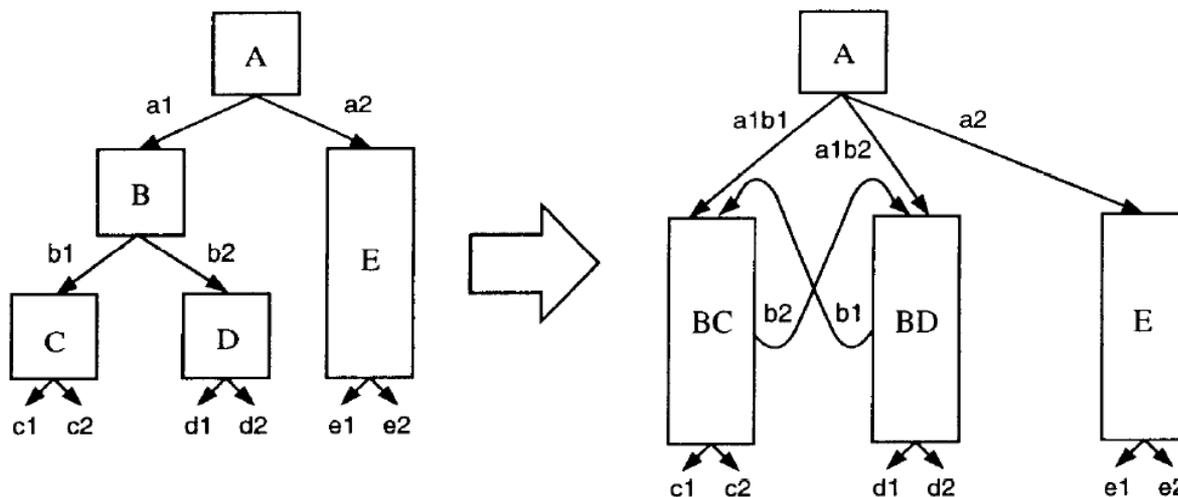
# Can We Do Better?

---

- Hyperblock still
  - Profile dependent
  - Requires fix-up code
  - And, requires predication support
- Single-entry, single-exit enlarged blocks
  - Block-structured ISA
    - Optimizes multiple paths (can use predication to enlarge blocks)
    - No need for fix-up code (duplication instead of fixup)

# 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



# Block Structured ISA (II)

---

- Advantages:
  - + Larger atomic blocks → larger units can be fetched from I-cache
  - + Aggressive compiler optimizations (e.g. reordering) can be enabled within atomic blocks (no side entries or exits)
  - + Can explicitly represent dependencies among operations within an enlarged block
- Disadvantages:
  - “Fault operations” can lead to work to be wasted (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
- Optimizations
  - Within an enlarged block, the compiler can perform optimizations that cannot normally/easily be performed across basic blocks

# Block Structured ISA (III)

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

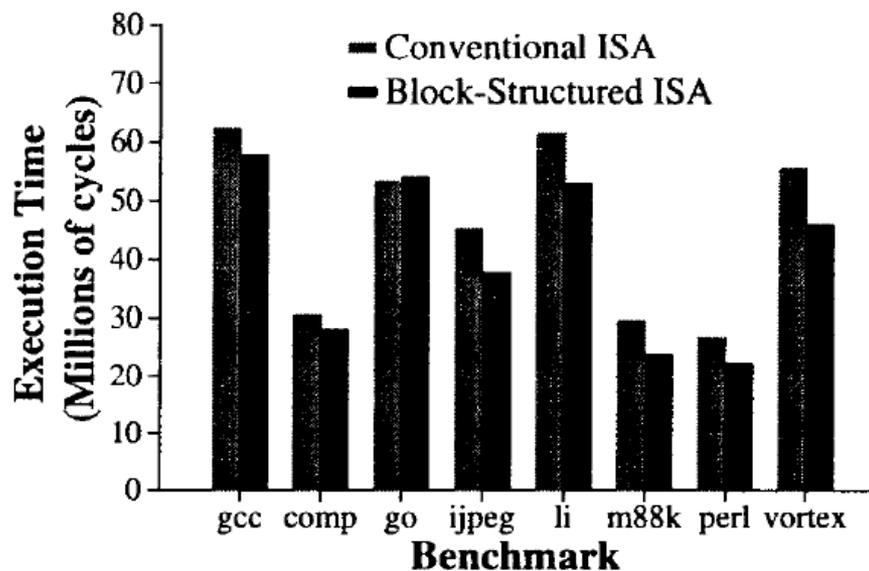


Figure 3. Performance comparison of block-structured ISA executables and conventional ISA executables.

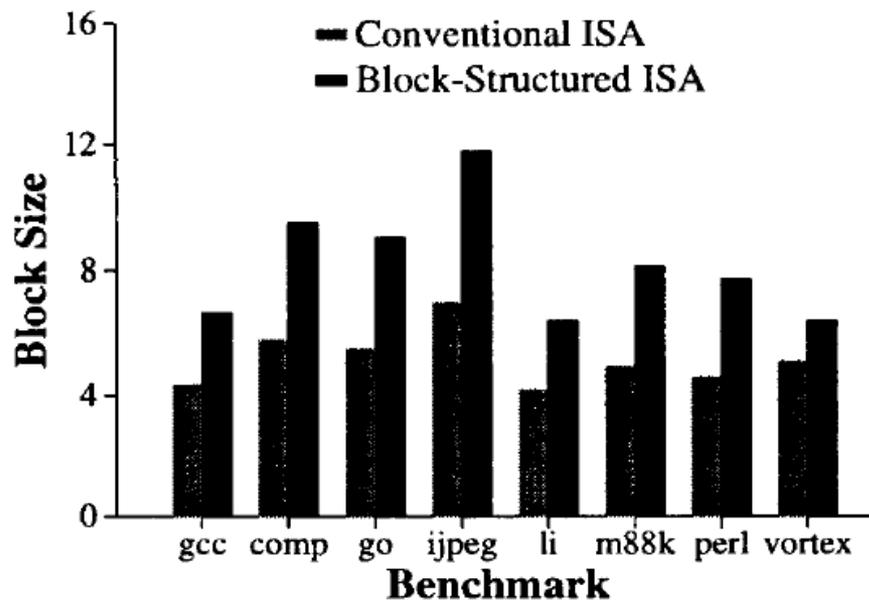


Figure 5. Average block sizes for block-structured and conventional ISA executables.

# Superblock vs. BS-ISA

---

- Superblock
  - Single-entry, multiple exit code block
  - Not atomic
  - Compiler inserts fix-up code on superblock side exit
  
- BS-ISA blocks
  - Single-entry, single exit
  - Atomic
  - Need to roll back to the beginning of the block on fault

# 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

# IA-64: A Complicated VLIW

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



- 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

```
{ .mi1
  add r1 = r2, r3
  sub r4 = r4, r5 ;;
  shr r7 = r4, r12 ;;
}
{ .mm1
  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

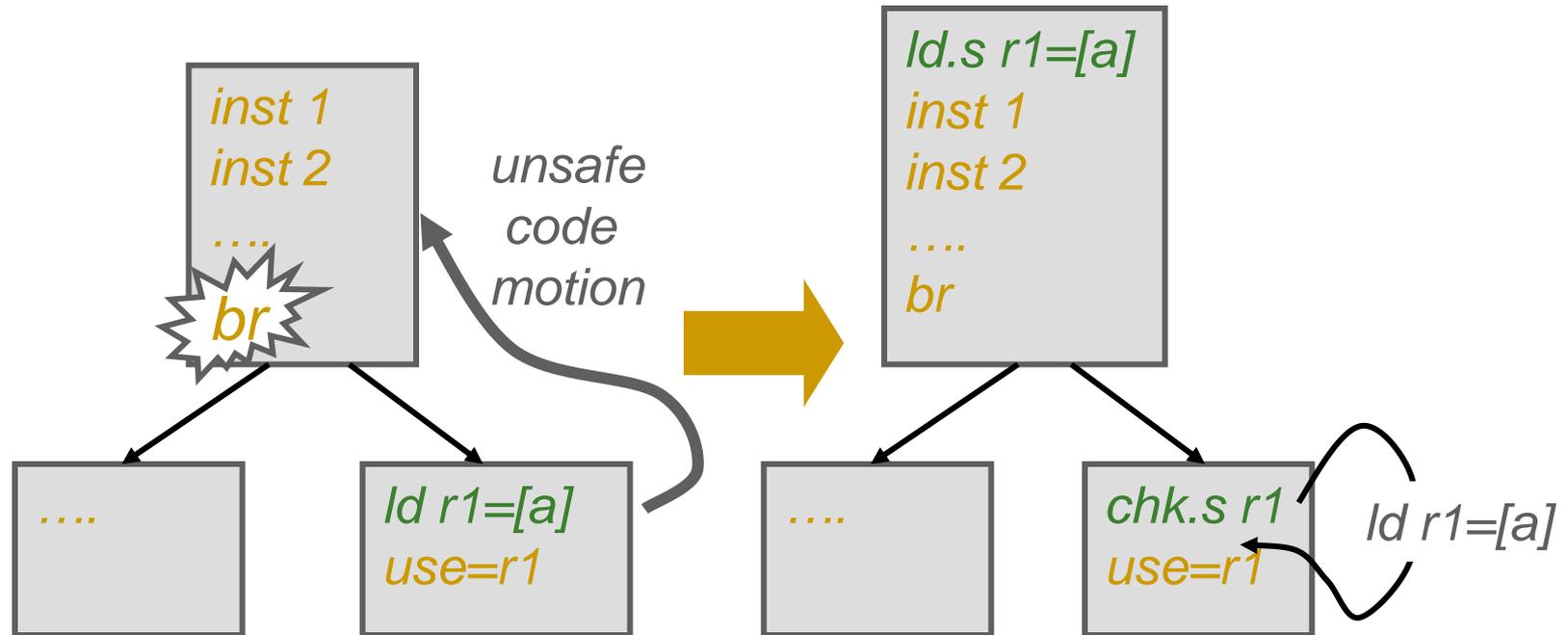
- Specify two things
  - Stop information: Boundary of independent instructions
  - Functional unit information: Where should each instruction be routed

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 <sup>a</sup>
05	M-unit	L-unit	X-unit <sup>a</sup>
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

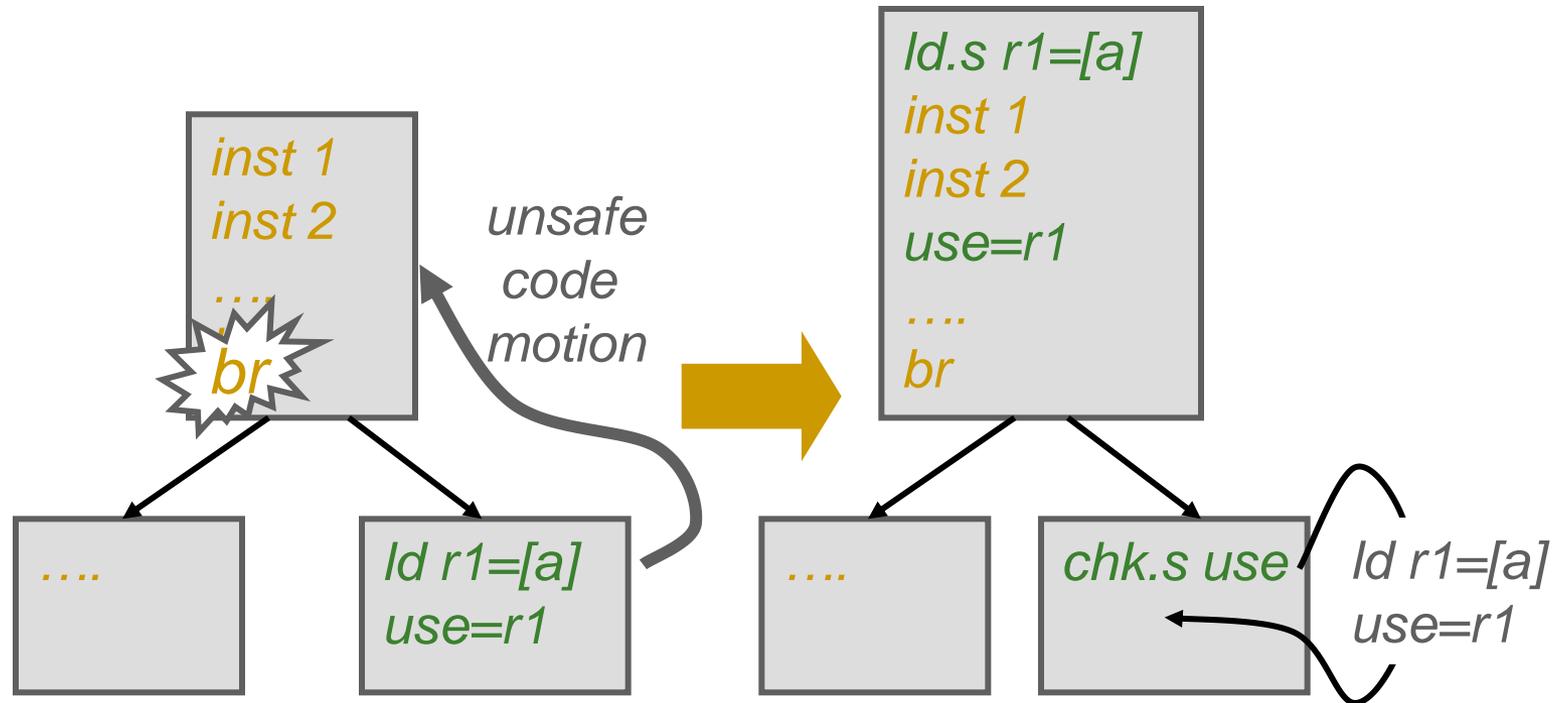
Template	Slot 0	Slot 1	Slot 2
13	M-unit	B-unit	B-unit
14			
15			
16	B-unit	B-unit	B-unit
17	B-unit	B-unit	B-unit
18	M-unit	M-unit	B-unit
19	M-unit	M-unit	B-unit
1A			
1B			
1C	M-unit	F-unit	B-unit
1D	M-unit	F-unit	B-unit
1E			
1F			

# Non-Faulting Loads and Exception Propagation



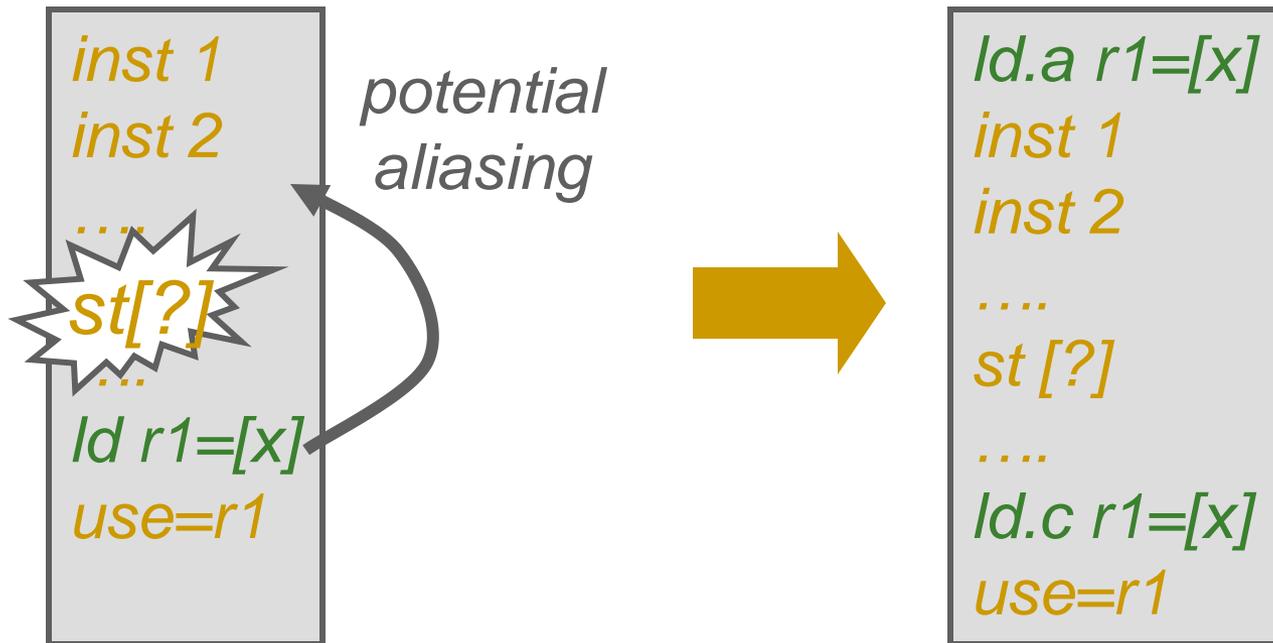
- *ld.s* fetches *speculatively* from memory  
i.e. any exception due to *ld.s* is suppressed
- If *ld.s r1* did not cause an exception then *chk.s r1* is a NOP, else a branch is taken (to execute some compensation code)

# Non-Faulting Loads and Exception Propagation in IA-64



- 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



- *ld.a* 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

---

