# Computer Architecture: Static Instruction Scheduling

Prof. Onur Mutlu (Editted by Seth)
Carnegie Mellon University

## 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



## 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



## VLIW (Very Long Instruction Word

- Simple hardware with multiple function units
  - Reduced hardware complexity
  - Little or no scheduling done in hardware, e.g., in-order
  - Hopefully, faster clock and less power
- Compiler required to group and schedule instructions (compare to OoO superscalar)
  - Predicated instructions to help with scheduling (trace, etc.)
  - More registers (for software pipelining, etc.)
- Example machines:
  - Multiflow, Cydra 5 (8-16 ops per VLIW)
  - IA-64 (3 ops per bundle)
  - TMS32xxxx (5+ ops per VLIW)
  - Crusoe (4 ops per VLIW)

# Comparison between SS ↔ VLIW

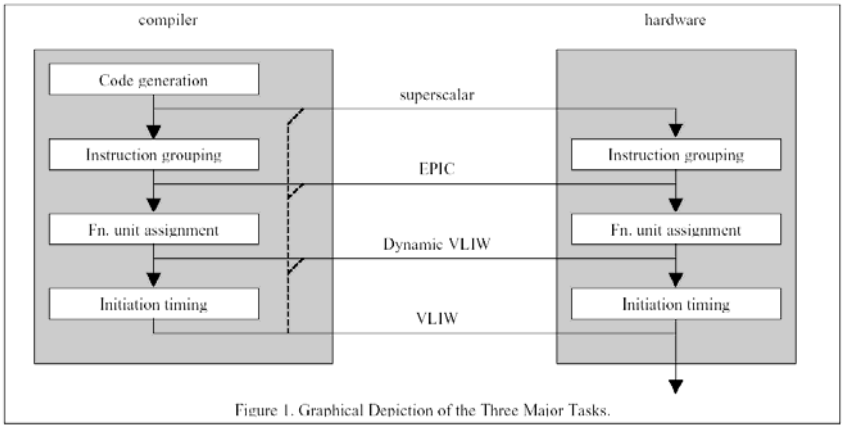
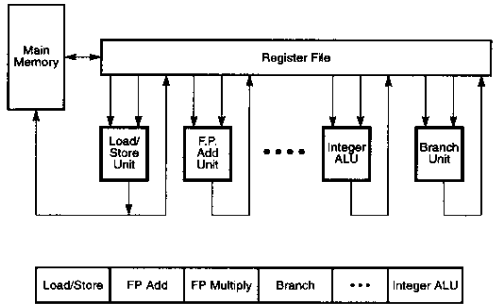


Figure 1. Graphical Depiction of the Three Major Tasks.

From Mark Smotherman, "Understanding EPIC Architectures and Implementations"

# Comparison: CISC, RISC, VLIW

| ARCHITECTURE CHARACTERISTIC | CISC | RISC | VLIW |
|---|---|---|---|
| INSTRUCTION SIZE | Varies | One size, usually 32 bits | One size |
| INSTRUCTION FO RMAT | Field placement varies | Regular, consistent placement of fields | Regular, consistent placement of fields |
| INSTRUCTION SEMANTICS | Varies from simple to complex; possibly many dependent operations per instruction | Almost always one simple operation | Many simple, independent operations |
| REGISTERS | Few, sometimes special | Many, general-purpose | Many, general-purpose |
| MEMORY REFE RENCES | Bundled with operations in many different types of instructions | Not bundled with operations, i.e., load/store architecture | Not bundled with operations, i.e., load/store architecture |
| HARDWARE DESIGN FOCUS | Exploit microcoded implementations | Exploit implementations with one pipeline and & no microcode | Exploit implementations with multiple pipelines, no microcode & no complex dispatch logic |
| PICTURE OF FIVE TYPICAL INSTRU CTIONS = 1 BYTE | | | |



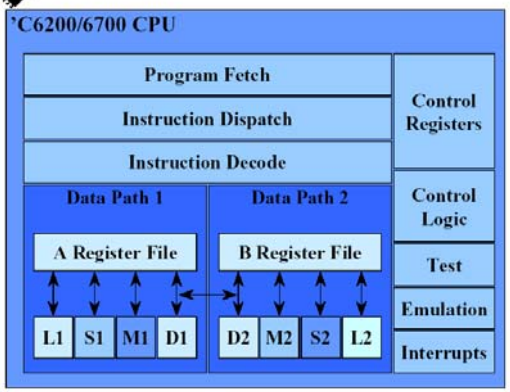
(a) A typical VLIW processor and instruction format

(b) VLIW execution with degree $m = 3$

**Figure 4.14 The architecture of a very long instruction word (VLIW) processor and its pipeline operations.** (Courtesy of Multiflow Computer, Inc., 1987)

# TMS320C6000 CPUs






- Advanced VLIW CPU (VelociTI™)
  - Load-Store RISC
  - Dual Identical Data Paths
    - 4 Functional Units /Each
    - Fetches 8 x 32-Bit Instructions/cycle
  - 2 16 x 16 Integer Multipliers
    - 2 Multiply ACcumulates/cycle (MAC)
  - 32/40-bit arithmetic
  - Byte-Addressable
- 'C6200 Integer CPU
  - 4 ns cycle time
  - 2000 MIPS @ 250 MHz
  - 500 MMACS (Mega MACs per Second)

# 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)

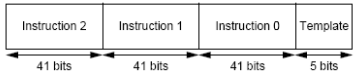
+ No lock-step execution
+ Static reordering of stores and loads + dynamic checking
-- Hardware needs to perform dependency checking (albeit aided by software)
-- Other disadvantages of VLIW still exist

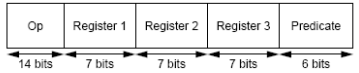
- Huck et al., "Introducing the IA-64 Architecture," IEEE Micro, Sep/Oct 2000.



# 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

```
{ .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



# 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
  - Software Pipelining

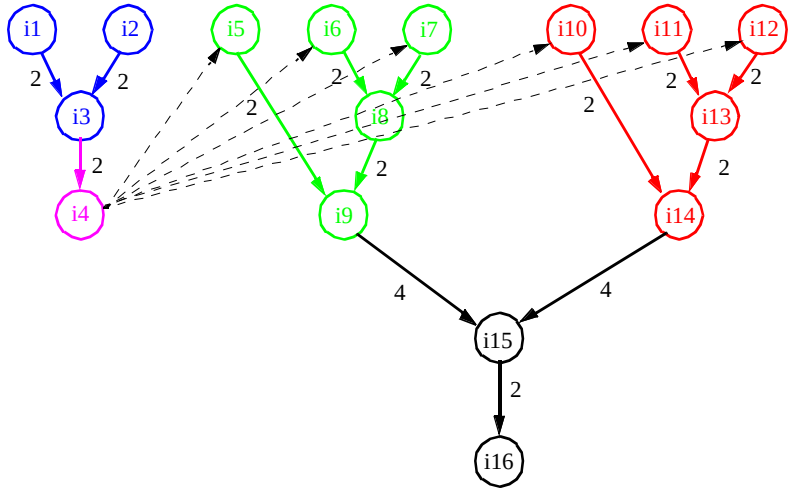
It's all about the compiler and how to **schedule** the instructions to maximize parallelism

# List Scheduling: For 1 basic block

- 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



# Data Precedence Graph





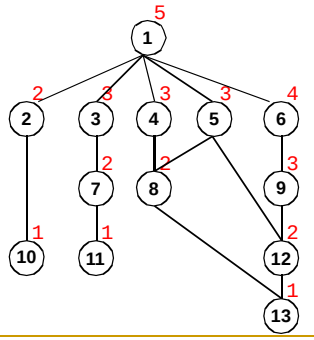

# Instruction Prioritization Heuristics

- 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} |

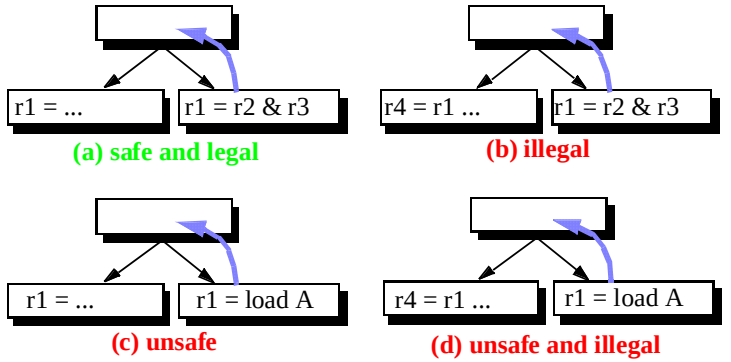
# Extending the scheduling domain

- Basic block is too small to get any real parallelism
- How to extend the basic block?
  - Why do we have basic blocks in the first place?
  - Loops
    - Loop unrolling
    - Software pipelining
  - Non-loops
    - Will almost always involve some speculation
    - And, thus, profiling may be very important



# 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:






# Code Movement Constraints

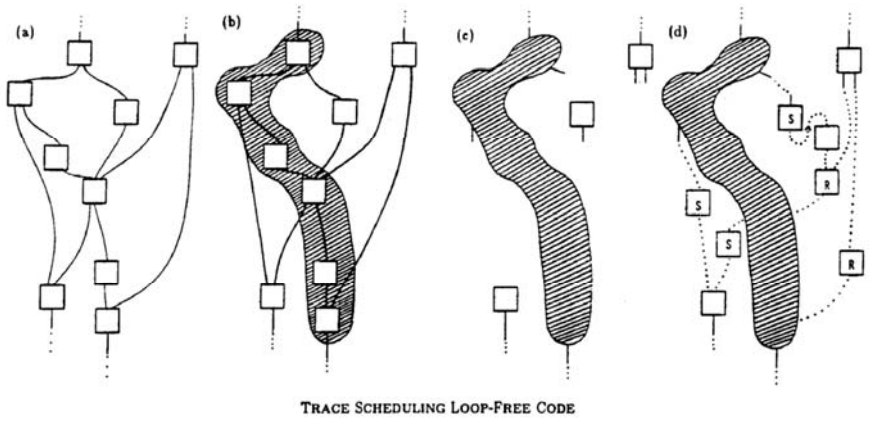
- Downward
  - When moving an operation from a BB to one of its dest BB's,
    - all the other dest basic blocks should still be able to use the result of the operation
    - the other source BB's of the dest BB should not be disturbed
- Upward
  - When moving an operation from a BB to its source BB's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




TRACE SCHEDULING LOOP-FREE CODE

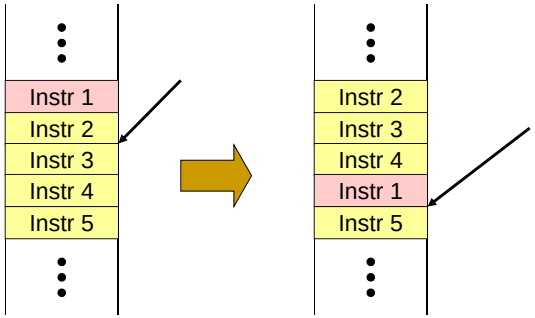


# Trace Scheduling (II)

- 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.



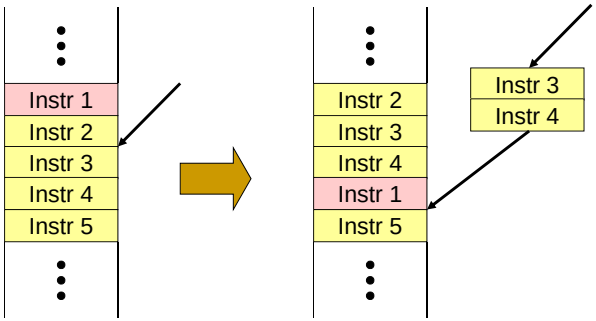
# Trace Scheduling (III)



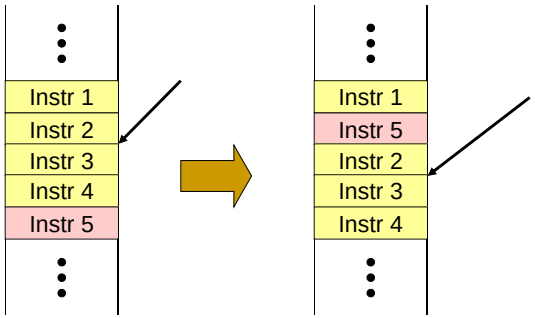

What bookeeping is required when Instr 1 is moved below the side entrance in the trace?



# Trace Scheduling (IV)



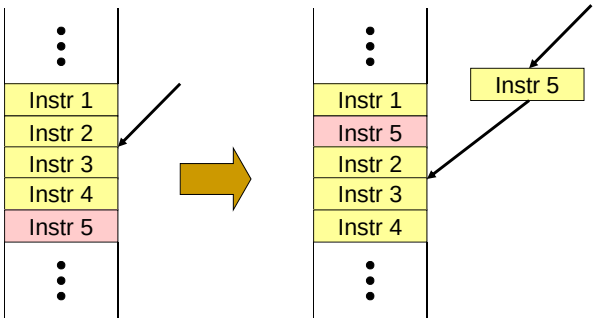
# Trace Scheduling (V)




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



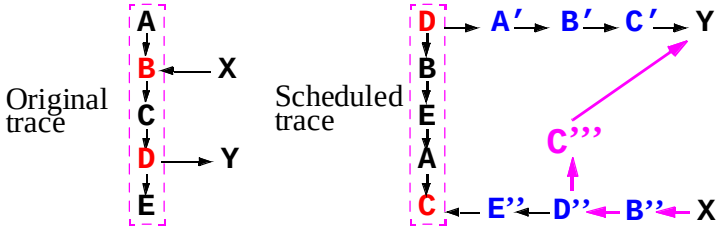
# Trace Scheduling (VI)

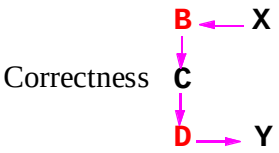





# Trace Scheduling Fixup Code Issues

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



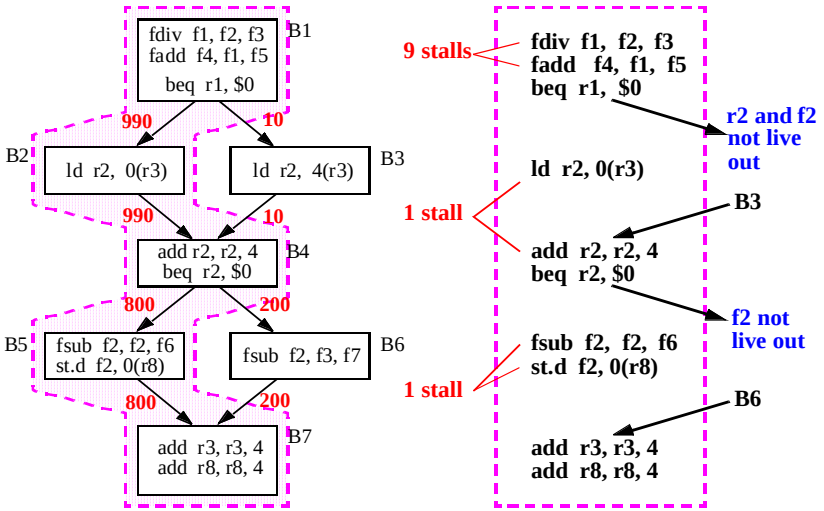






# Trace Scheduling Overview

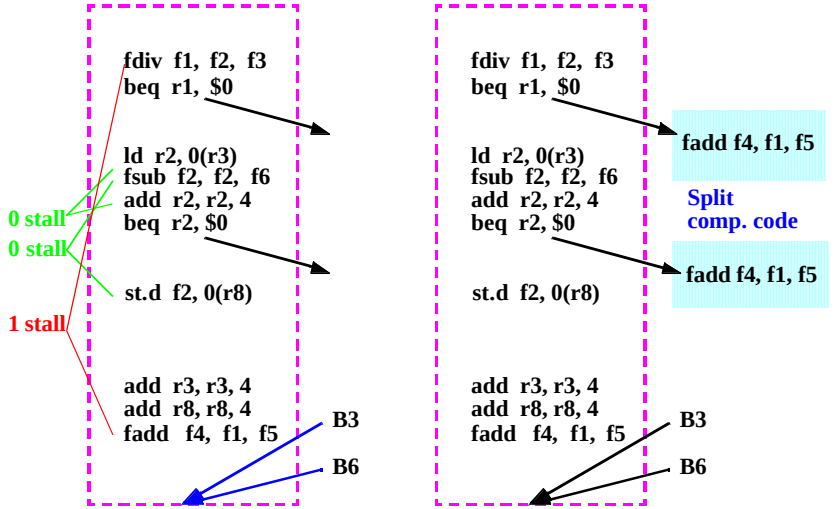
- Trace Selection
  - select seed block (the highest frequency basic block)
  - extend trace (along the highest frequency edges)
    forward (successor of the last block of the trace)
    backward (predecessor of the first block of the trace)
  - don't cross loop back edge
  - bound max_trace_length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

# Trace Scheduling Example (I)



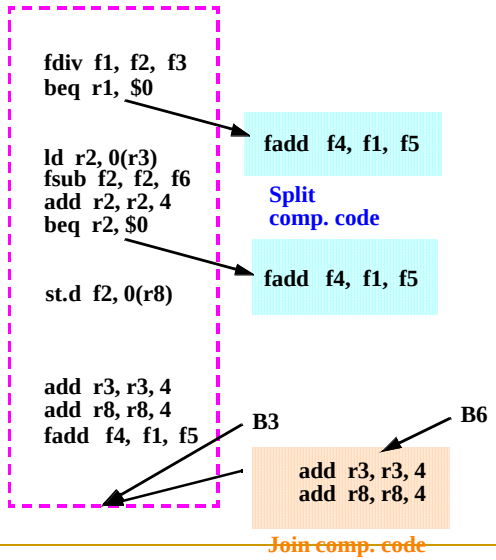



# Trace Scheduling Example (II)



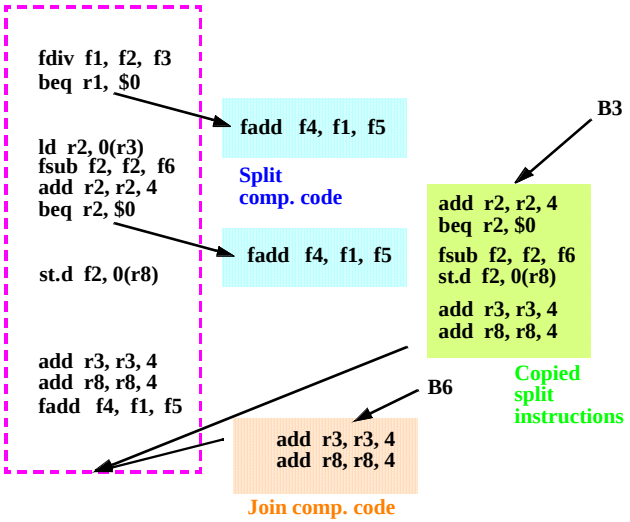



# Trace Scheduling Example (III)

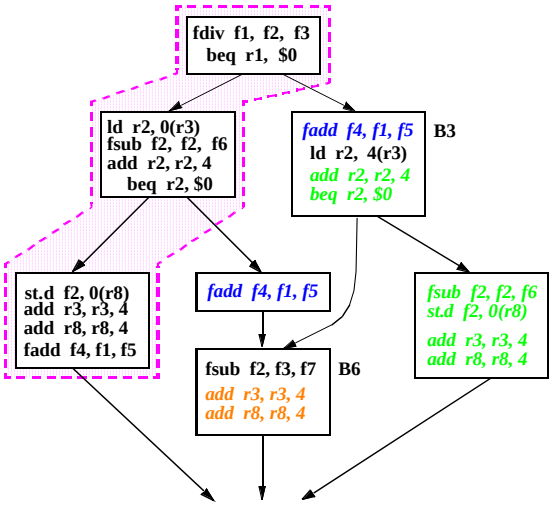





# Trace Scheduling Example (IV)



# Trace Scheduling Example (V)






# Trace Scheduling Tradeoffs

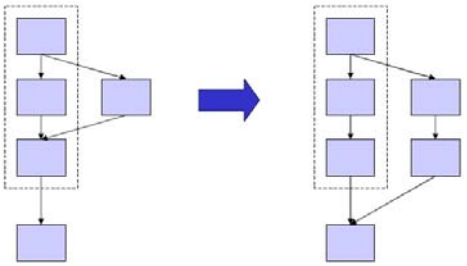
- Advantages
  - + Enables the finding of more independent instructions → fewer NOPs in a VLIW instruction
- Disadvantages
  - -- Profile dependent
    - -- What if dynamic path deviates from trace → lots of NOPs in the VLIW instructions
  - -- Code bloat and additional fix-up code executed
    - -- 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

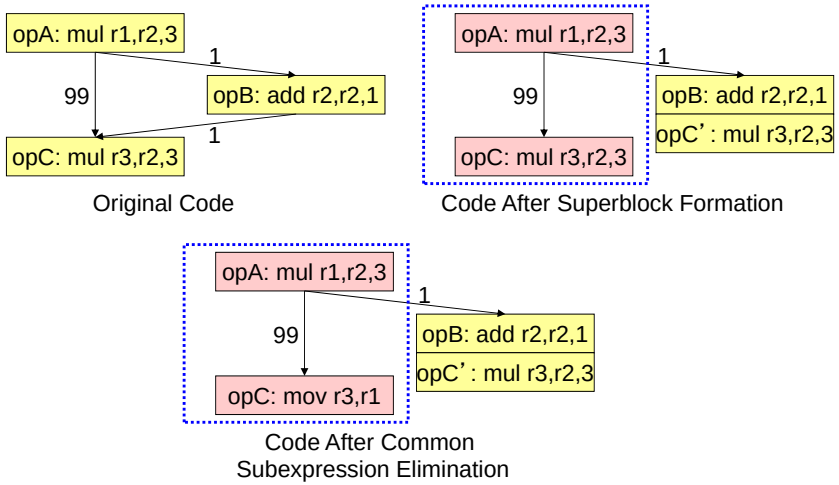
+ More optimization/scheduling opportunity than traces
+ Eliminates "difficult" bookkeeping due to side entrances



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



# Can You Do This with a Trace?



# 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 scheduled block size
- Disadvantages

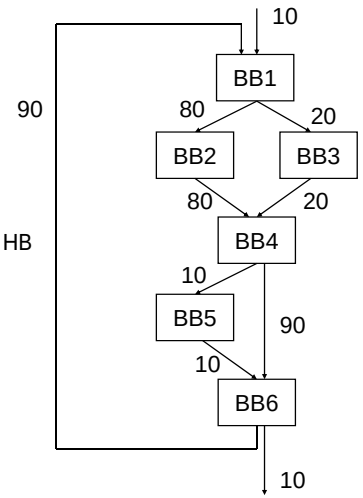
  -- Requires predicated execution support

  -- 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

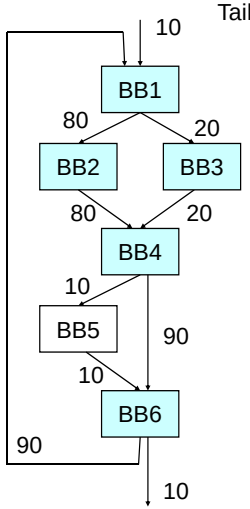


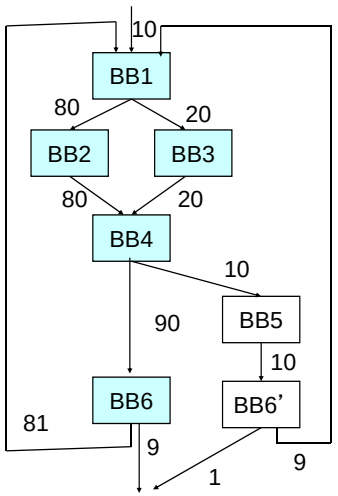

- Mahlke et al., "Effective Compiler Support for Predicated Execution Using the Hyperblock," MICRO 1992.



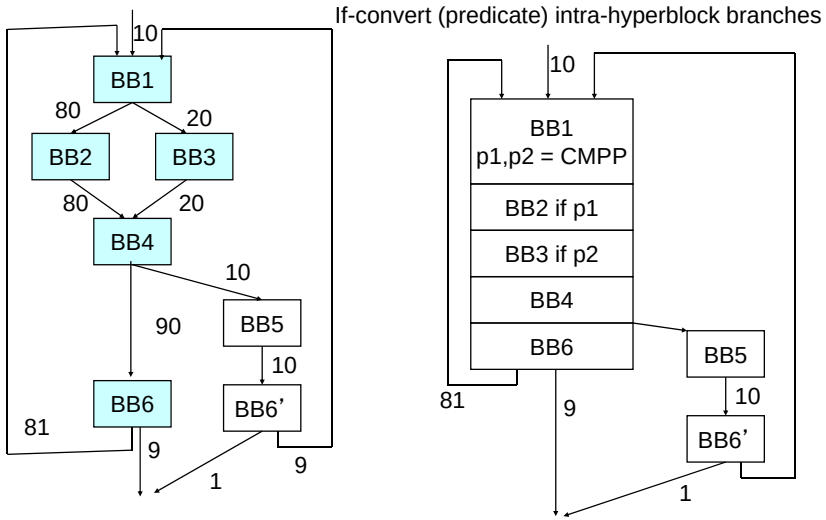
# Hyperblock Formation (II)

Tail duplication same as with Superblock formation



# Hyperblock Formation (III)



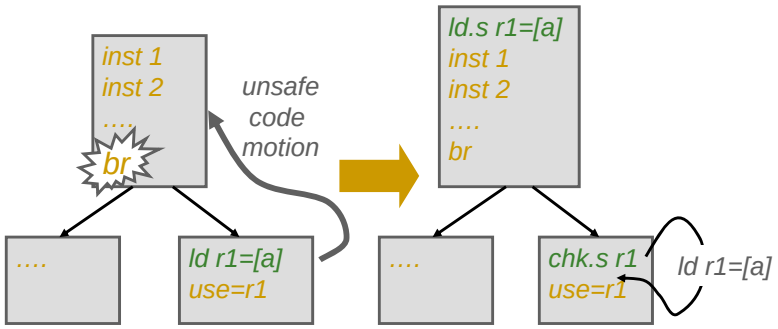



# Can We Do Better?

- 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)



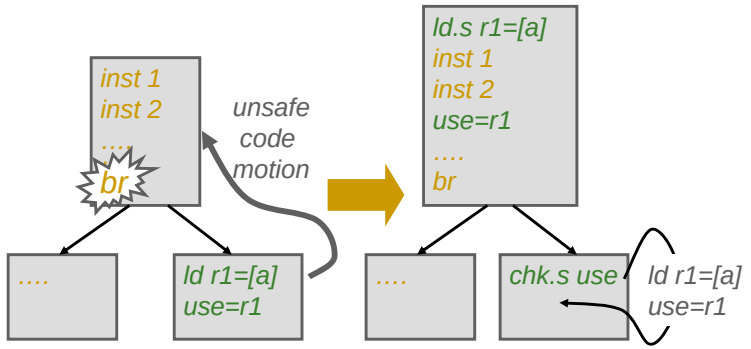
# Non-Faulting Loads and Exception Propagation




- *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)

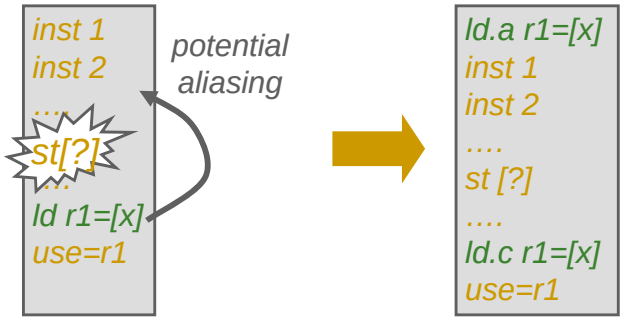


# Non-Faulting Loads and Exception Propagation in IA-64




- 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

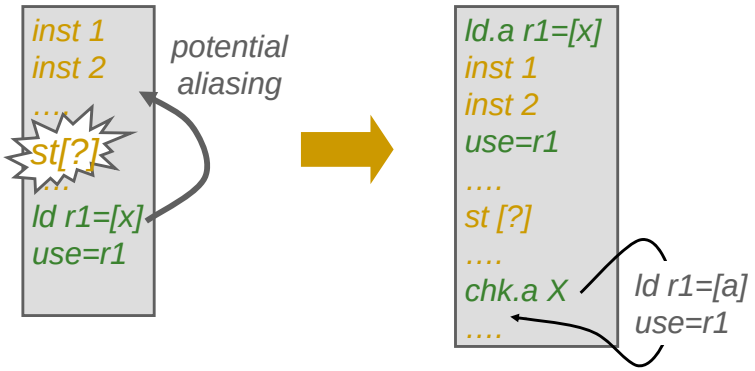
# Aggressive ST-LD Reordering in IA-64




- *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






# 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



# What about loops?

- Unrolling
- Software pipelining

# 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



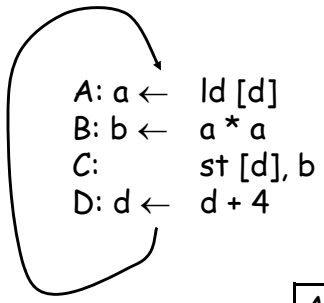
# Software Pipelining

- Software pipelining is an instruction scheduling technique that reorders the instructions in a loop.
  - Possibly moving instructions from one iteration to the previous or the next iteration.
  - Very large improvements in running time are possible.
- The first serious approach to software pipelining was presented by Aiken & Nicolau.
  - Aiken's 1988 Ph.D. thesis.
  - Impractical as it ignores resource hazards (focusing only on data-dependence constraints).
    - But sparked a large amount of follow-on research.

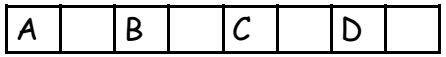


# Goal of SP

- Increase distance between dependent operations by moving destination operation to a later iteration




Assume all have latency of 2

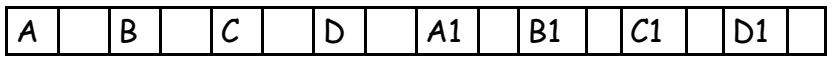





# Can we decrease the latency?

- Lets unroll

```
A:  a ← ld [d]
B:  b ← a * a
C:    st [d], b
D:  d ← d + 4
A1: a ← ld [d]
B1: b ← a * a
C1:   st [d], b
D1: d ← d + 4
```



# Rename variables

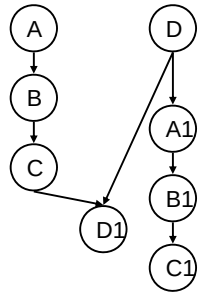
```
A:  a ← ld [d]
B:  b ← a * a
C:      st [d], b
D:  d1 ← d + 4
A1: a1 ← ld [d1]
B1: b1 ← a1 * a1
C1:     st [d1], b1
D1: d ← d1 + 4
```

| A | | B | | C | | D | | A1 | | B1 | | C1 | | D1 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|



# Schedule

```
A:  a ← ld [d]
B:  b ← a * a
C:      st [d], b
D:  d1 ← d + 4
A1: a1 ← ld [d1]
B1: b1 ← a1 * a1
C1:     st [d1], b1
D1: d ← d1 + 4
```

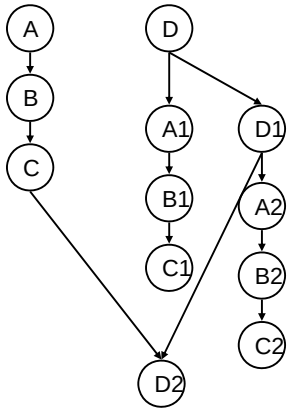



| A | | B | | C | | D1 | |
|---|---|---|---|---|---|---|---|
| D | | A1 | | B1 | | C1 | |



# Unroll Some More

```
A:  a ← ld [d]
B:  b ← a * a
C:      st [d], b
D:  d1 ← d + 4
A1: a1 ← ld [d1]
B1: b1 ← a1 * a1
C1:     st [d1], b1
D1: d2 ← d1 + 4
A2: a2 ← ld [d2]
B2: b2 ← a2 * a2
C2:     st [d2], b2
D2: d ← d2 + 4
```

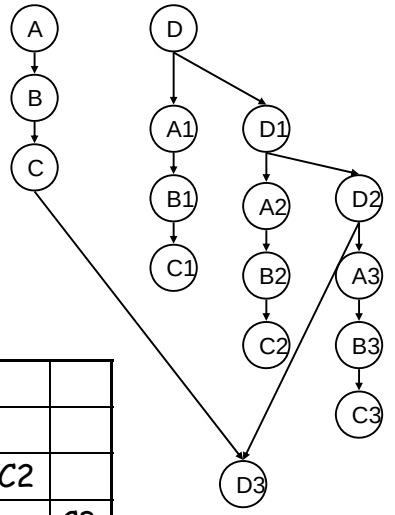



| A | | B | | C | | D2 | |
|---|---|---|---|---|---|---|---|
| D | | A1 | | B1 | | C1 | |
| | D1 | | A2 | | B2 | | C2 |



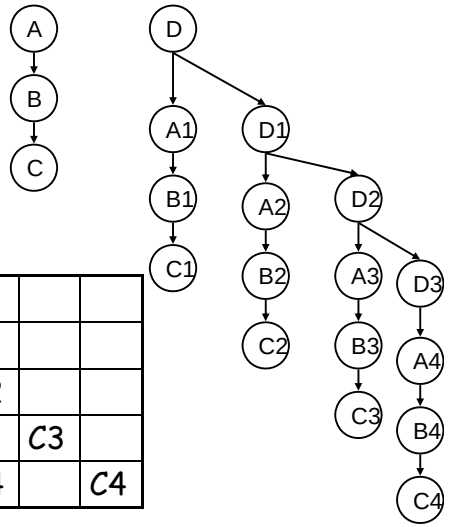
# Unroll Some More

```
A:  a ← ld [d]
B:  b ← a * a
C:      st [d], b
D:  d1 ← d + 4
A1: a1 ← ld [d1]
B1: b1 ← a1 * a1
C1:     st [d1], b1
D1: d2 ← d1 + 4
A2: a2 ← ld [d2]
B2: b2 ← a2 * a2
C2:     st [d2], b2
D2: d ← d2 + 4
```




| A | | B | | C | | D3 | | |
|---|---|---|---|---|---|---|---|---|
| D | | A1 | | B1 | | C1 | | |
| | D1 | | A2 | | B2 | | C2 | |
| | | D2 | | A3 | | B3 | | C3 |

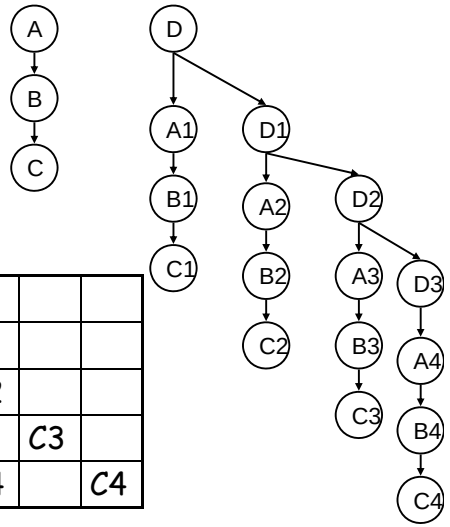
# One More Time




| A | | B | | C | | D4 | | | |
|---|---|---|---|---|---|---|---|---|---|
| D | | A1 | | B1 | | C1 | | | |
| | D1 | | A2 | | B2 | | C2 | | |
| | | D2 | | A3 | | B3 | | C3 | |
| | | | D3 | | A4 | | B4 | | C4 |



# Can Rearrange

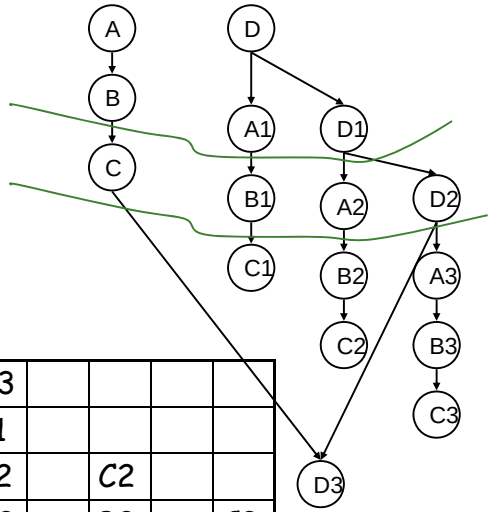



| A | | B | | C | | D4 | | | |
|---|---|---|---|---|---|---|---|---|---|
| D | | A1 | | B1 | | C1 | | | |
| | D1 → | | A2 | | B2 | | C2 | | |
| | | D2 → | | A3 | | B3 | | C3 | |
| | | | D3 | | A4 | | B4 | | C4 |



# Rearrange

```
A:  a ←   ld [d]
B:  b ←   a * a
C:        st [d], b
D:  d1 ←  d + 4
A1: a1 ←  ld [d1]
B1: b1 ←  a1 * a1
C1:       st [d1], b1
D1: d2 ←  d1 + 4
A2: a2 ←  ld [d2]
B2: b2 ←  a2 * a2
C2:       st [d2], b2
D2: d ←   d2 + 4
```

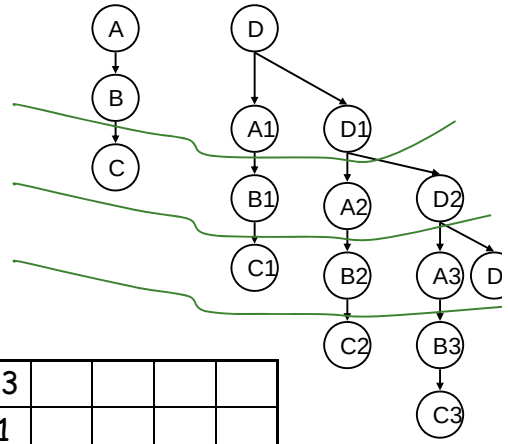



| A | | B | | C | | D3 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| D | | A1 | | B1 | | C1 | | | | |
| | | D1 | | A2 | | B2 | | C2 | | |
| | | | | D2 | | A3 | | B3 | | C3 |



# Rearrange

```
A:  a ←   ld [d]
B:  b ←   a * a
C:        st [d], b
D:  d1 ←  d + 4
A1: a1 ←  ld [d1]
B1: b1 ←  a1 * a1
C1:       st [d1], b1
D1: d2 ←  d1 + 4
A2: a2 ←  ld [d2]
B2: b2 ←  a2 * a2
C2:       st [d2], b2
D2: d ←   d2 + 4
```




| A | | B | | C | | D3 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| D | | A1 | | B1 | | C1 | | | | |
| | | D1 | | A2 | | B2 | | C2 | | |
| | | | | D2 | | A3 | | B3 | | C3 |

# SP Loop

```
A:   a ←    ld [d]
B:   b ←    a * a
D:   d1 ←   d + 4
A1:  a1 ←   ld [d1]
D1:  d2 ←   d1 + 4

C:          st [d], b
B1:  b1 ←   a1 * a1
A2:  a2 ←   ld [d2]
D2:  d ←    d2 + 4

B2:  b2 ←   a2 * a2
C1:         st [d1], b1
D3:  d2 ←   d1 + 4
C2:         st [d2], b2
```

Prolog

Body

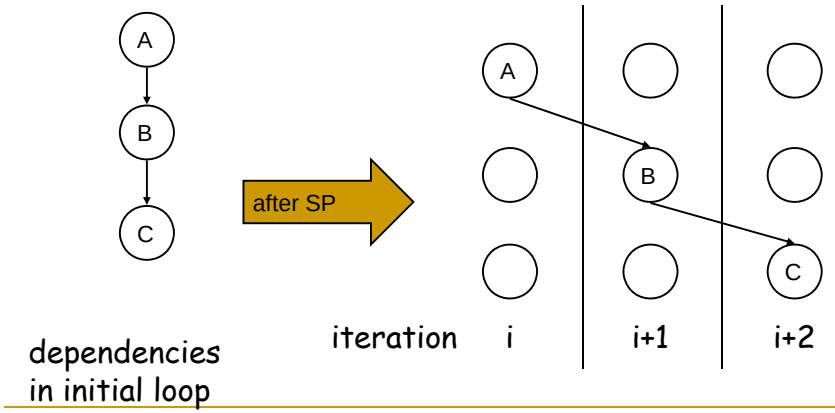
Epilog

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| A | | B | | C | C | C | D3 | | |
| D | | A1 | | B1 | B1 | B1 | C1 | | |
| | | D1 | | A2 | A2 | A2 | B2 | | C2 |
| | | | | D2 | D2 | D2 | | | |



# Goal of SP

- Increase distance between dependent operations by moving destination operation to a later iteration





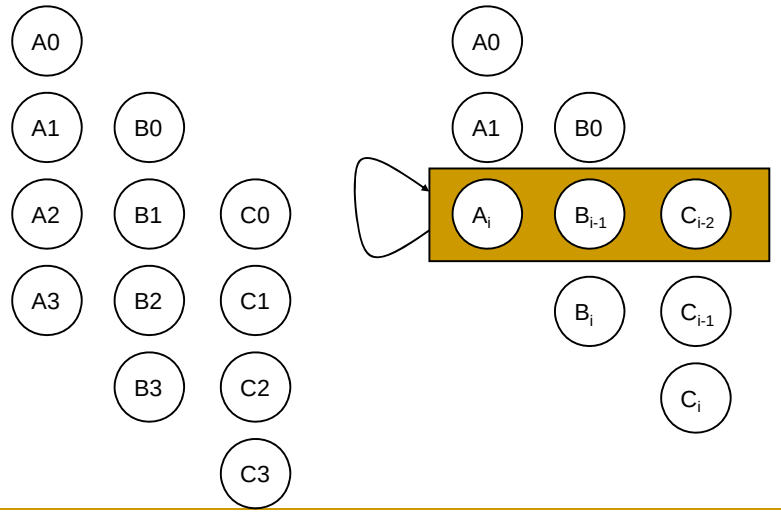

# Goal of SP

- Increase distance between dependent operations by moving destination operation to a later iteration
- But also, to uncover ILP across iteration boundaries!



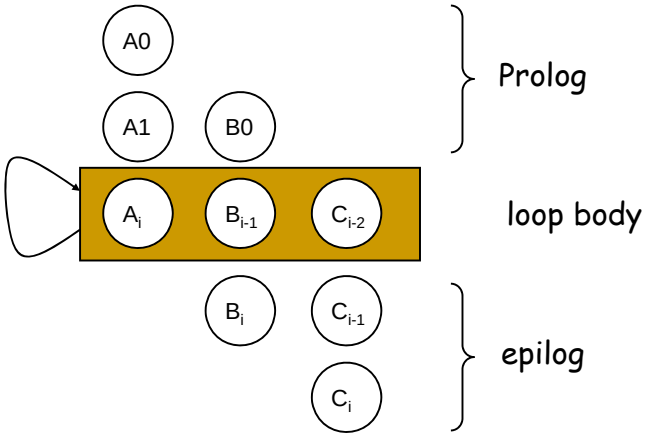
# Example

Assume operating on a infinite wide machine



# Example

Assume operating on a infinite wide machine






# Dealing with exit conditions

```
for (i=0; i<N;
i++)
{
    Ai
    Bi
    Ci
}
```

```
i=0
if (i >= N) goto done
A0
B0
if (i+1 == N) goto last
i=1
A1
if (i+2 == N) goto epilog
i=2
```

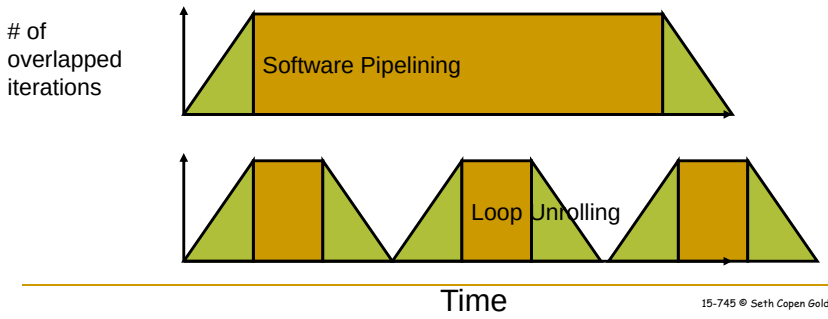
```
loop:
    Ai
    Bi-1
    Ci-2
    i++
    if (i < N) goto loop
epilog:
    Bi
    Ci-1
last:
    ci
done:
```



# Loop Unrolling V. SP

For SuperScalar or VLIW

- Loop Unrolling reduces loop overhead
- Software Pipelining reduces fill/drain
- Best is if you combine them






# VLIW

- Depends on the compiler
  - As often is the case: compiler algs developed for VLIW are relevant to superscalar, e.g., software pipelining.
  - Why wouldn't SS dynamically "software pipeline?"
- As always: Is there enough statically knowable parallelism?
- What about wasted Fus? Code bloat?
- Many DSPs are VLIW. Why?