

15-740/18-740
Computer Architecture
Lecture 27: VLIW

Prof. Onur Mutlu
Carnegie Mellon University

Announcements

- Project Poster Session
 - December 10
 - NSH Atrium
 - 2:30-6:30pm
- Project Report Due
 - December 12
 - The report should be like a good conference paper
- Focus on Projects
 - All group members should contribute
 - Use the milestone feedback from the TAs

Final Project Report and Logistics

- Follow the guidelines in project handout
 - We will provide the Latex format
- Good papers should be similar to the best conference papers you have been reading throughout the semester
- Submit all code, documentation, supporting documents and data
 - Provide instructions as to how to compile and use your code
 - This will determine part of your grade
- This is the single most important part of the project

Best Projects

- Best projects will be encouraged for a top conference submission
 - Talk with me if you are interested in this

- Examples from past:
 - Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter, "[ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers](#)," HPCA 2010 Best Paper Session
 - George Nychis, Chris Fallin, Thomas Moscibroda, and Onur Mutlu, "[Next Generation On-Chip Networks: What Kind of Congestion Control Do We Need?](#)," HotNets 2010.
 - Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter, "[Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior](#)," MICRO 2010. (IEEE Micro Top Picks 2010)

Today

- Alternative approaches to concurrency
 - SISD/SIMD/MISD/MIMD classification
 - Decoupled Access/Execute
 - VLIW
 - Vector Processors and Array Processors
 - Data Flow

Alternative Approaches to Concurrency

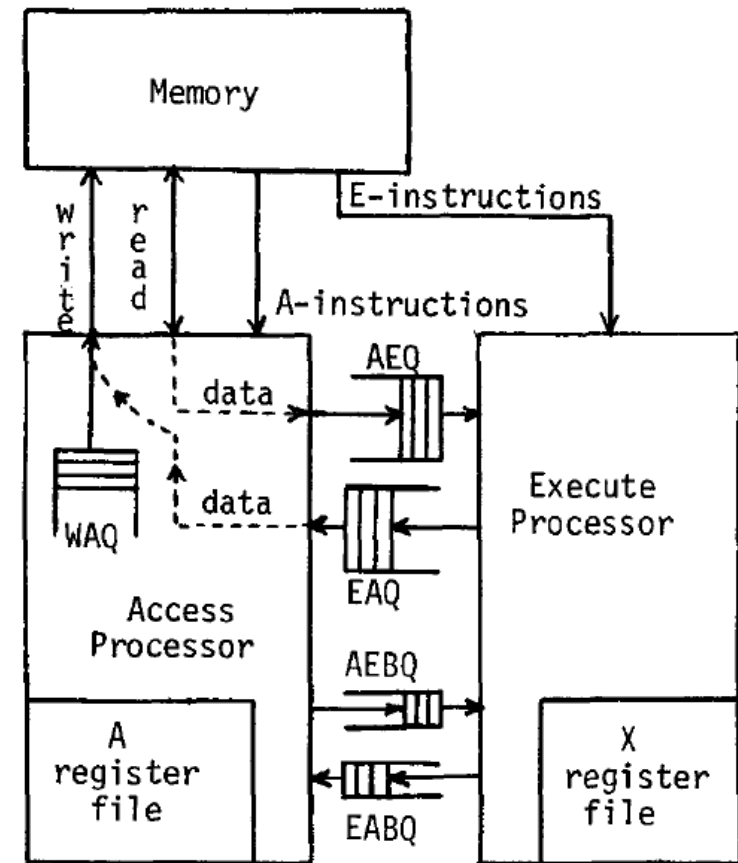
Readings

- Required:
 - Fisher, “**Very Long Instruction Word architectures and the ELI-512,**” ISCA 1983.
 - Huck et al., “**Introducing the IA-64 Architecture,**” IEEE Micro 2000.

- Recommended:
 - Russell, “**The CRAY-1 computer system,**” CACM 1978.
 - 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.

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.



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
}
```

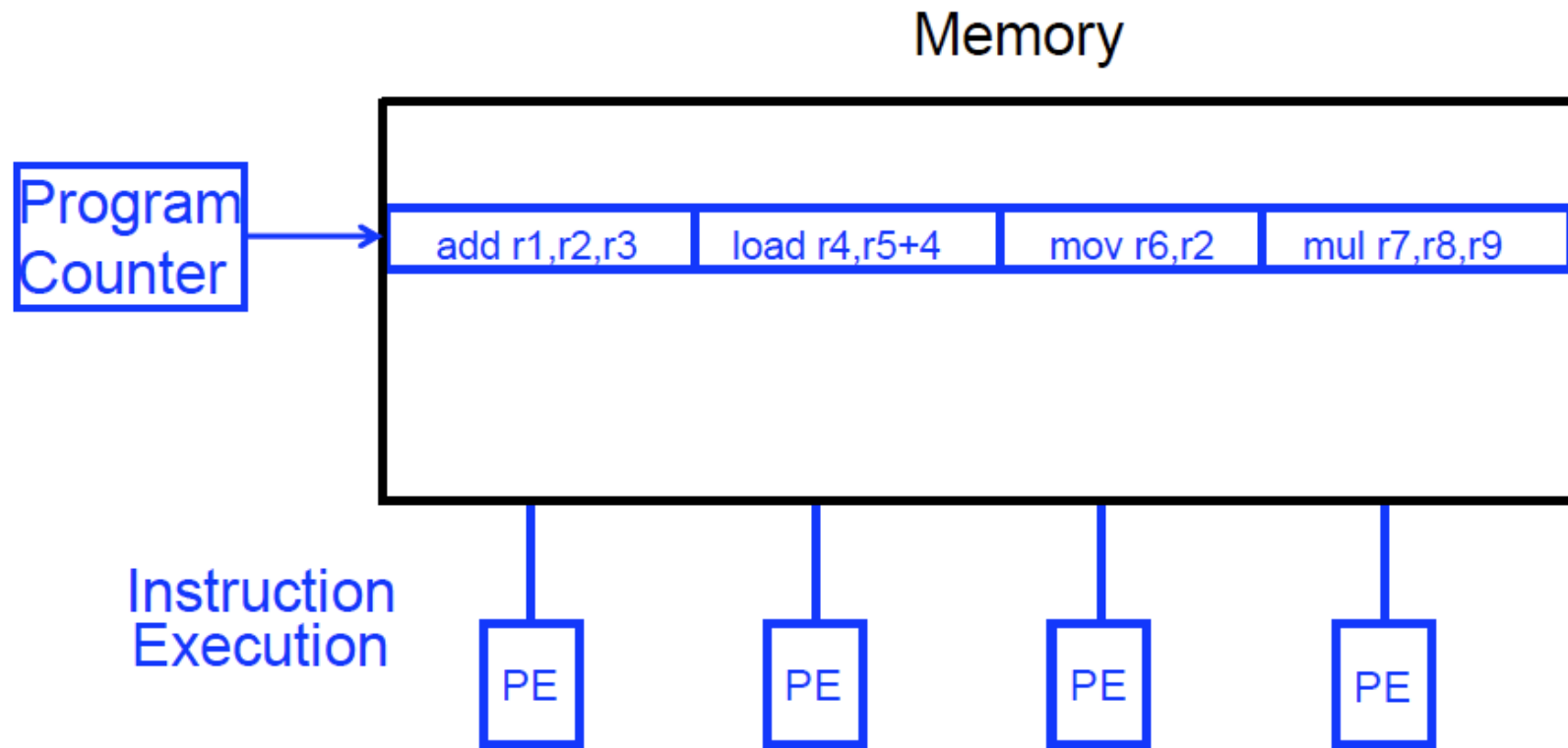
```
i = 1;
while ( i < 100 ) {
    a[i] = b[i+1] + (i+1)/m
    b[i] = a[i-1] - i/m
    a[i+1] = b[i+2] + (i+2)/m
    b[i+1] = a[i] - (i+1)/m
    i = i + 2
}
```

- 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 (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)
- Idea: Compiler finds independent instructions and statically schedules (i.e. packs/bundles) them into a single VLIW instruction
- Traditional Characteristics
 - Multiple functional units
 - Each instruction in a bundle executed in **lock step**
 - **Instructions** in a bundle **statically aligned** to be directly fed into the functional units

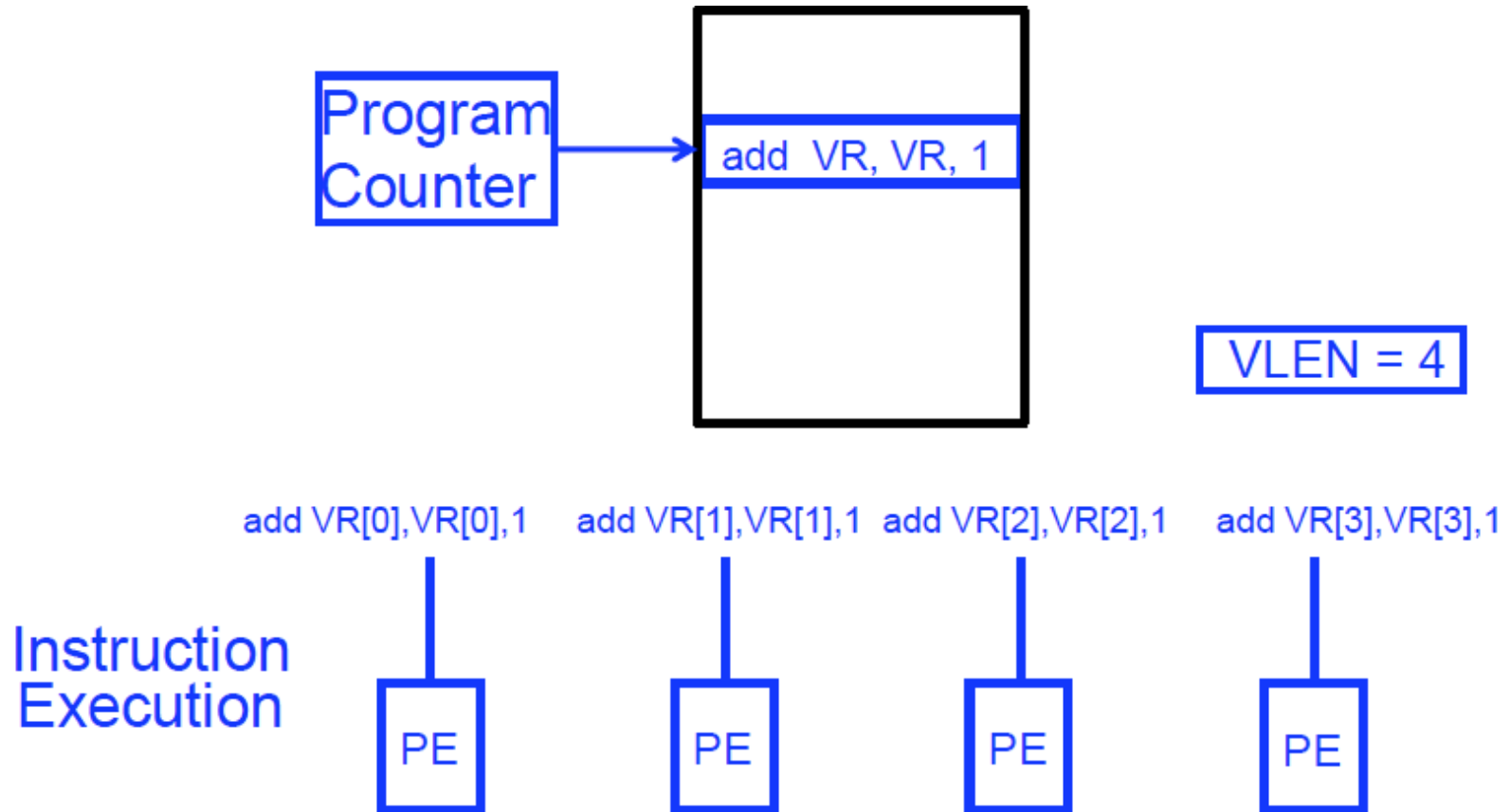
VLIW Concept



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

SIMD Array Processing vs. VLIW

- Array processor



VLIW Philosophy

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

Commercial VLIW Machines

- Multiflow TRACE, Josh Fisher
- Cydrome Cydra 5, Bob Rau
- Transmeta Crusoe: x86 binary-translated into internal VLIW
- TI C6000, Trimedia, STMicro (DSP & embedded processors)
 - 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

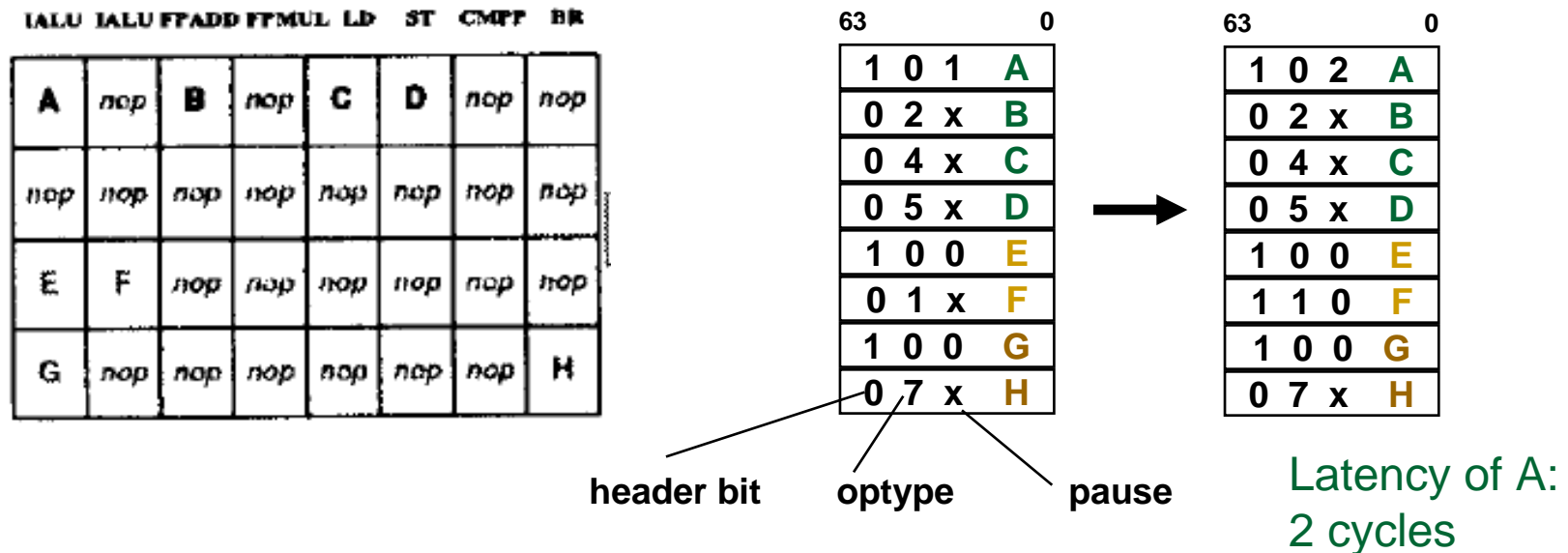
- Compiler needs to find N independent operations
 - If it cannot, inserts NOPs in a VLIW instruction
 - Parallelism loss AND code size increase
- Recompile required when execution width (N), instruction latencies, functional units change (Unlike superscalar processing)
- Lockstep execution causes independent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

VLIW NOPs

- Cause code bloat + reduce performance
 - Early VLIW machines suffered from this (Multiflow, Cydrome)
- Modern EPIC machines use compaction encoding
- Idea: Encode the existence or lack of NOPs rather than explicitly inserting NOPs
- VLIW Instruction: Variable-length bundles of instructions
- Instruction: Fixed length
- Instruction format
 - Header bit (cycle starting with this operation)
 - Operation type (dispersement)
 - Pause specifier (cycles of NOPs inserted after current cycle)

VLIW NOP Encoding

- Conte et al., "Instruction Fetch Mechanisms for VLIW Architectures with Compressed Encodings," MICRO 1996.



- + No NOPs in the code or I-cache
- Variable length VLIW instructions, more work decoding
- Does not eliminate the performance degradation of NOPs

Precursor to IA-64 instruction encodings

Static Instruction Scheduling

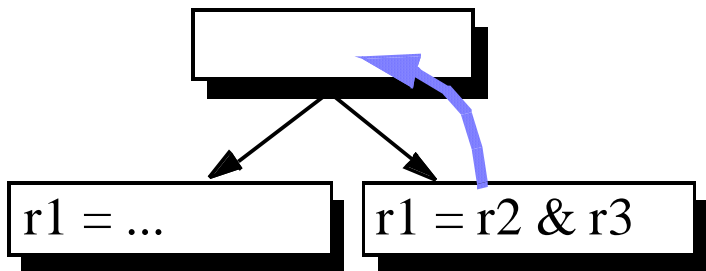
- What does the compiler need to know?
- For VLIW scheduling and instruction formation
 - VLIW width
 - Functional unit types and organization
 - Functional unit latencies
- For scheduling in superscalar, in-order processors
 - Superscalar width
 - Functional unit latencies

VLIW: Finding Independent Operations

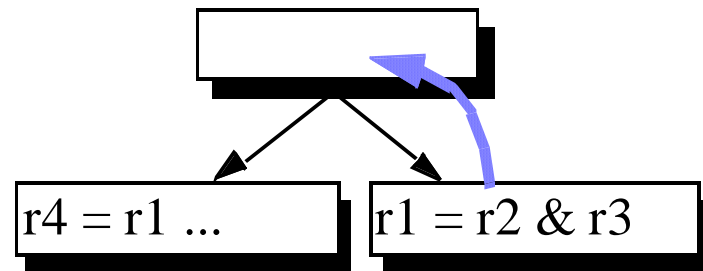
- Within a basic block, there is limited instruction-level parallelism
- To find multiple instructions to be executed in parallel, the compiler needs to consider multiple basic blocks
- Problem: Moving instructions above a branch is unsafe because instruction is not guaranteed to be executed
- Idea: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

Safety and Legality in Code Motion

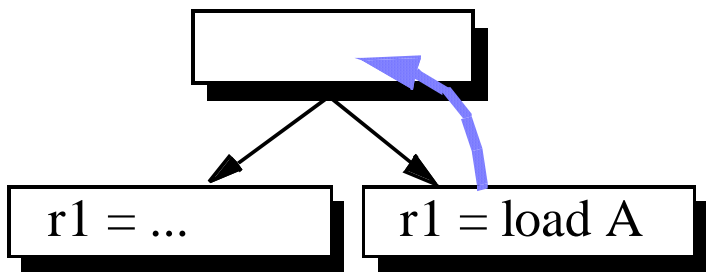
- Two characteristics of speculative code motion:
 - Safety: whether or not spurious exceptions may occur
 - Legality: whether or not result will be correct always
- Four possible types of code motion:



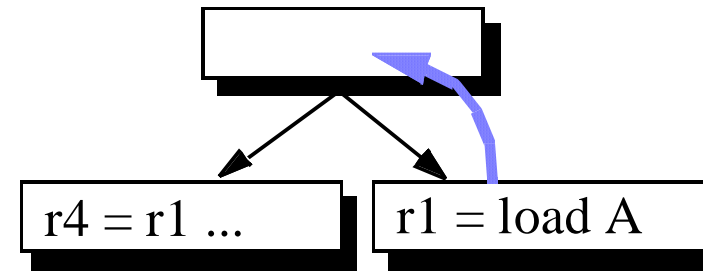
(a) safe and legal



(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

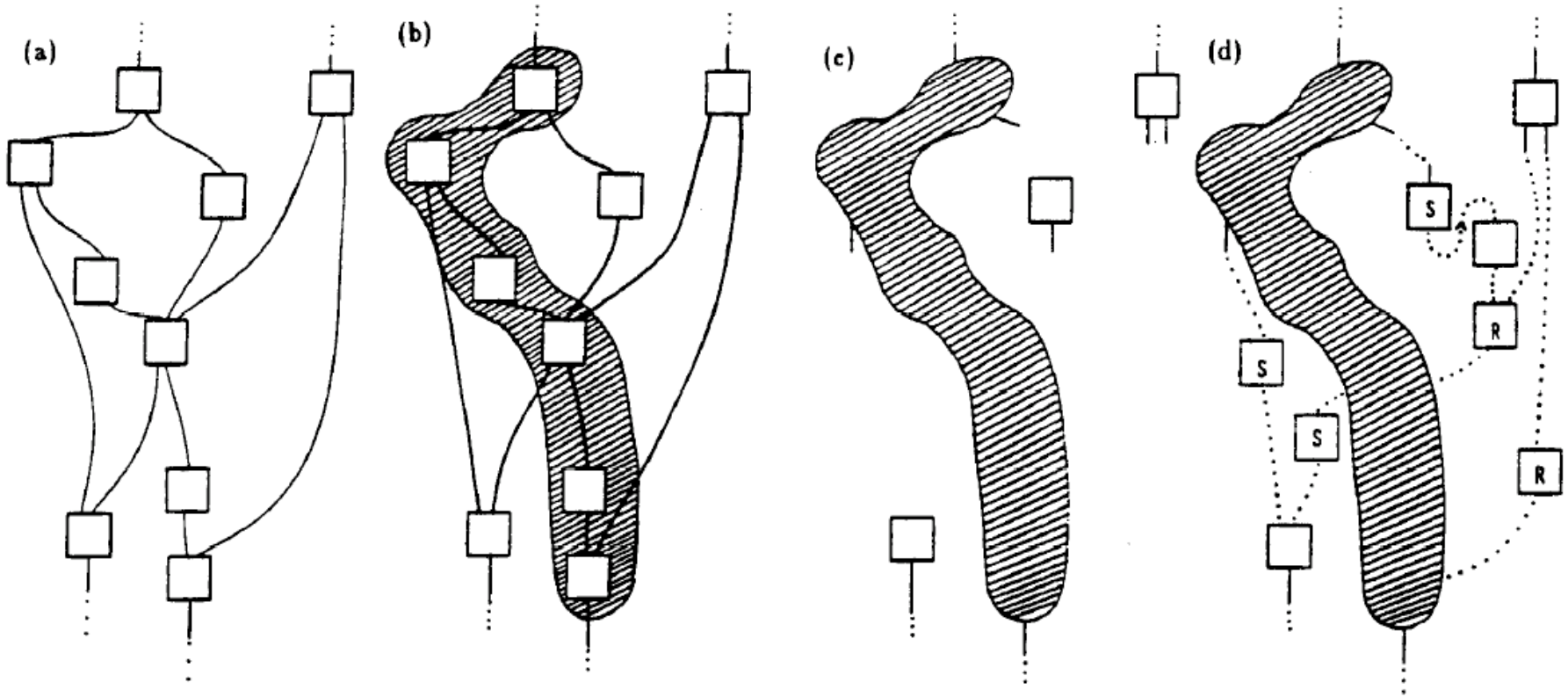
Trace Scheduling

- 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)

Trace Scheduling (II)

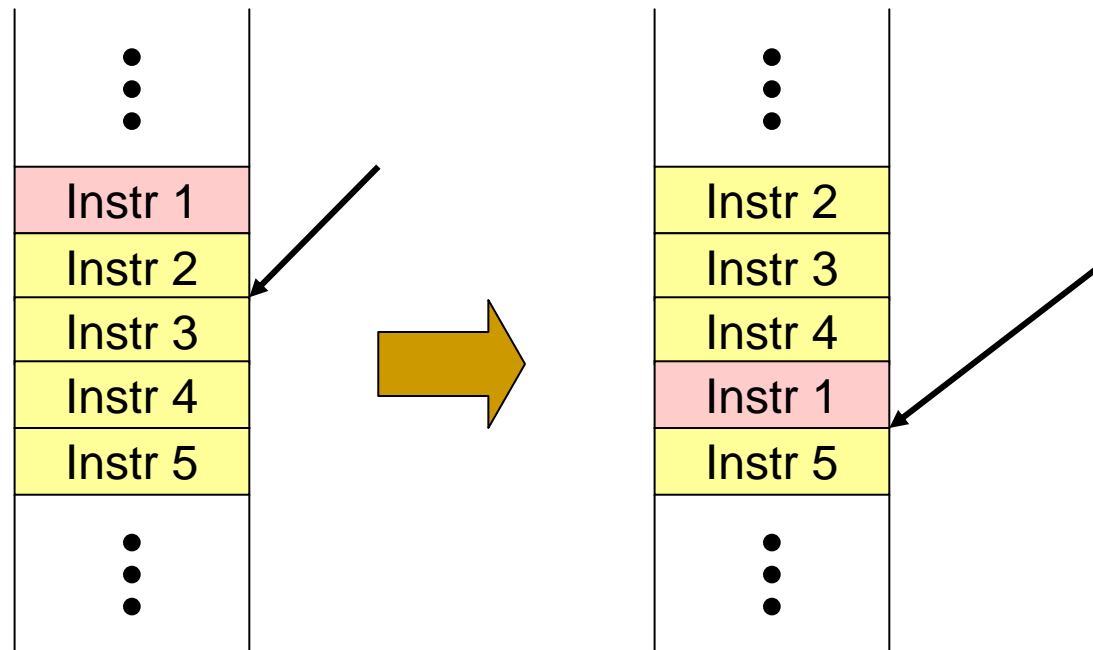
- There may be conditional branches from the middle of the trace (**side exits**) and transitions from other traces into the middle of the trace (**side entrances**).
- These control-flow transitions are ignored during trace scheduling.
- After scheduling, bookkeeping code is inserted to ensure the correct execution of off-trace code.
- Fisher, "**Trace scheduling: A technique for global microcode compaction**," IEEE TC 1981.

Trace Scheduling Idea



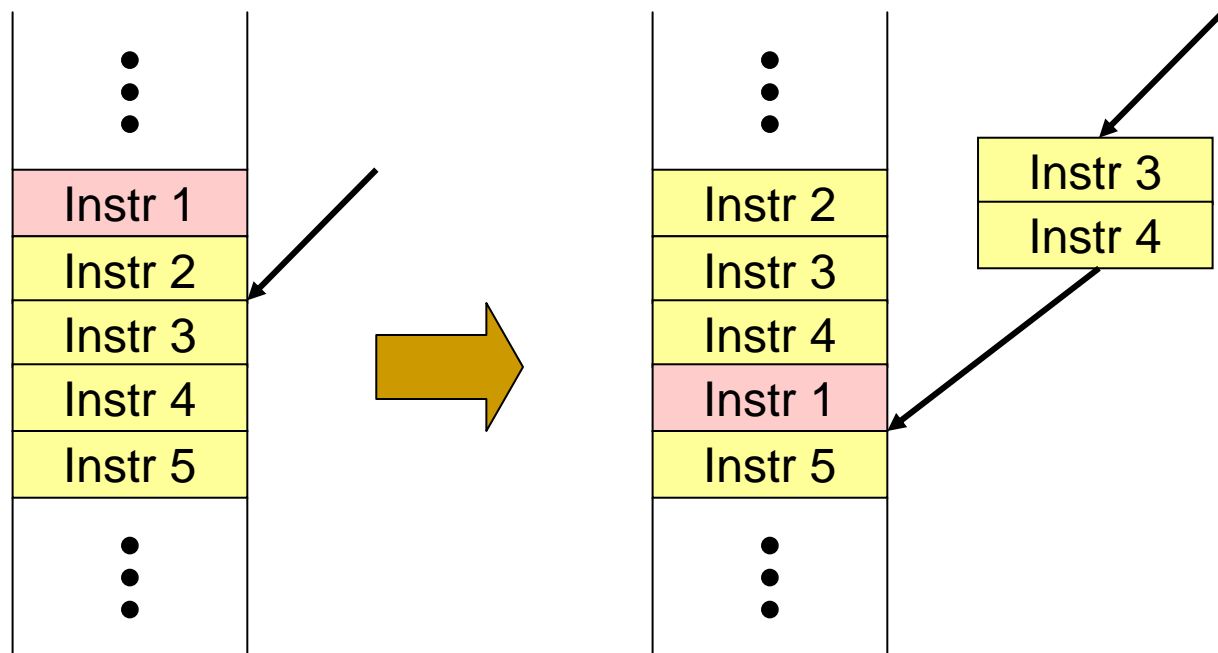
TRACE SCHEDULING LOOP-FREE CODE

Trace Scheduling (III)

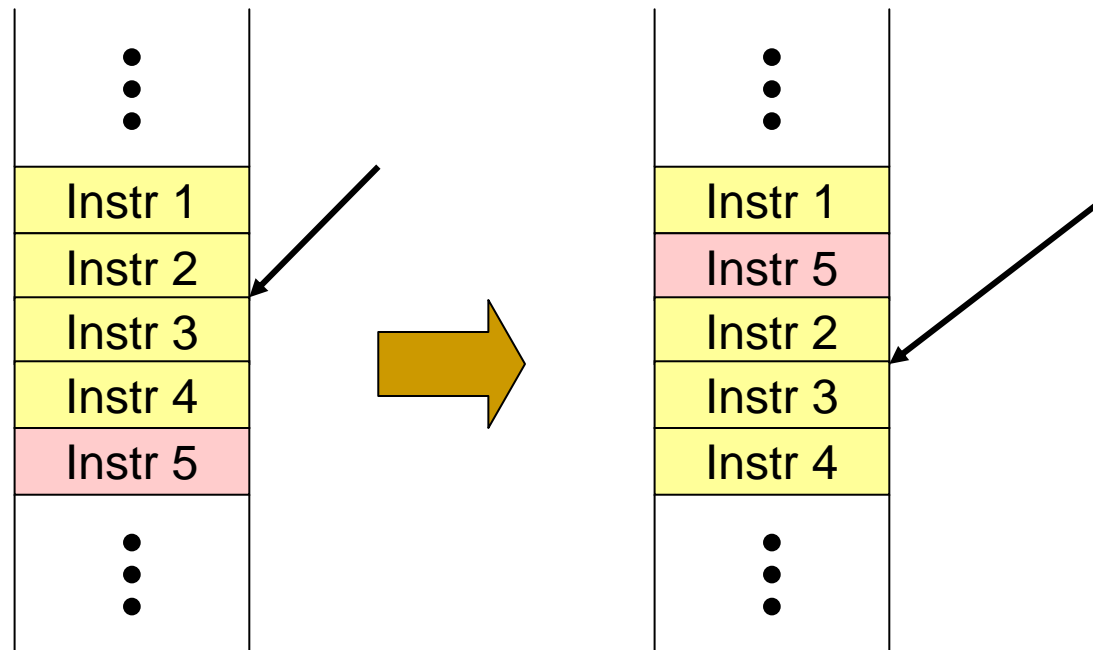


What bookkeeping is required when **Instr 1** is moved below the side entrance in the trace?

Trace Scheduling (IV)

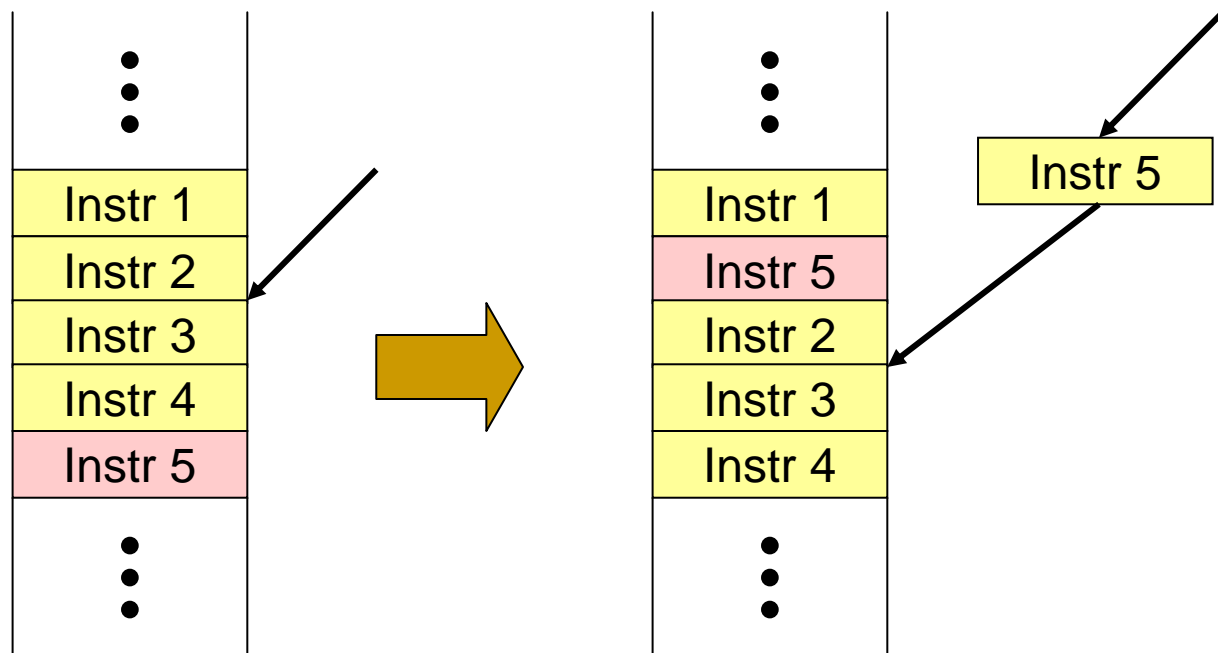


Trace Scheduling (V)



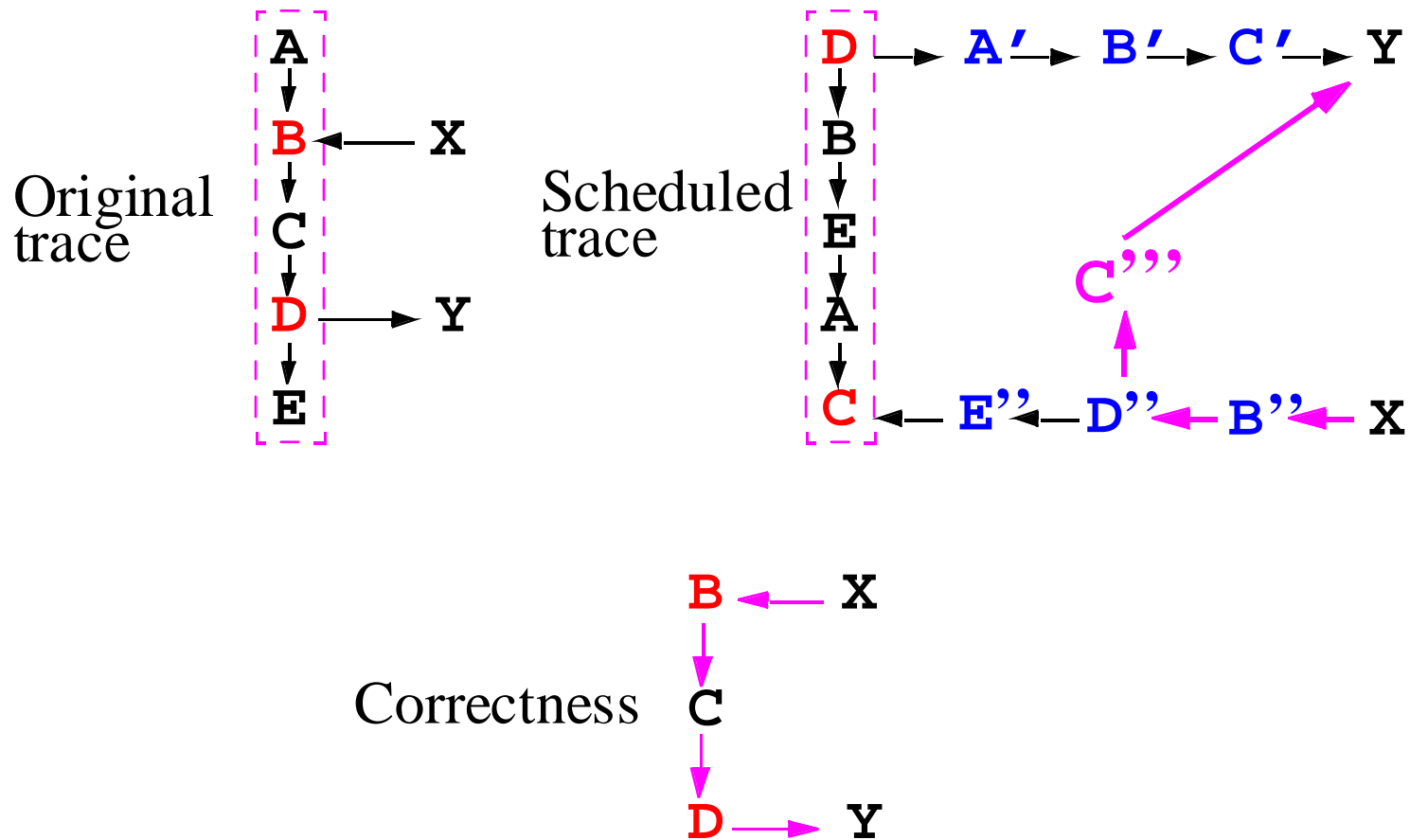
What bookkeeping is required when **Instr 5** moves above the side entrance in the trace?

Trace Scheduling (VI)



Trace Scheduling Fixup Code Issues

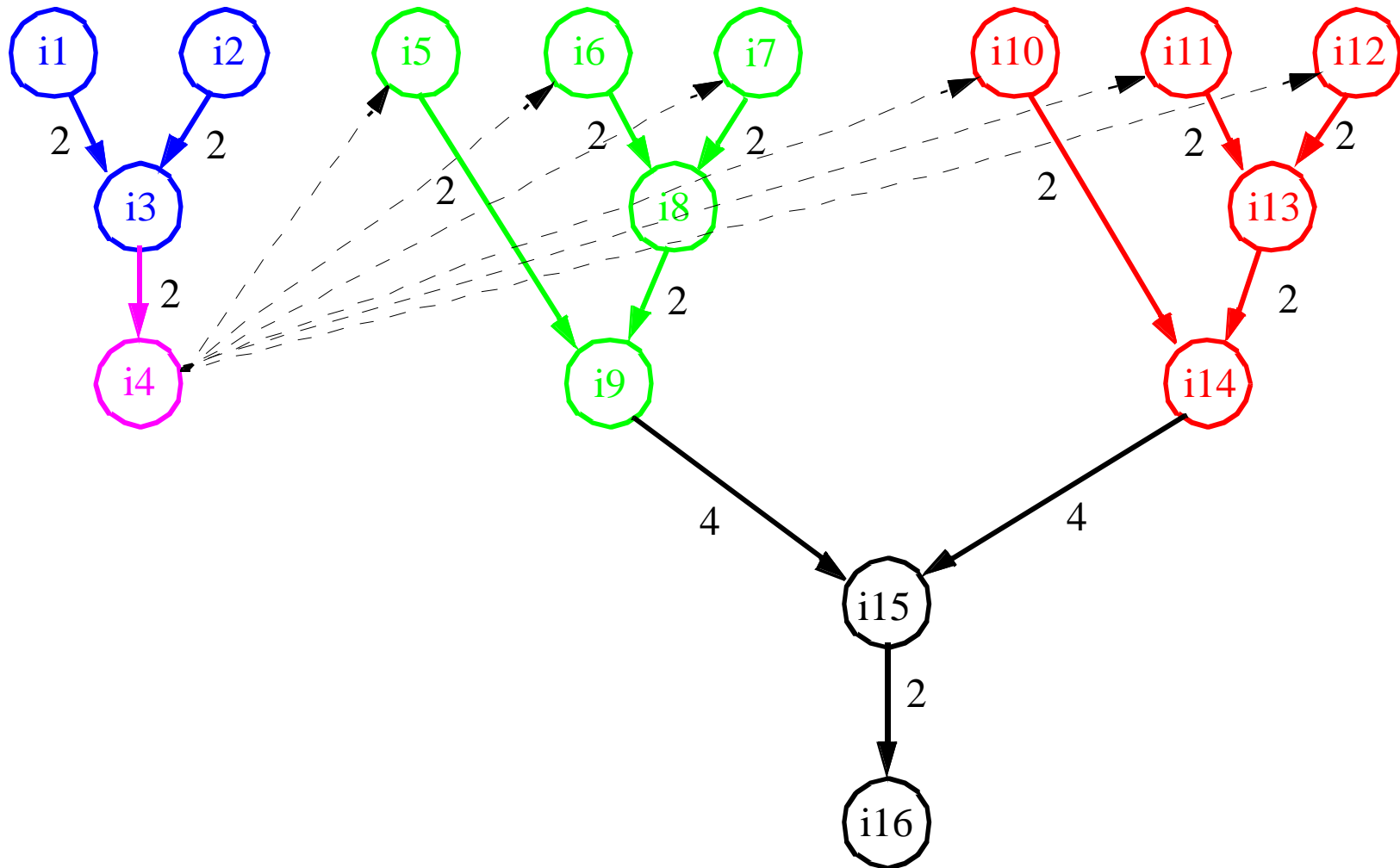
- Sometimes need to copy instructions more than once to ensure correctness on all paths (see C below)



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

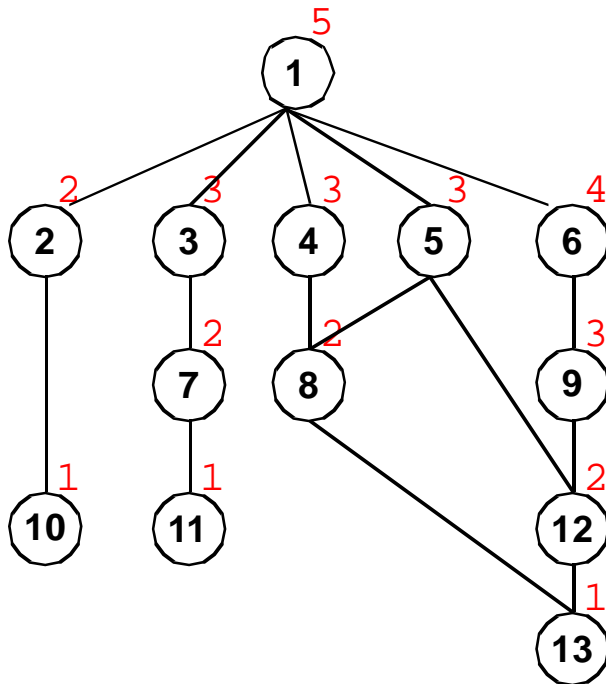
- Assign priority to each instruction
- Initialize ready list that holds all ready instructions
 - Ready = data ready and can be scheduled
- Choose one ready instruction / from ready list with the highest priority
 - Possibly using tie-breaking heuristics
- Insert / into schedule
 - Making sure resource 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

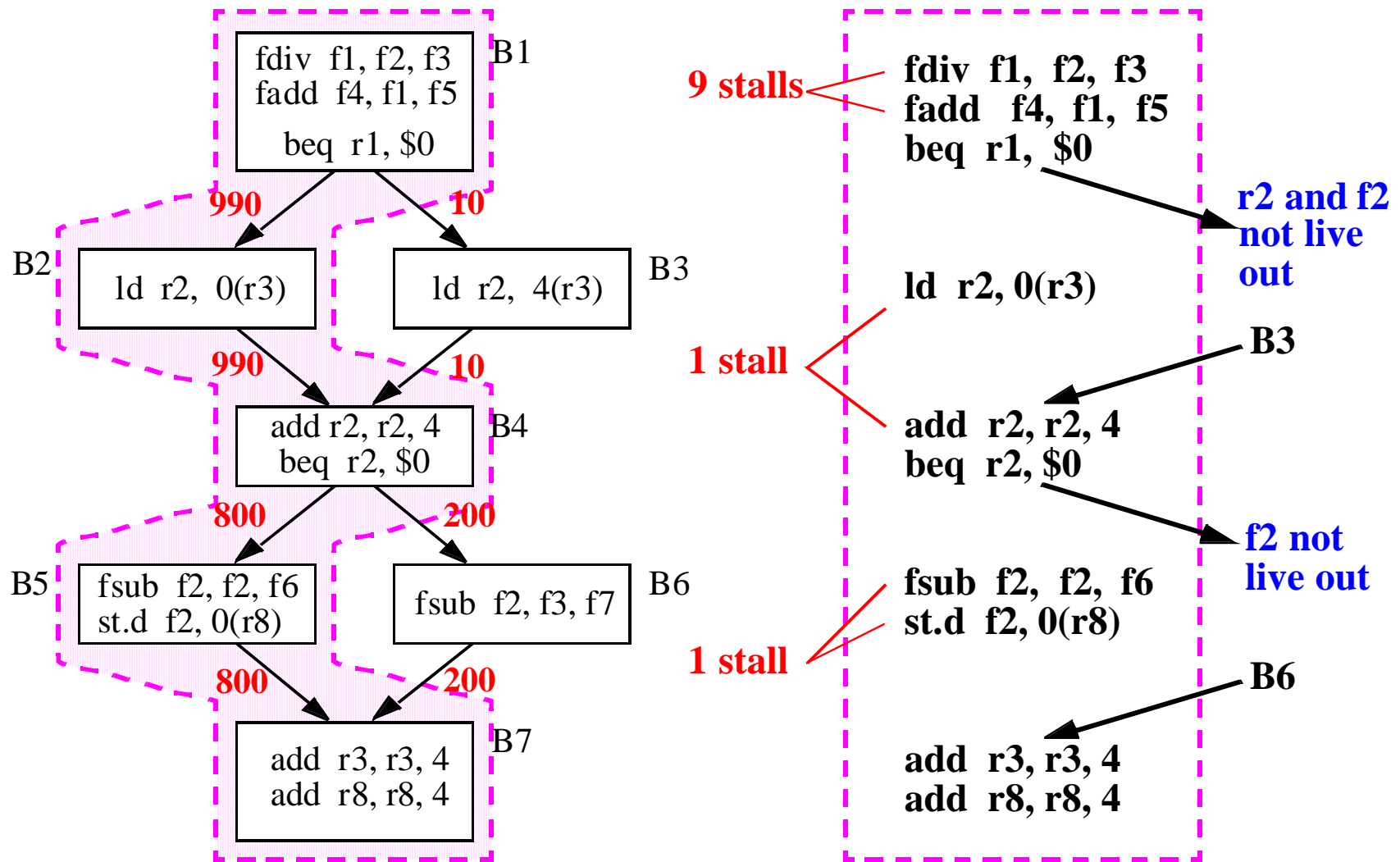
VLIW List Scheduling

- 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

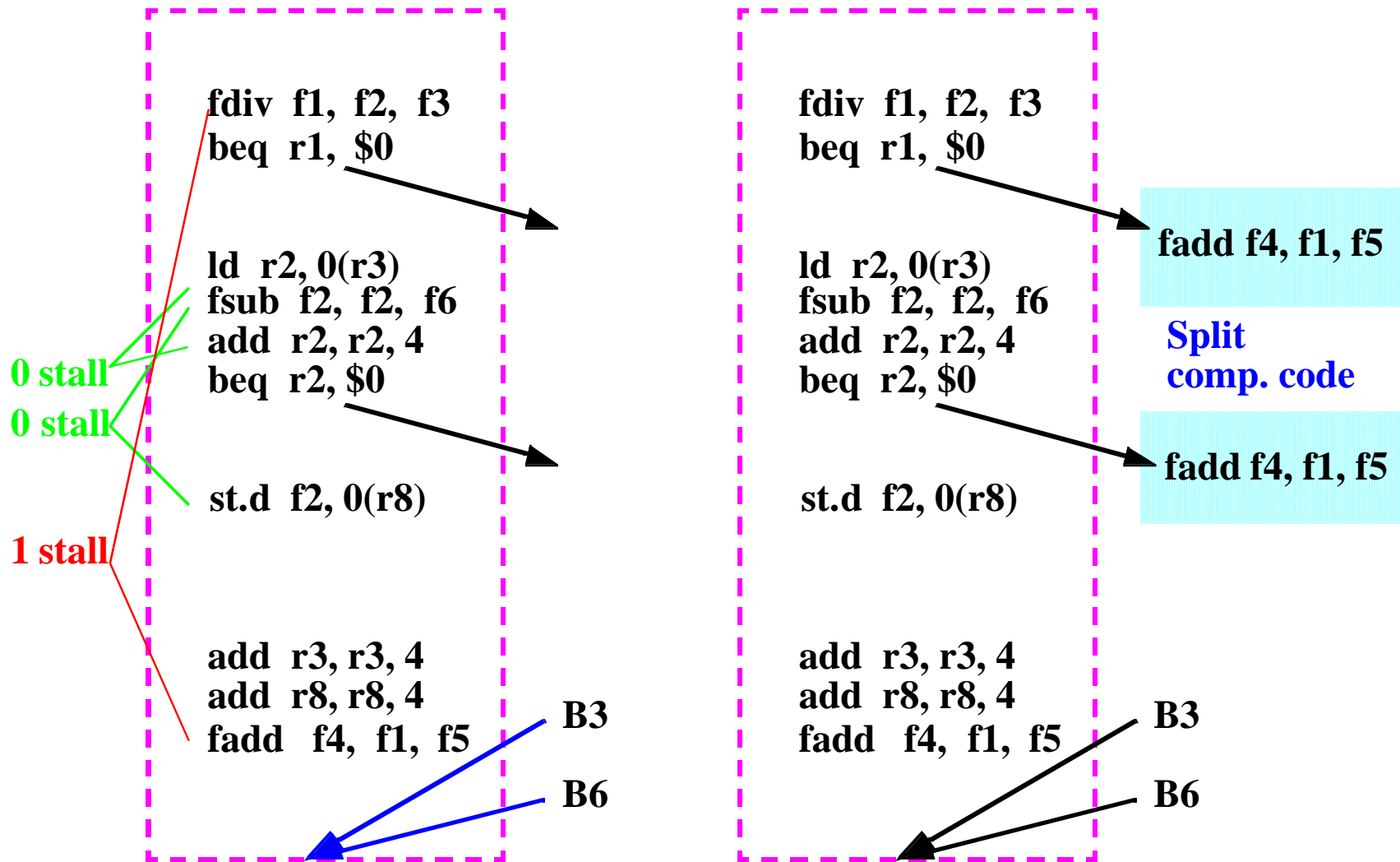


4-wide VLIW				Data Ready List
1				{1}
6	3	4	5	{2,3,4,5,6}
9	2	7	8	{2,7,8,9}
12	10	11		{10,11,12}
13				{13}

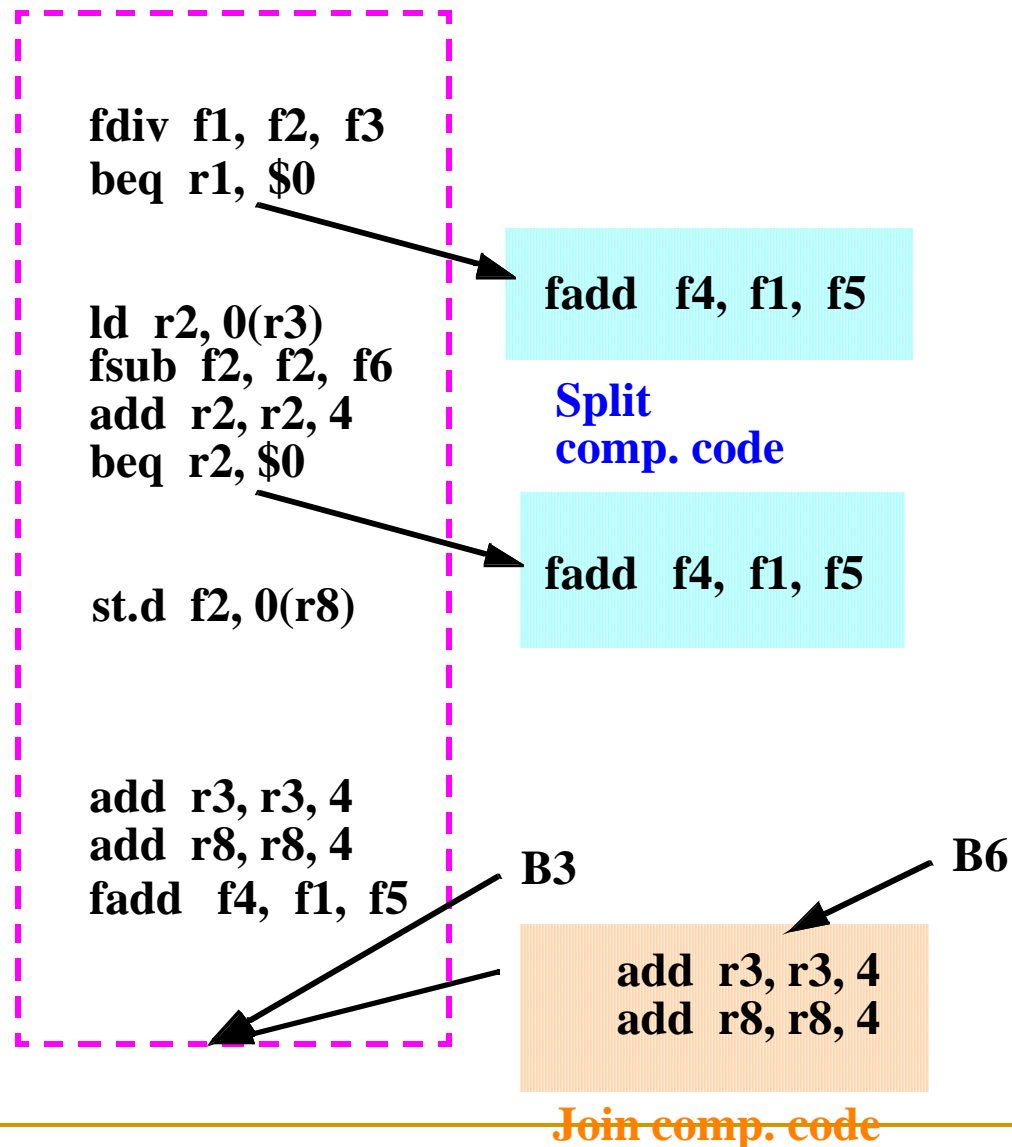
Trace Scheduling Example (I)



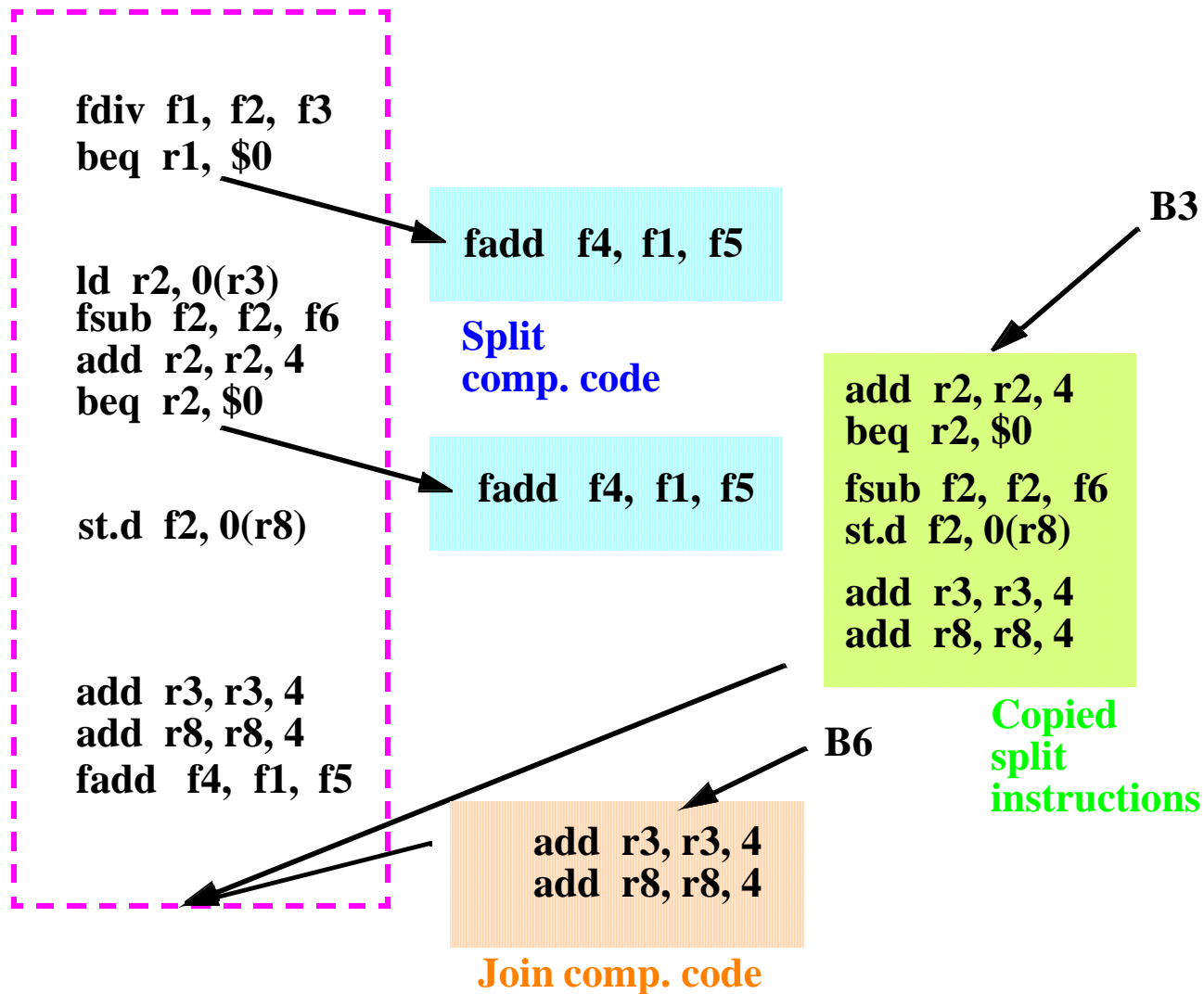
Trace Scheduling Example (II)



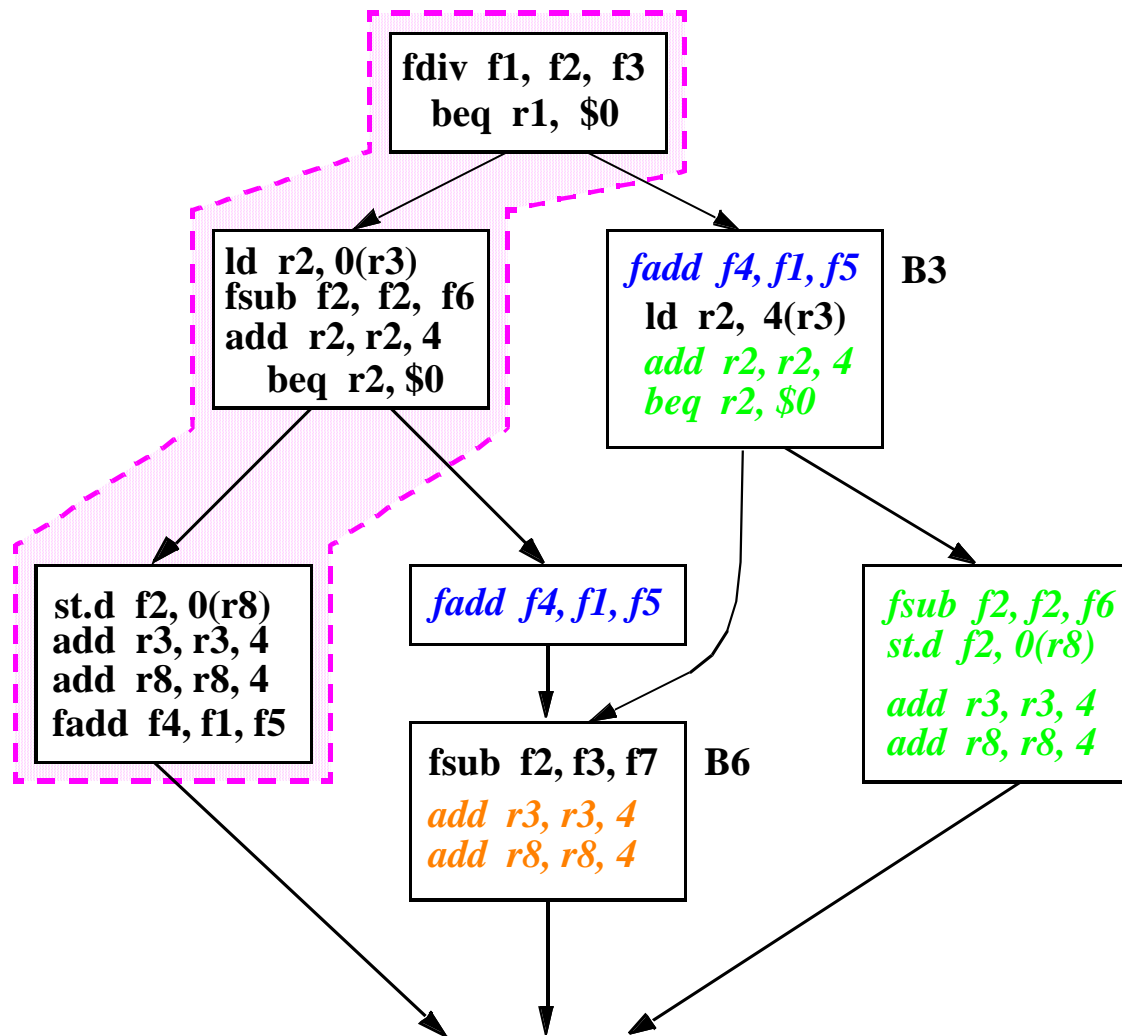
Trace Scheduling Example (III)



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

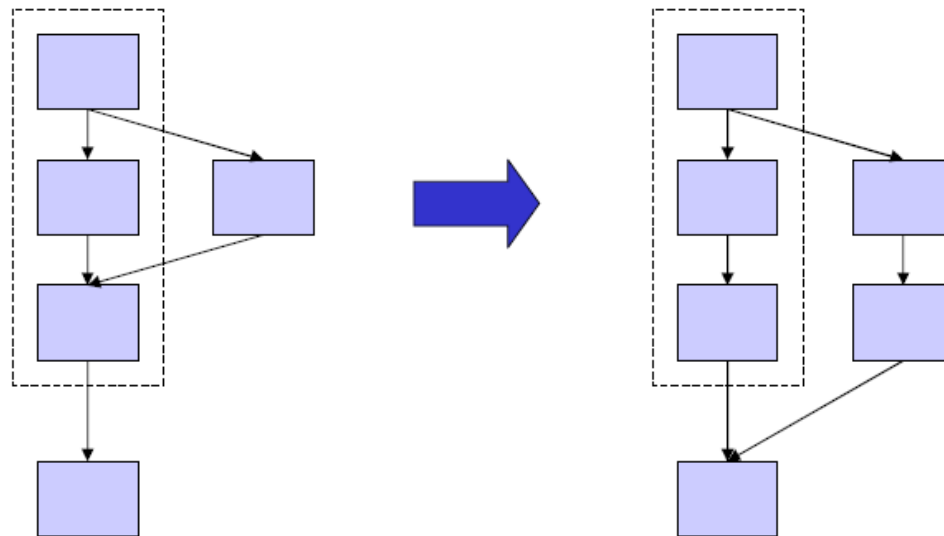
- **Infrequent paths interfere with the frequent path**

- Effectiveness depends on the bias of branches

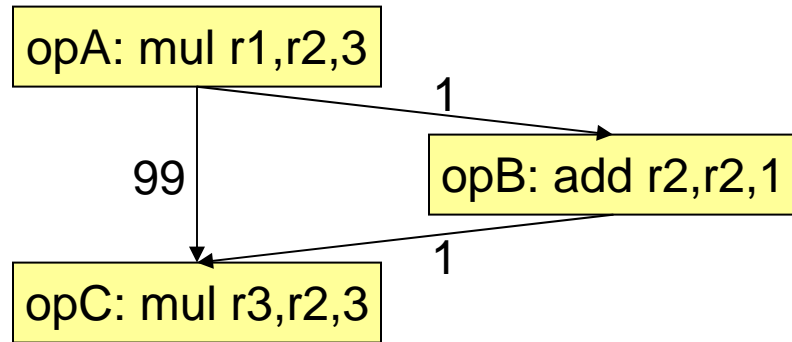
- Unbiased branches → smaller traces → less opportunity for finding independent instructions

Superblock Scheduling

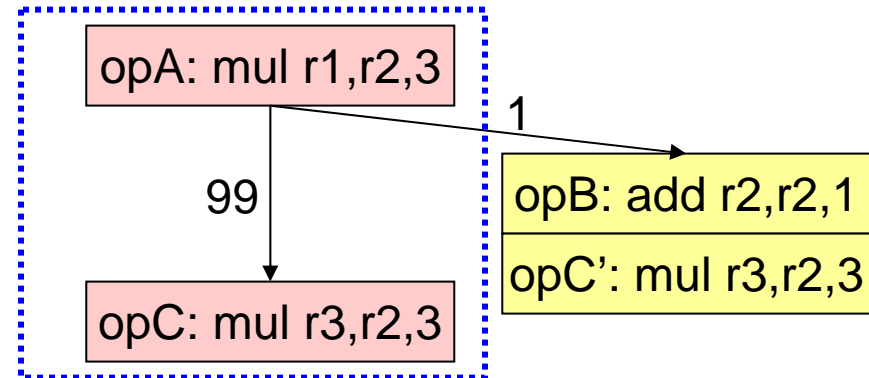
- Trace: multiple entry, multiple exit block
- Superblock: single-entry, multiple exit
 - A trace with side entrances are eliminated
 - Infrequent paths do not interfere with the frequent path
- + More optimization/scheduling opportunity than traces
- + Eliminates “difficult” bookkeeping due to side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



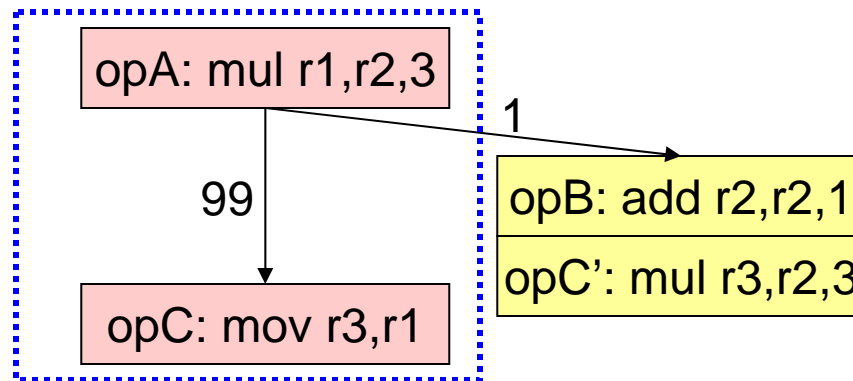
Can You Do This with a Trace?



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

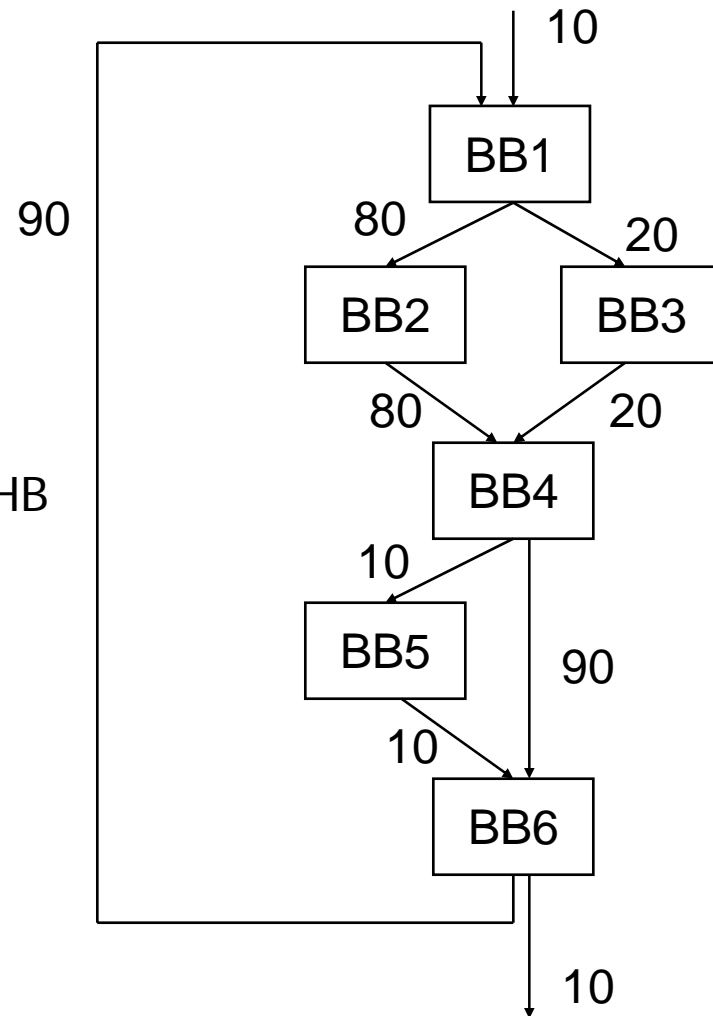
Hyperblock Scheduling

- 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

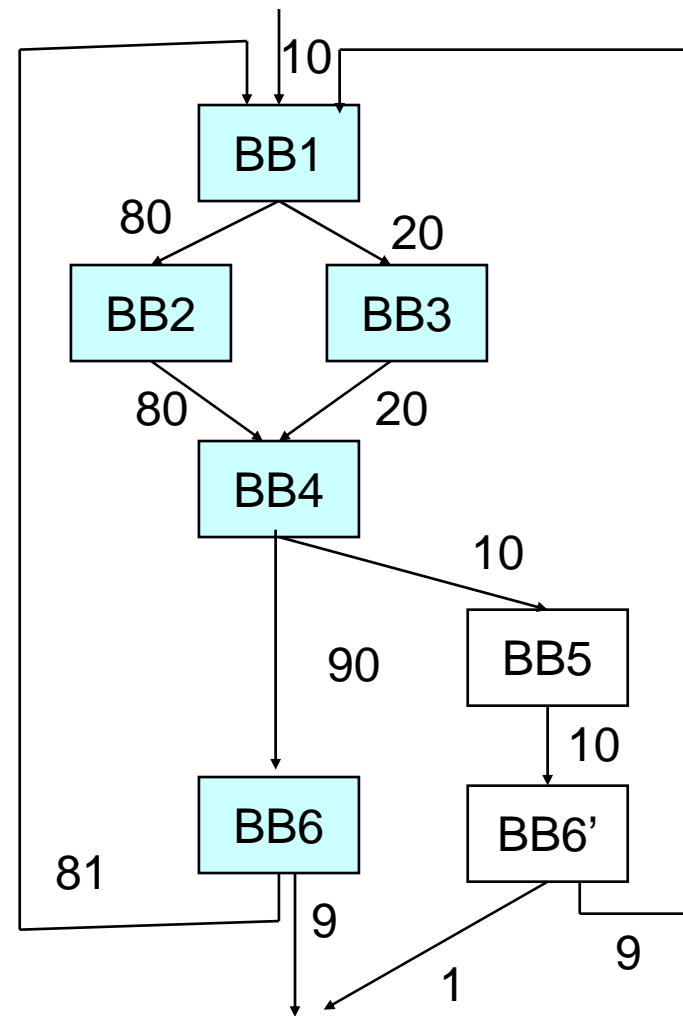
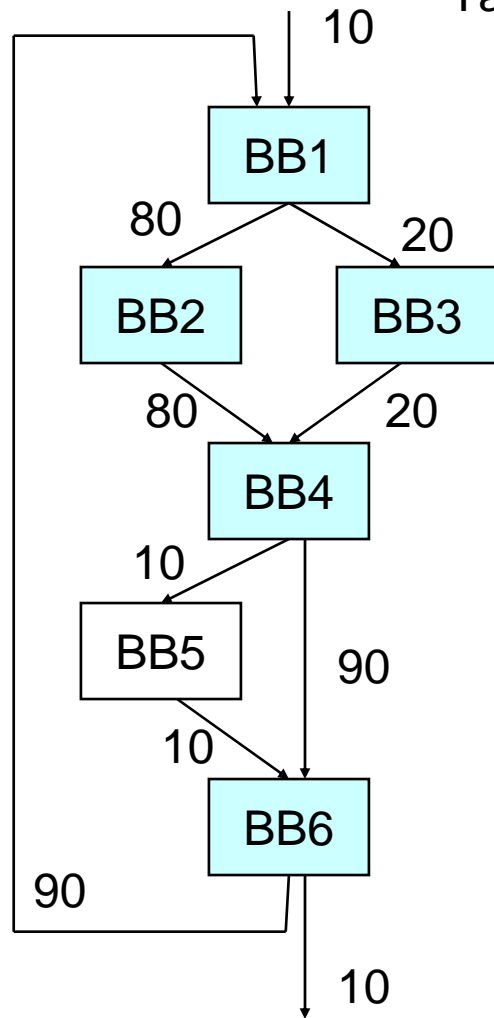
- Block selection
 - Select subset of BBs for inclusion in HB
 - Difficult problem
 - Weighted cost/benefit function
 - Height overhead
 - Resource overhead
 - Dependency overhead
 - Branch elimination benefit
 - Weighted by frequency



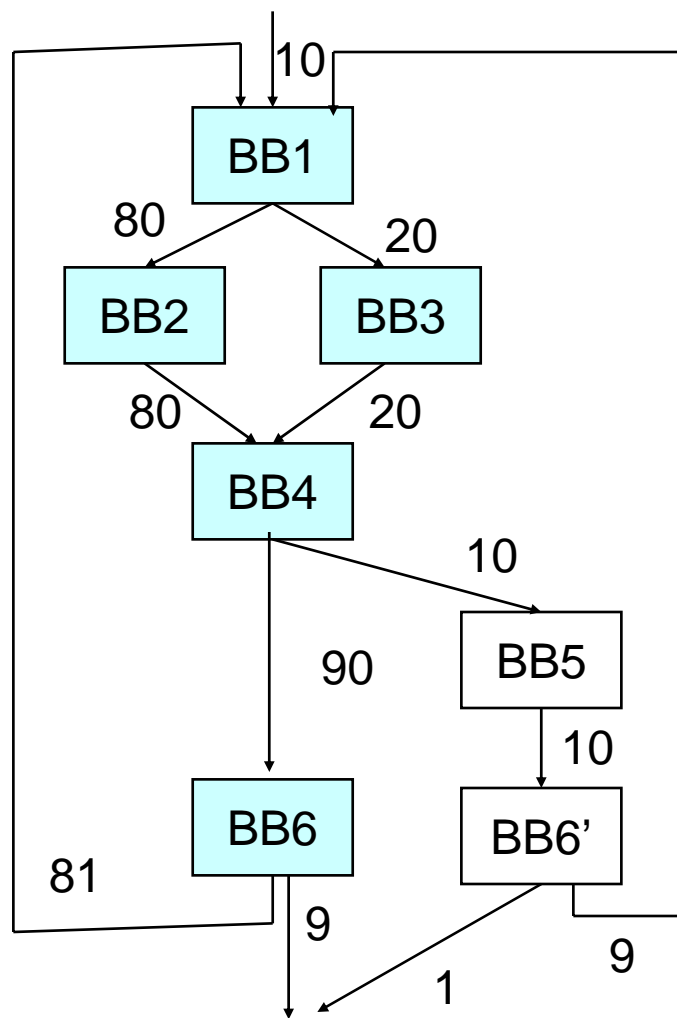
- Mahlke et al., "Effective Compiler Support for Predicated Execution Using the Hyperblock," ISCA 1992.

Hyperblock Formation (II)

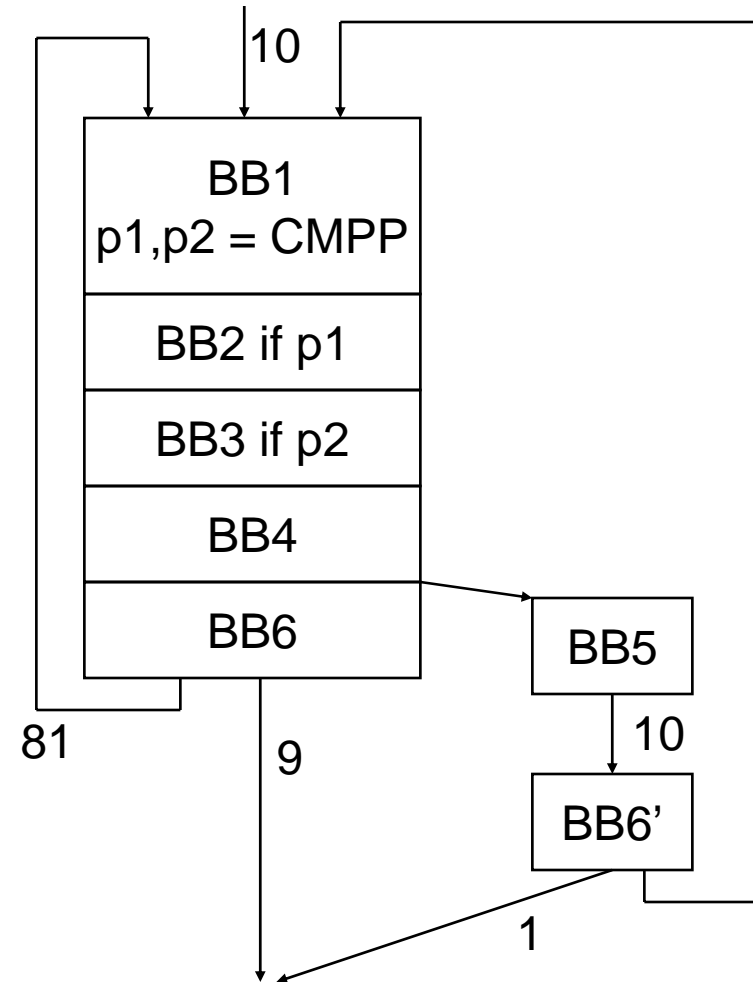
Tail duplication same as with Superblock formation



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)

VLIW Summary

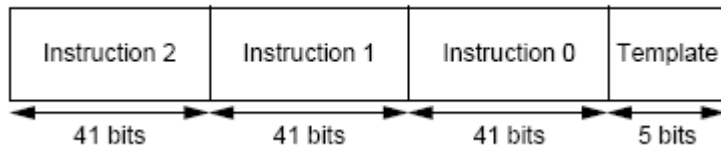
- VLIW simplifies hardware, but requires complex compiler techniques
- VLIW architectures have not been commercially successful in the general-purpose computing market. Why?
 - Too many NOPs (not enough parallelism discovered)
 - Static schedule intimately tied to microarchitecture
 - Code optimized for one generation performs poorly for next
 - No tolerance for variable or long-latency operations (lock step)
- Most compiler optimizations developed for VLIW employed in optimizing compilers (for superscalar compilation)
 - Enable code optimizations

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

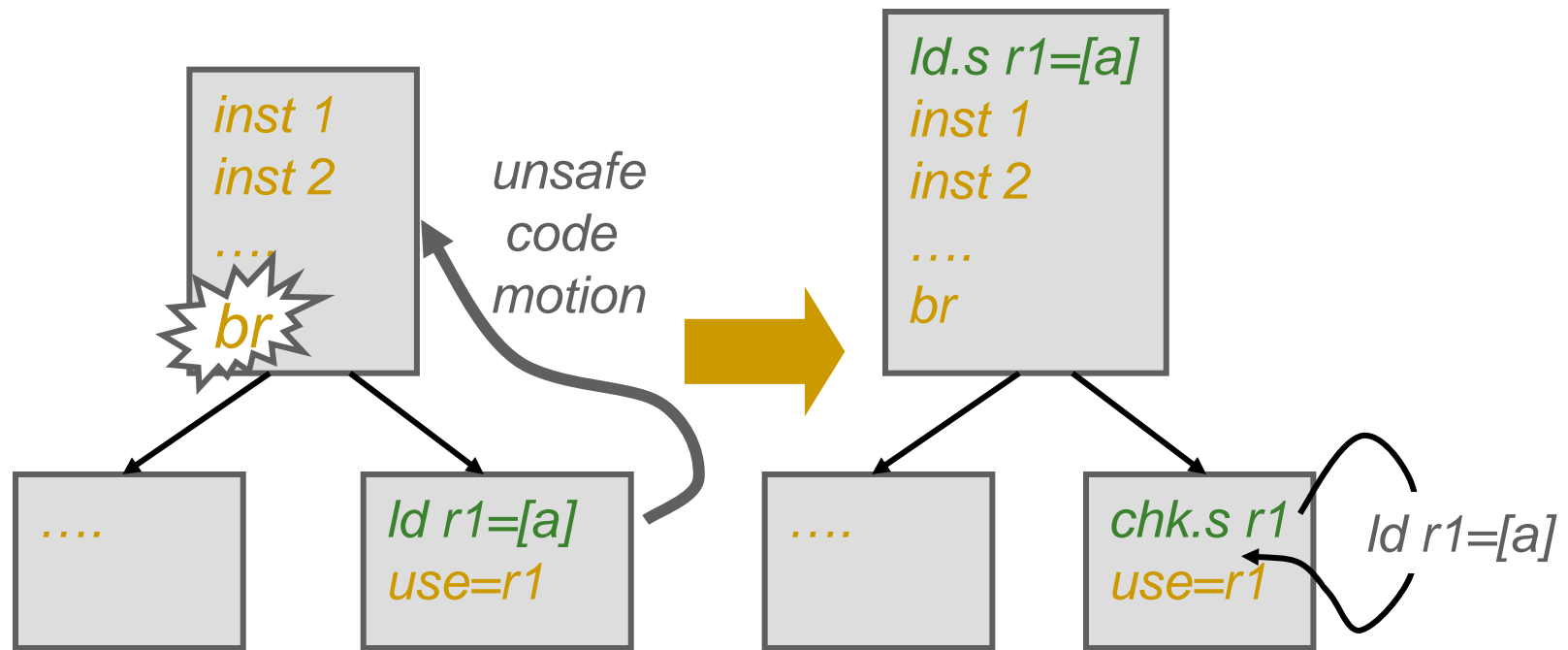
Template Bits

- 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

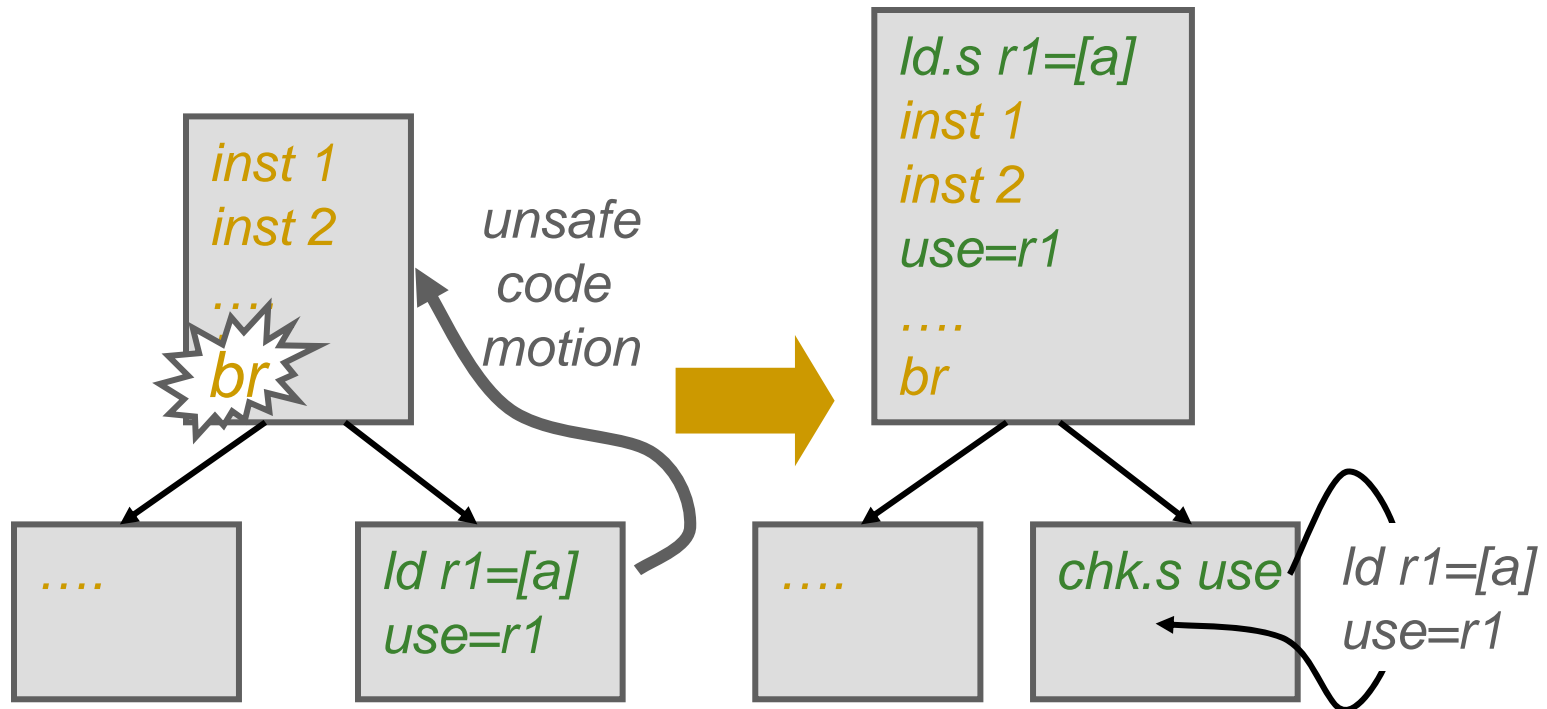
Template	Slot 0	Slot 1	Slot 2
13	M-unit	B-unit	B-unit
14			
15			
16	B-unit	B-unit	B-unit
17	B-unit	B-unit	B-unit
18	M-unit	M-unit	B-unit
19	M-unit	M-unit	B-unit
1A			
1B			
1C	M-unit	F-unit	B-unit
1D	M-unit	F-unit	B-unit
1E			
1F			

Non-Faulting Loads and Exception Propagation



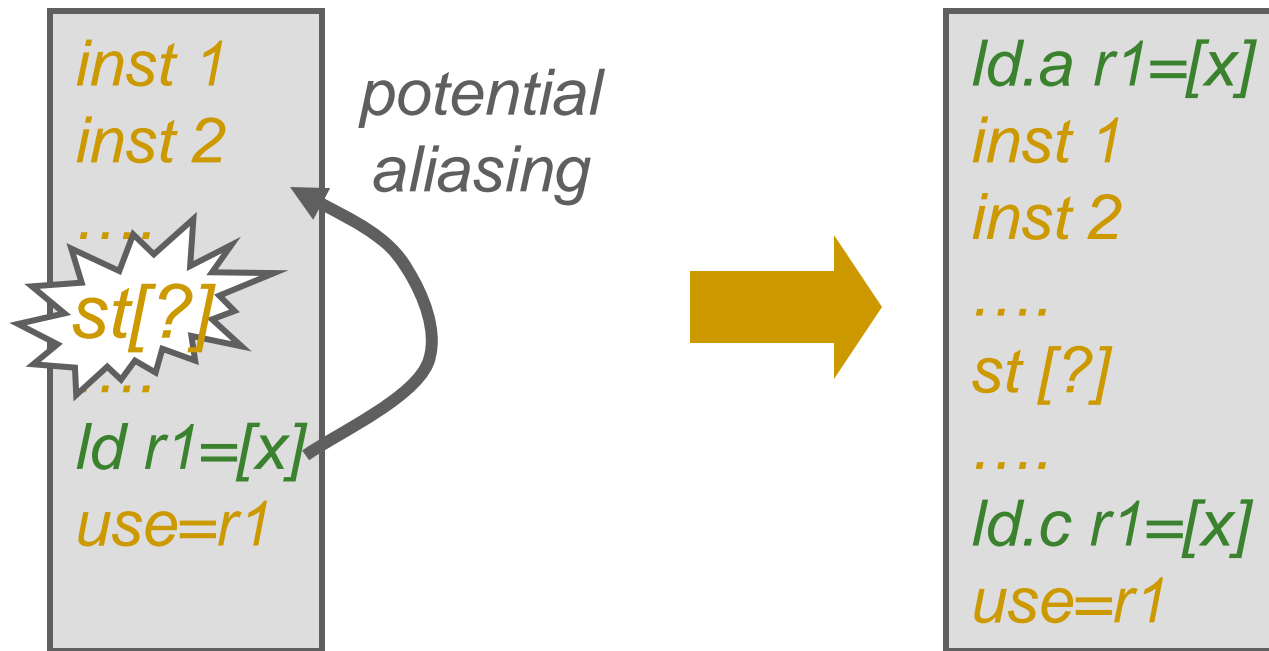
- *ld.s* fetches *speculatively* from memory
i.e. any exception due to *ld.s* is suppressed
- If *ld.s r* did not cause an exception then *chk.s r* is an NOP, else a branch is taken (to some compensation code)

Non-Faulting Loads and Exception Propagation in IA-64



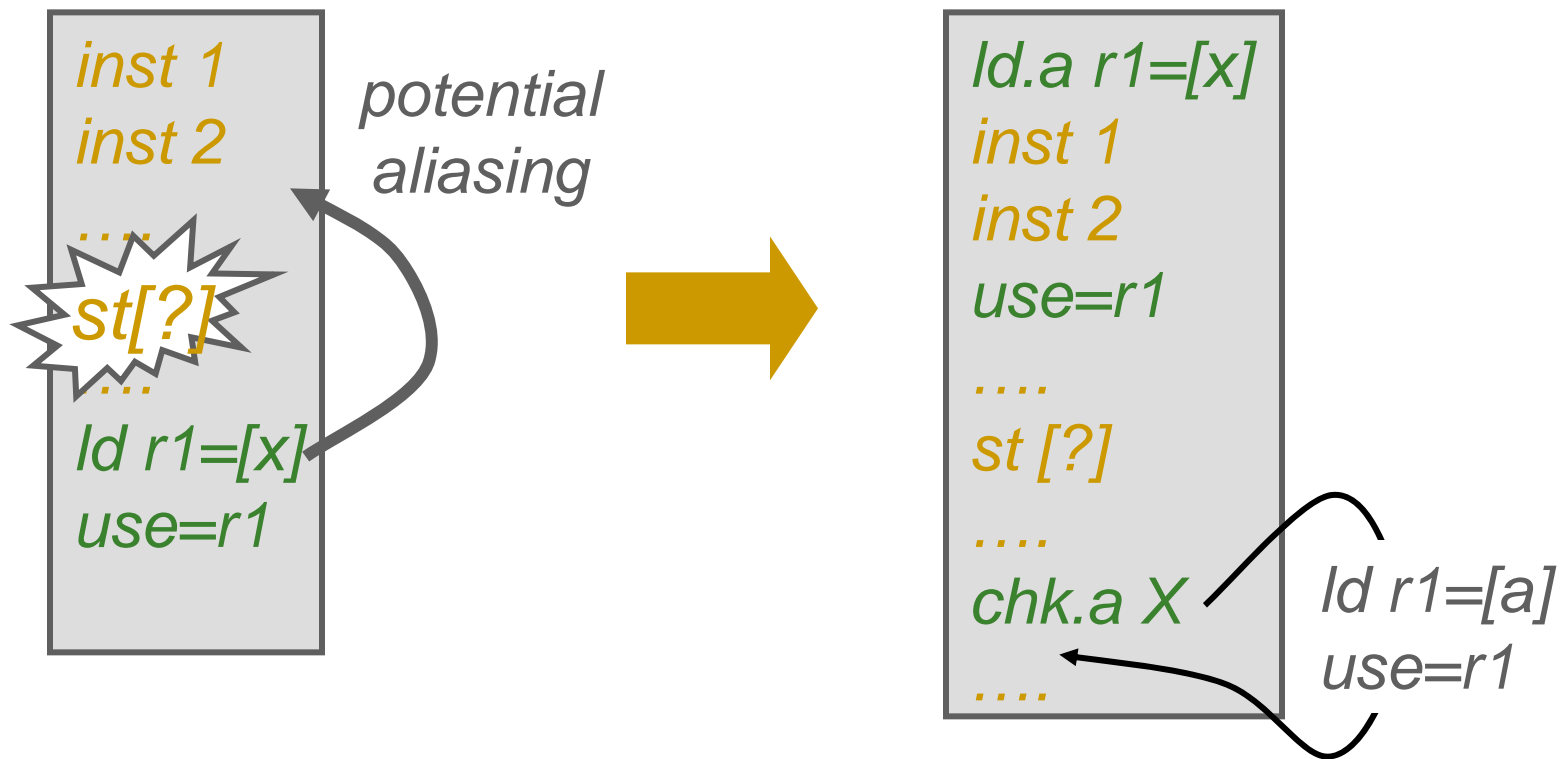
- Speculatively load data can be consumed prior to check
- "speculation" status is propagated with speculated data
- Any instruction that uses a speculative result also becomes speculative itself (i.e. suppressed exceptions)
- *chk.s* checks the entire dataflow sequence for exceptions

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



Midterm II Grade Distribution

