Static Instruction Scheduling 15740

Along with VLIW& Systolic

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

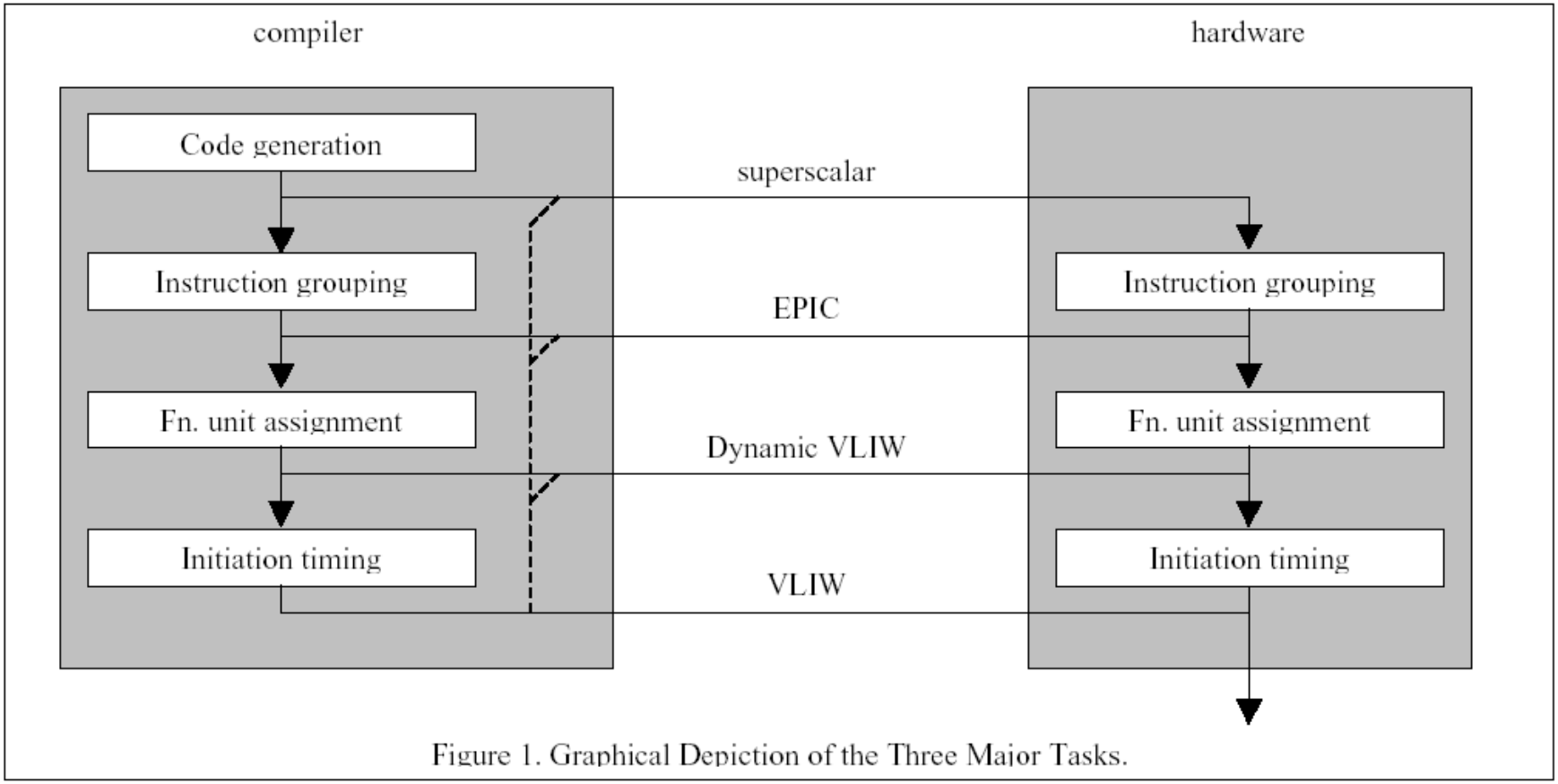
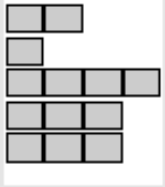
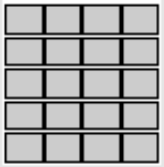
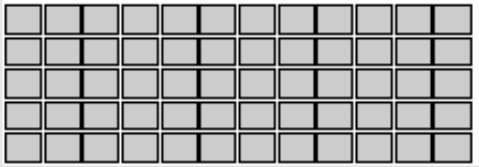
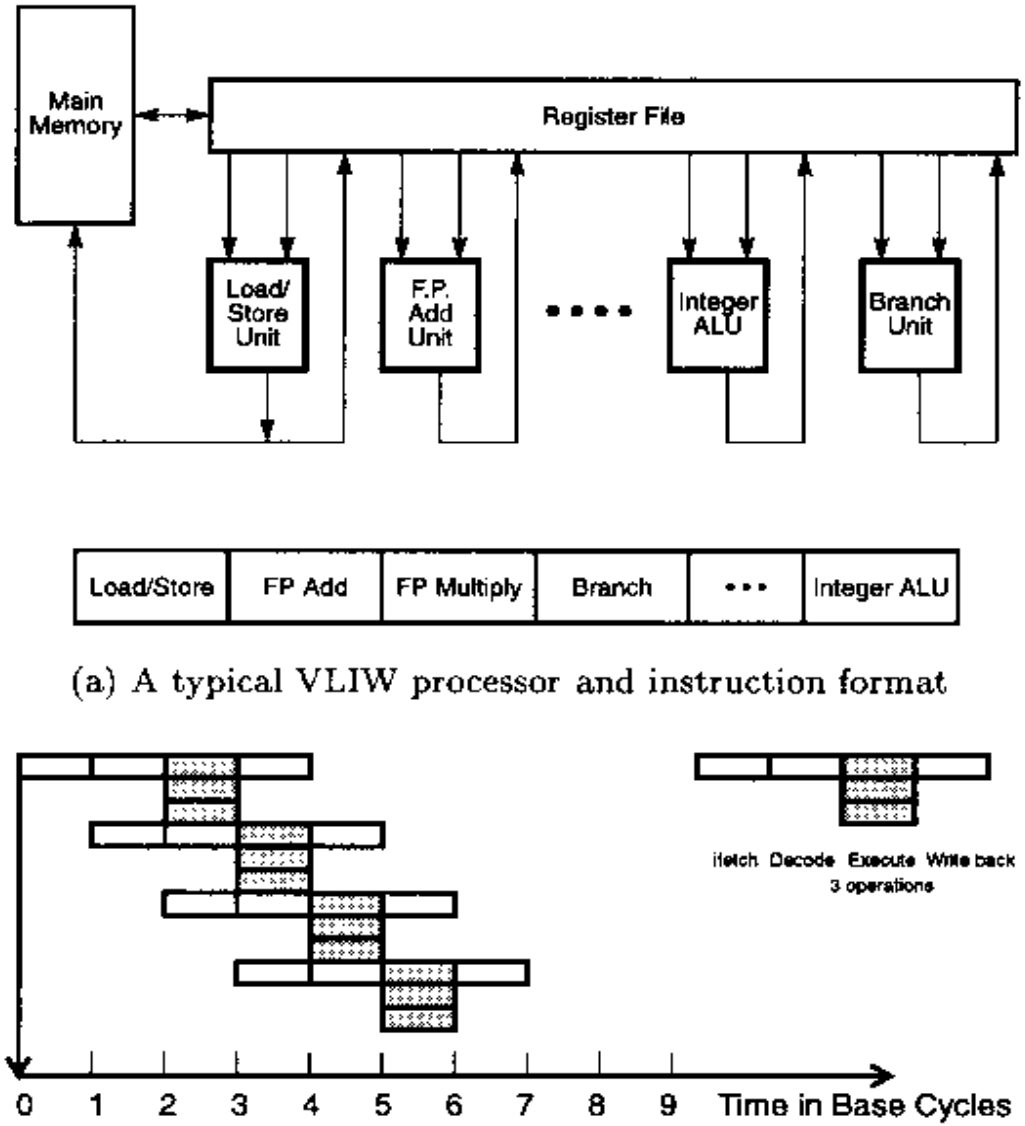


Figure 1. Graphical Depiction of the Three Major Tasks.

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			

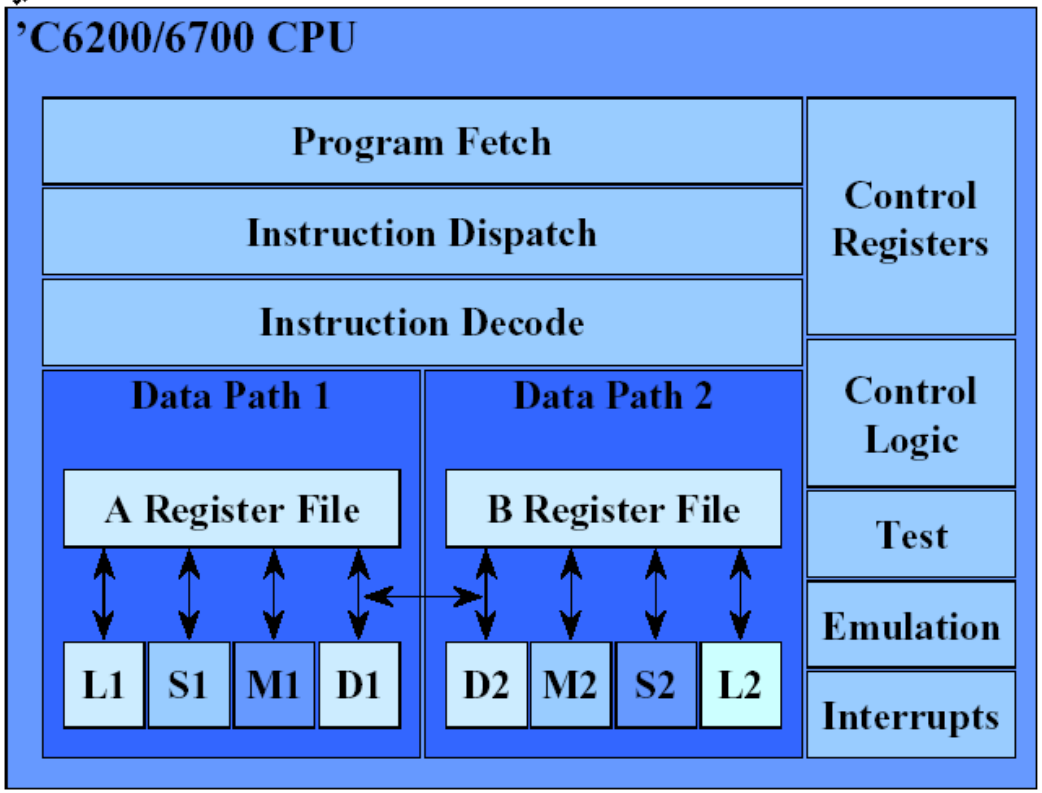


(a) A typical VLIW processor and instruction format

(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

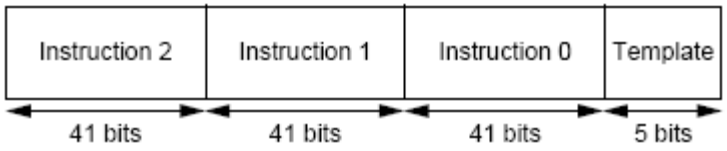
-- Hardware needs to perform dependency checking (albeit aided by software)

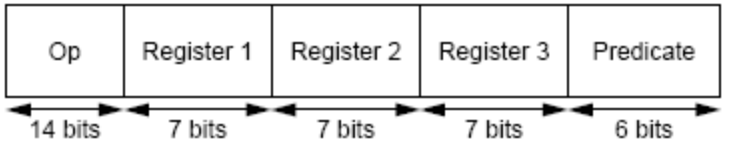
-- Other disadvantages of VLIW still exist

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

IA-64 Instructions

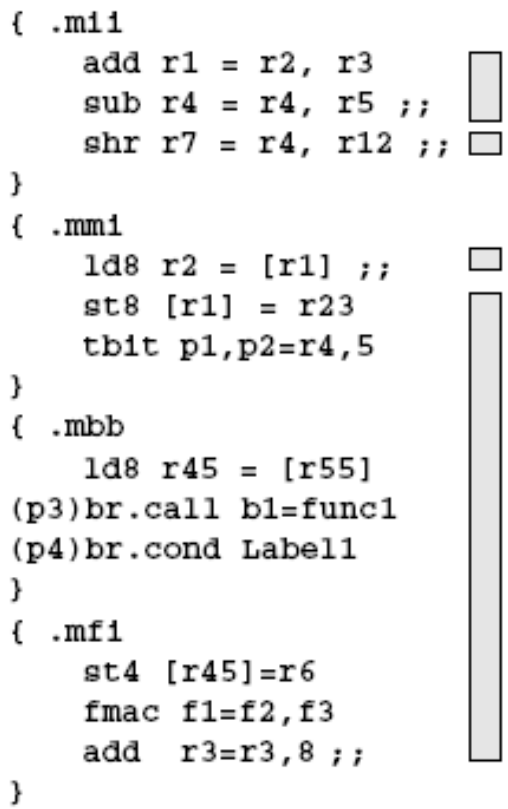
- IA-64 “Bundle” (~EPIC Instruction)
  - Total of 128 bits
  - Contains three IA-64 instructions
  - Template bits in each bundle specify dependencies within a bundle





- 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



- 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

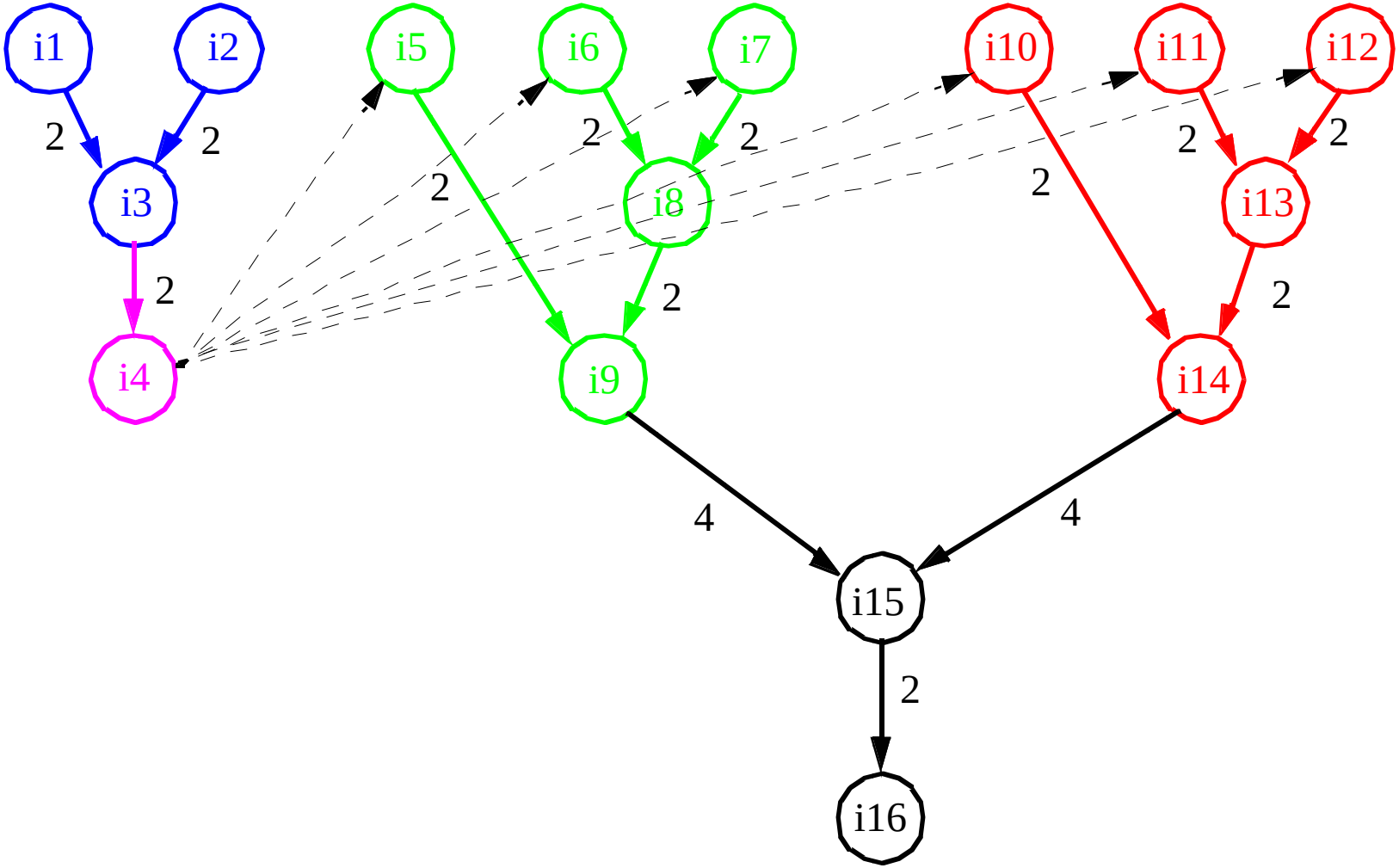
- Within a basic block, there is limited instruction-level parallelism
- To find multiple instructions to be executed in parallel, the compiler needs to consider multiple basic blocks
- Problem: Moving an instruction above a branch is unsafe because instruction is not guaranteed to be executed
- Idea: 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

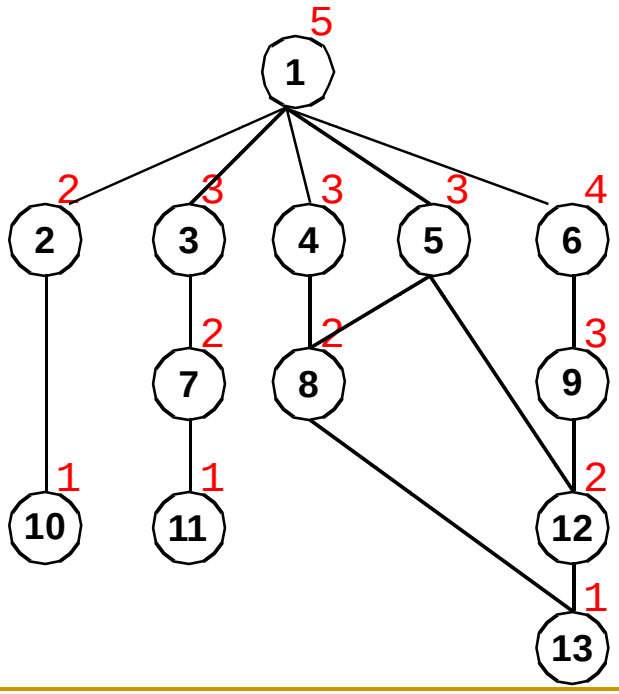


Instruction Prioritization Heuristics

- 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



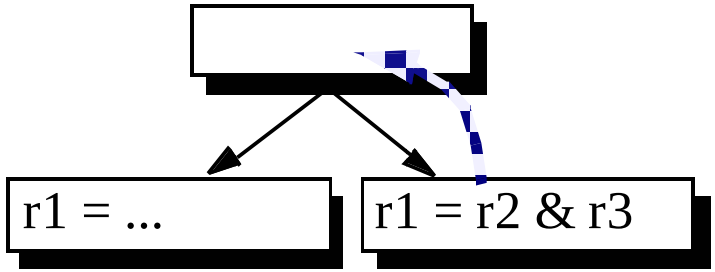
4-wide VLIW				Data Ready List
1				{1}
6	3	4	5	{2,3,4,5,6}
9	2	7	8	{2,7,8,9}
12	10	11		{10,11,12}
13				{13}

Extending the scheduling domain

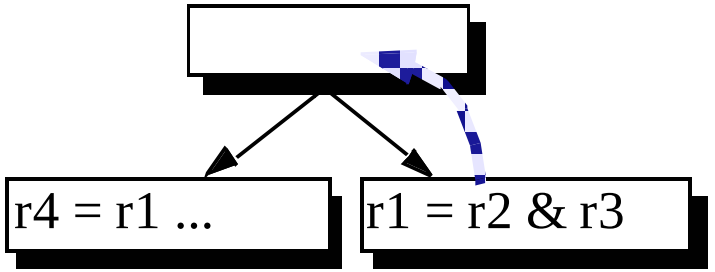
- Basic block is too small to get any real parallelism
- How to extend the basic block?
  - Why do we have basic blocks in the first place?
  - Loops
    - Loop unrolling
    - Software pipelining
  - Non-loops
    - Will almost always involve some speculation
    - And, thus, profiling may be very important

Safety and Legality in Code Motion

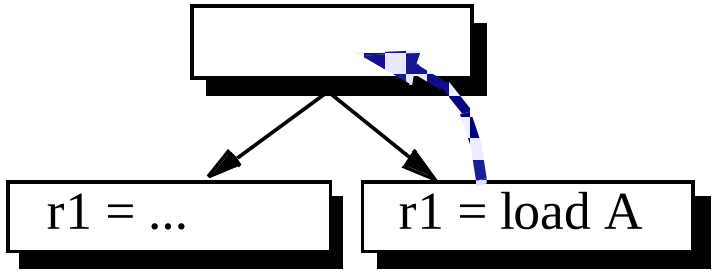
- Two characteristics of speculative code motion:
  - Safety: whether or not spurious exceptions may occur
  - Legality: whether or not result will be always correct
- Four possible types of code motion:



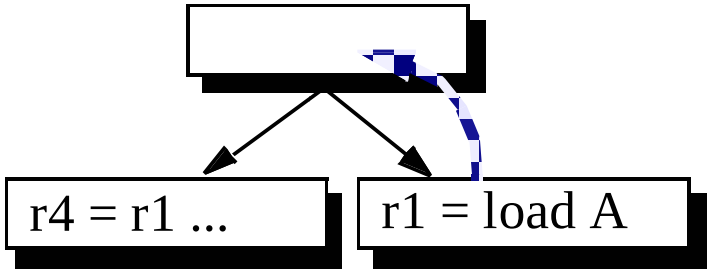
(a) safe and legal



(b) illegal



(c) unsafe



(d) unsafe and illegal

Code Movement Constraints

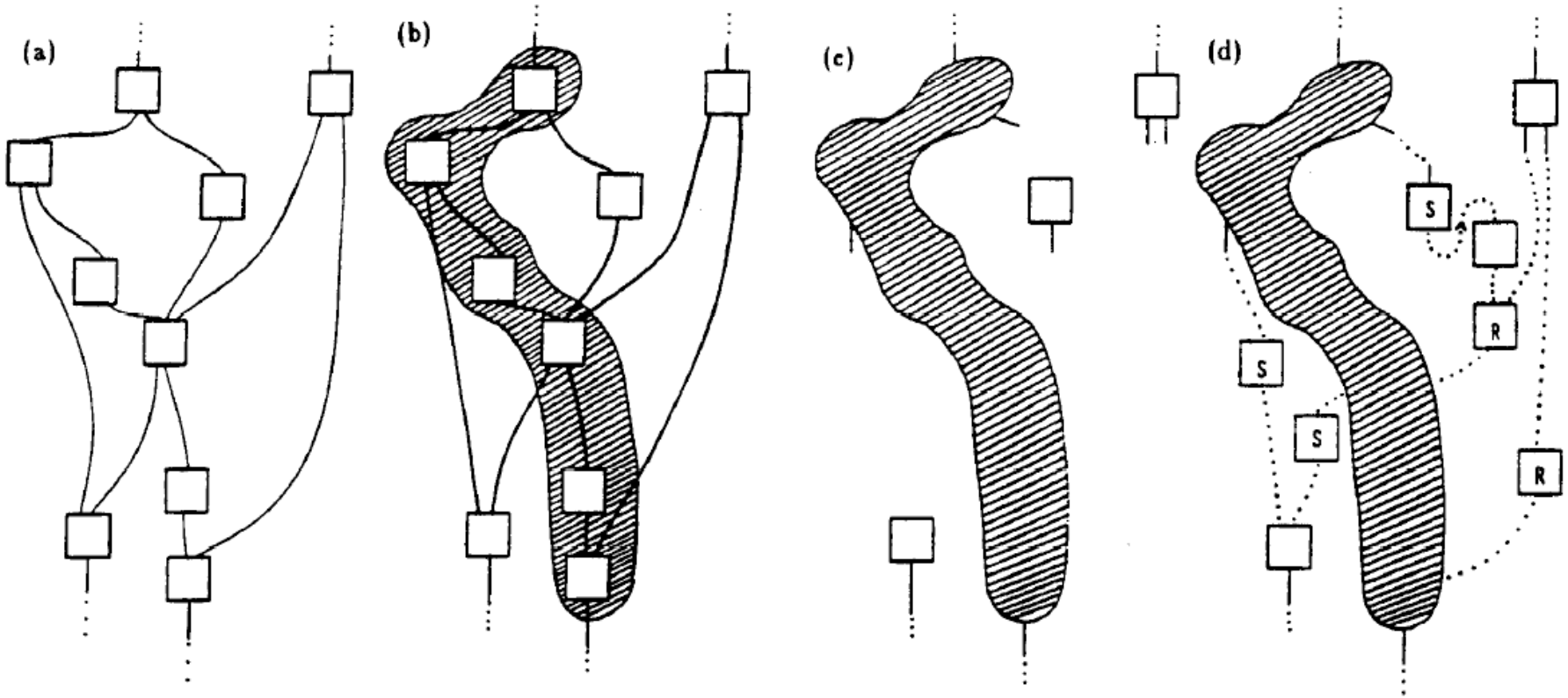
- Downward
  - When moving an operation from a BB to one of its dest BB's,
    - all the other dest basic blocks should still be able to use the result of the operation
    - the other source BB's of the dest BB should not be disturbed
- Upward
  - When moving an operation from a BB to its source BB's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

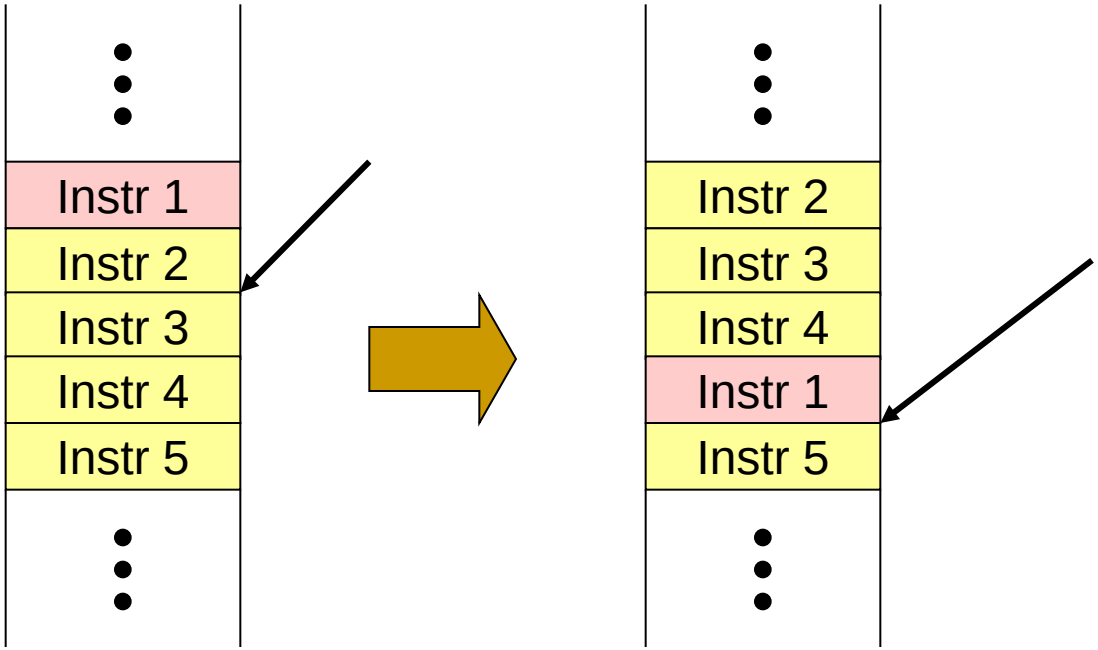


TRACE SCHEDULING LOOP-FREE CODE

Trace Scheduling (II)

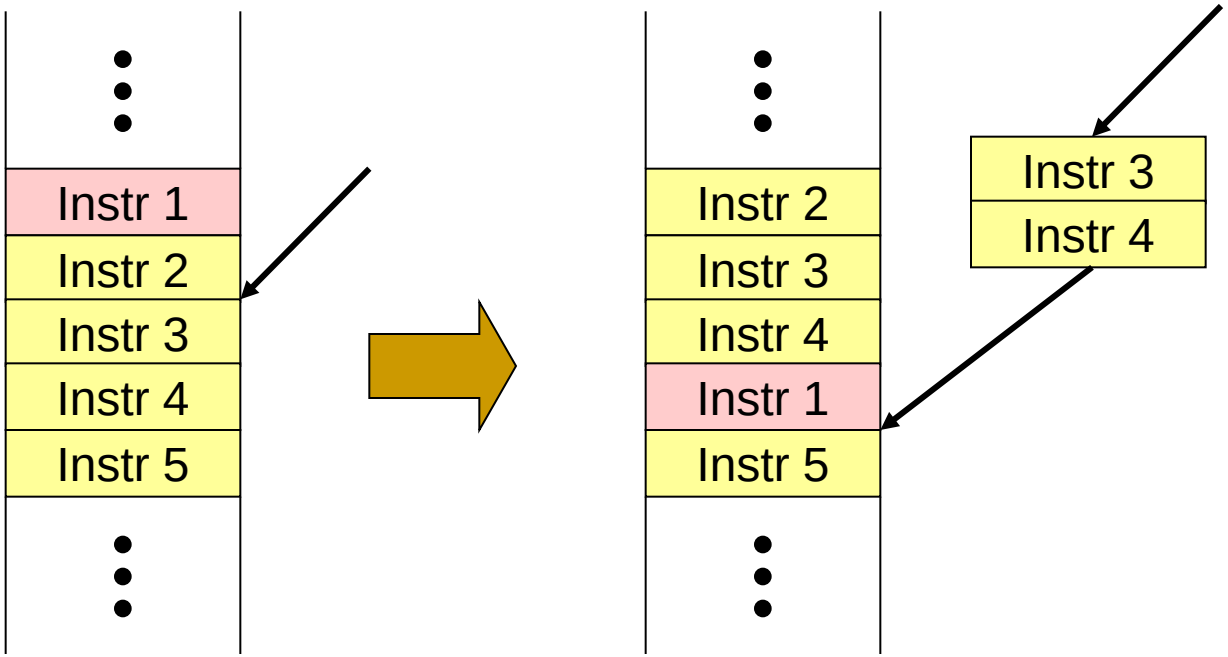
- There may be conditional branches from the middle of the trace (side exits) and transitions from other traces into the middle of the trace (side entrances).
- These control-flow transitions are ignored during trace scheduling.
- After scheduling, fix-up/bookkeeping code is inserted to ensure the correct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

Trace Scheduling (III)

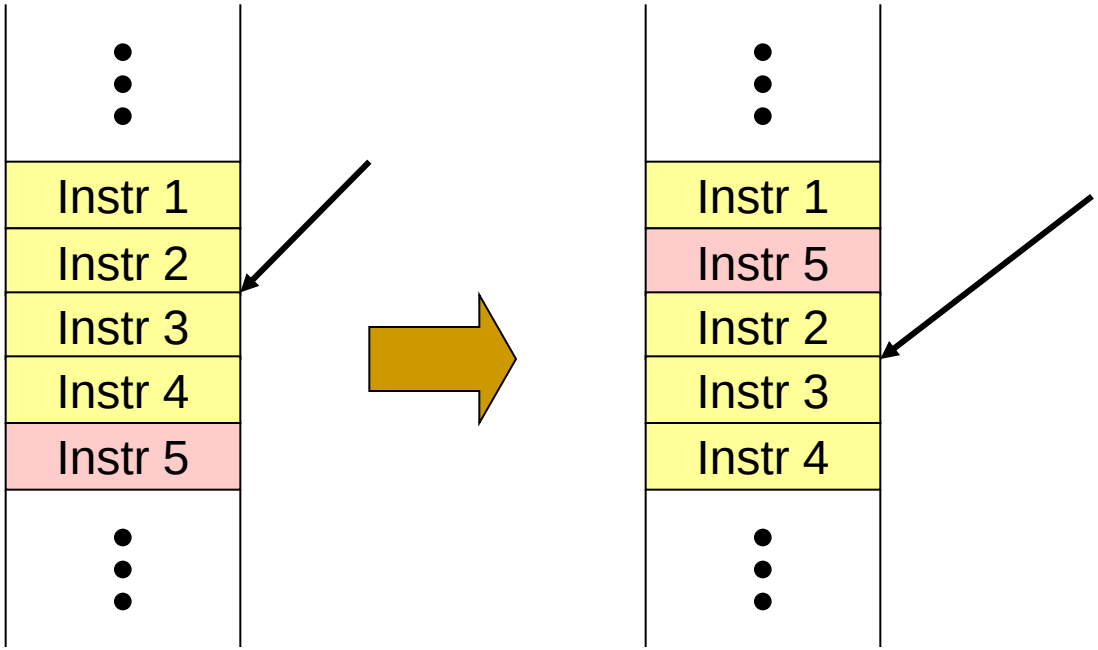


What bookeeping is required when Instr 1 is moved below the side entrance in the trace?

Trace Scheduling (IV)

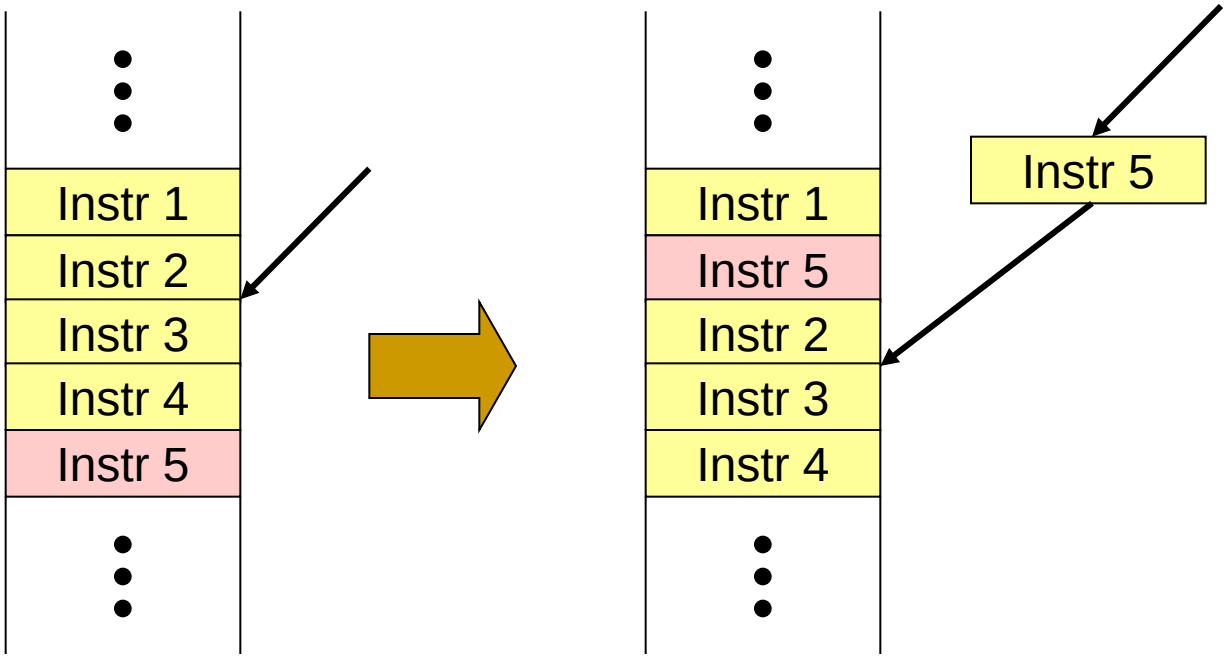


Trace Scheduling (V)



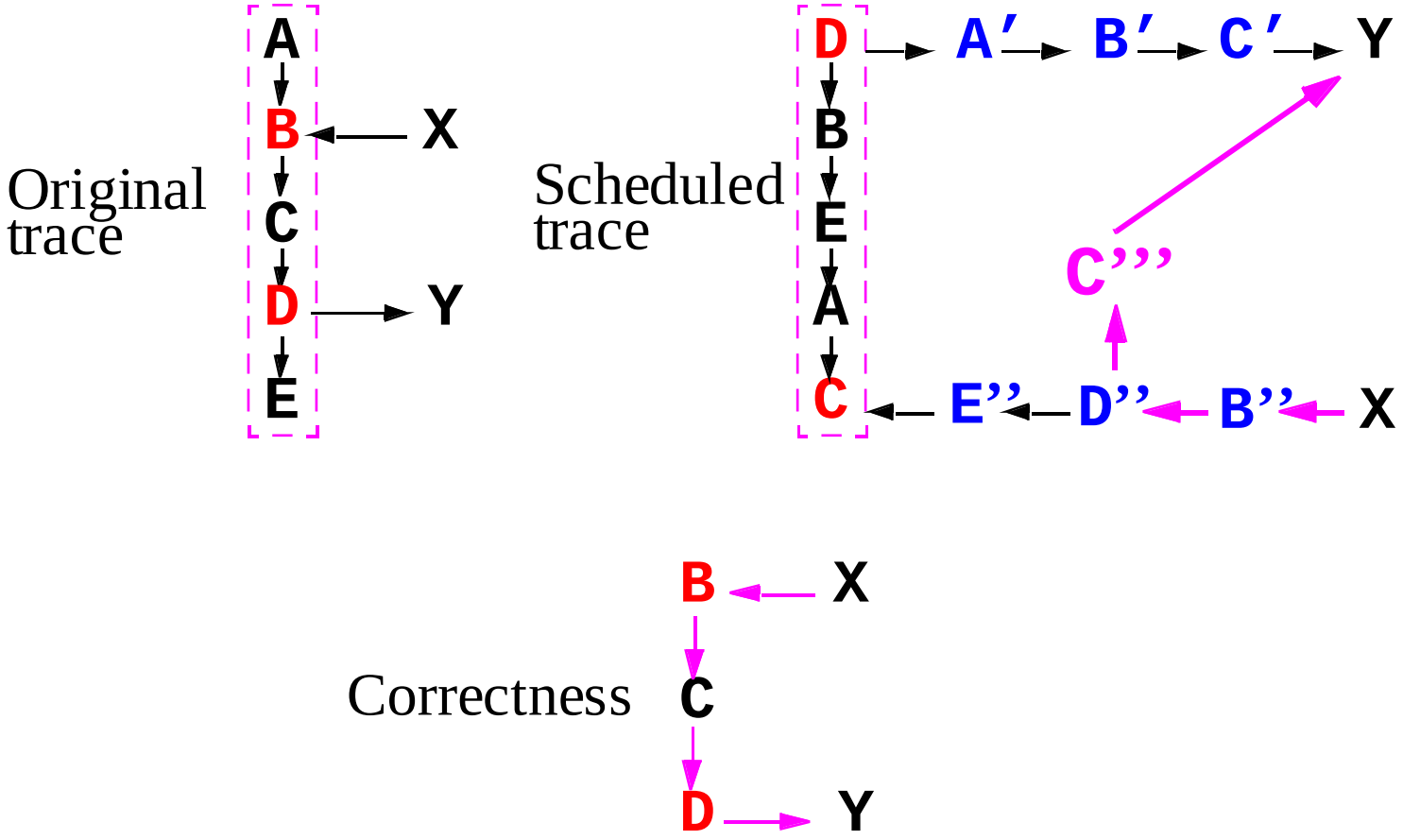
What bookeeping is required when Instr 5 moves above the side entrance in the trace?

Trace Scheduling (VI)



Trace Scheduling Fixup Code Issues

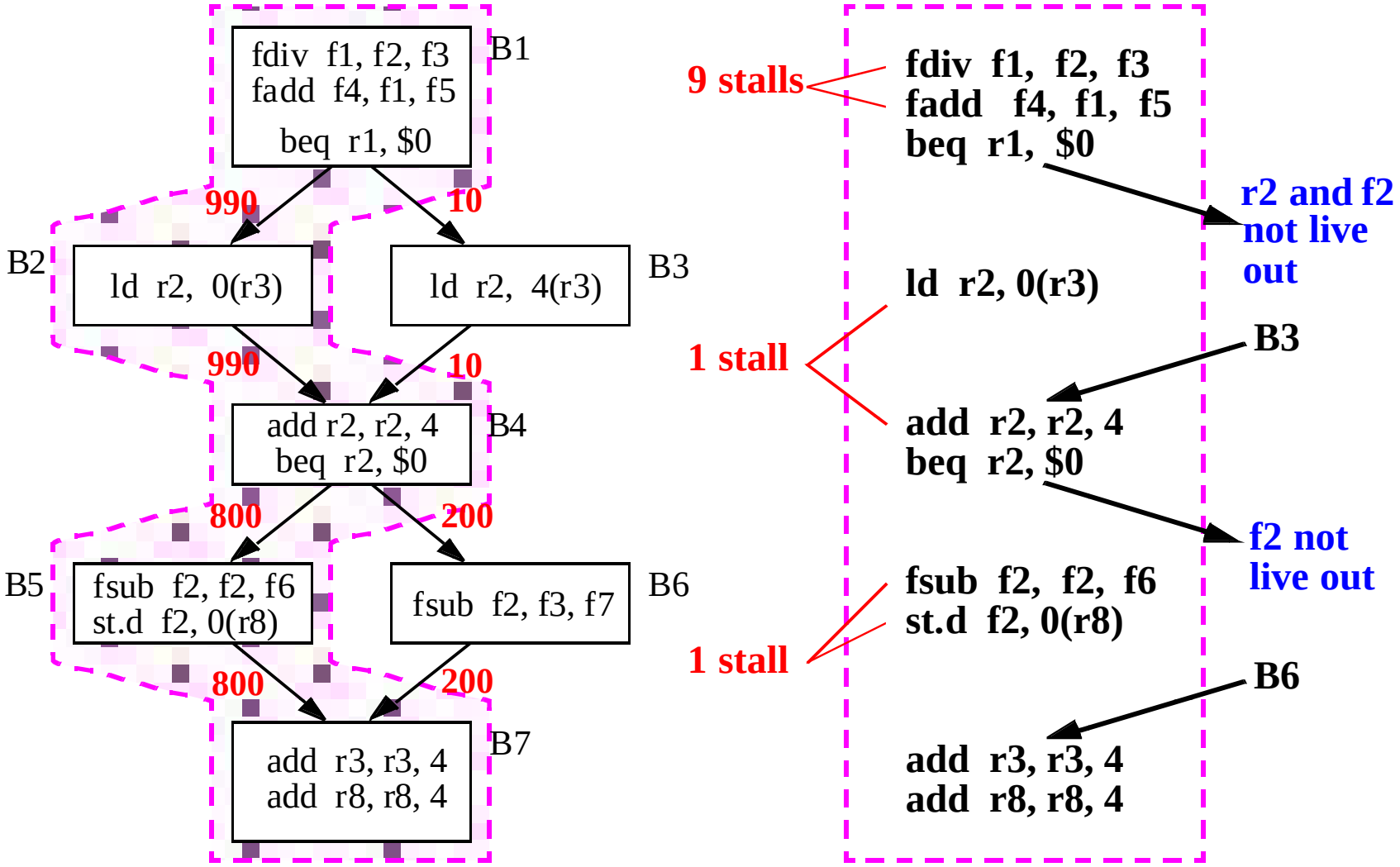
- Sometimes need to copy instructions more than once to ensure correctness on all paths (see C below)



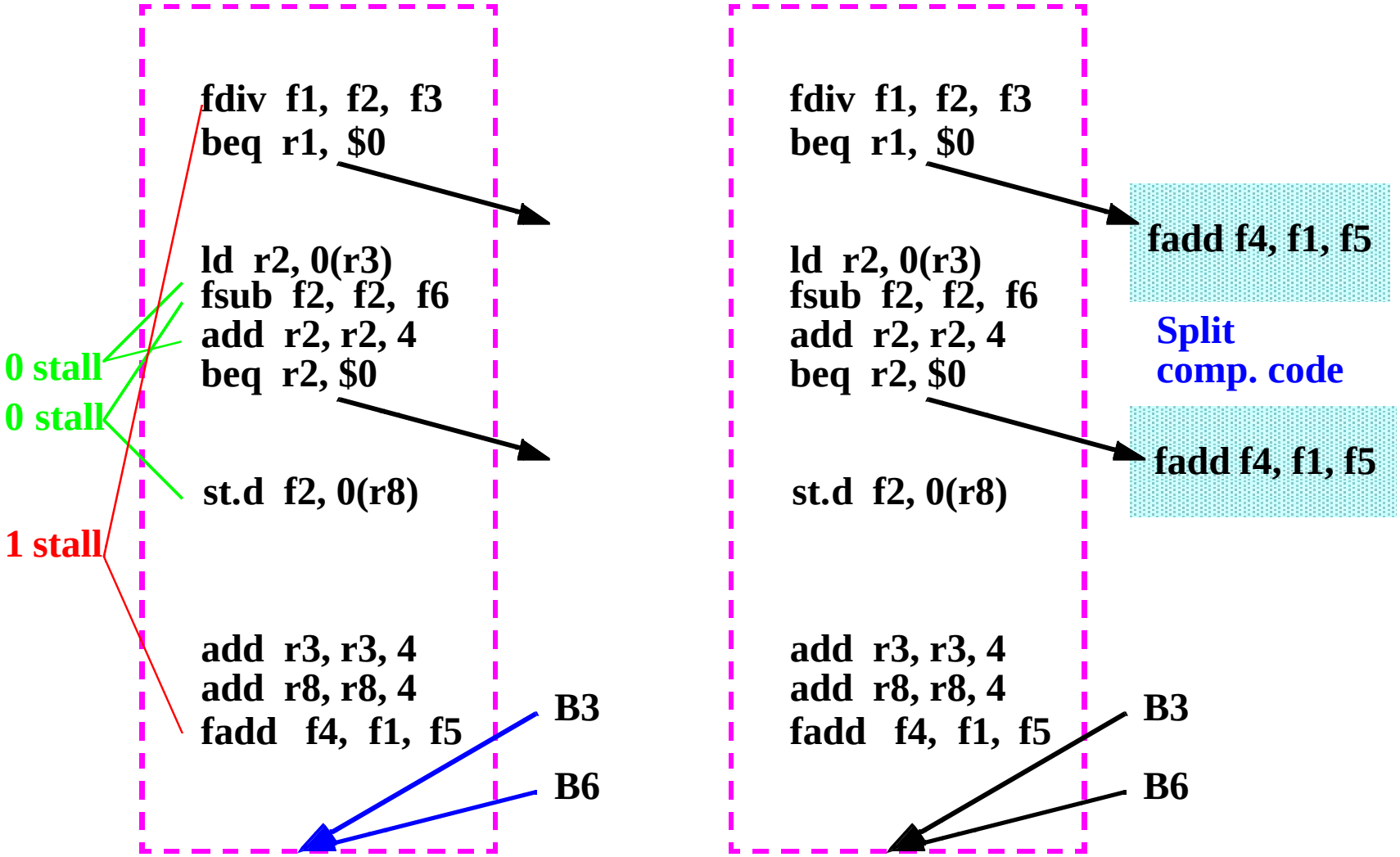
Trace Scheduling Overview

- 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

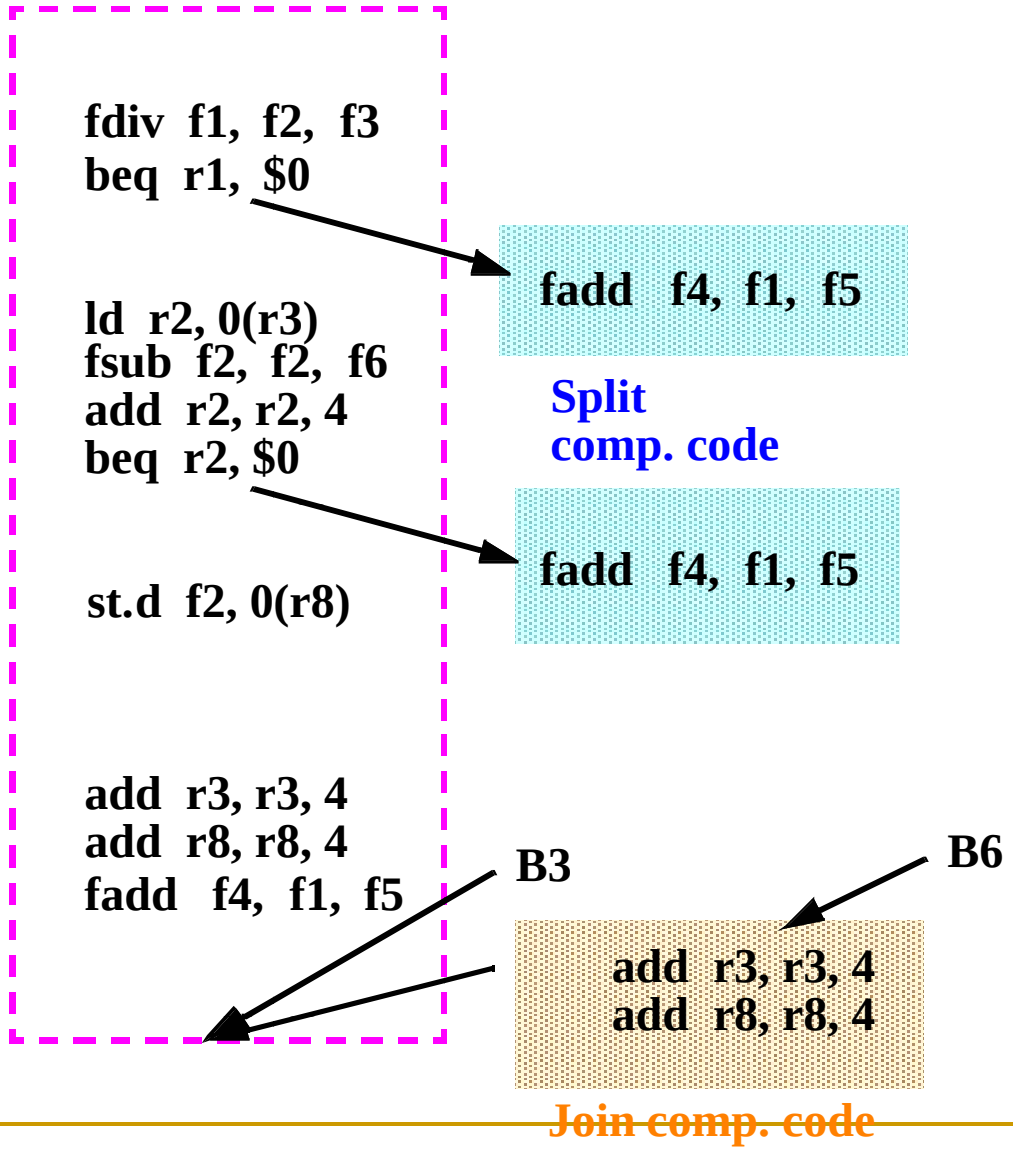
Trace Scheduling Example (I)



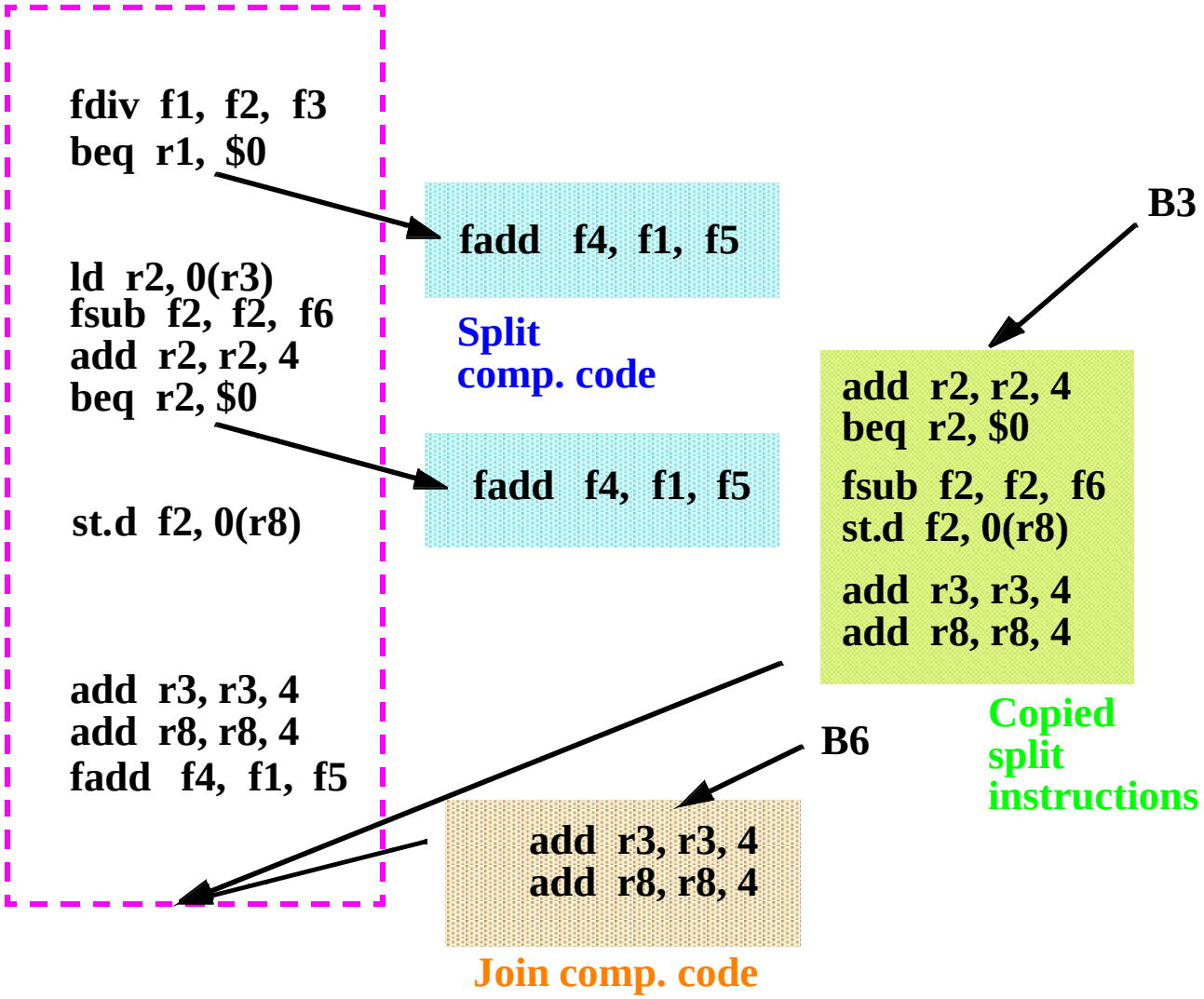
Trace Scheduling Example (II)



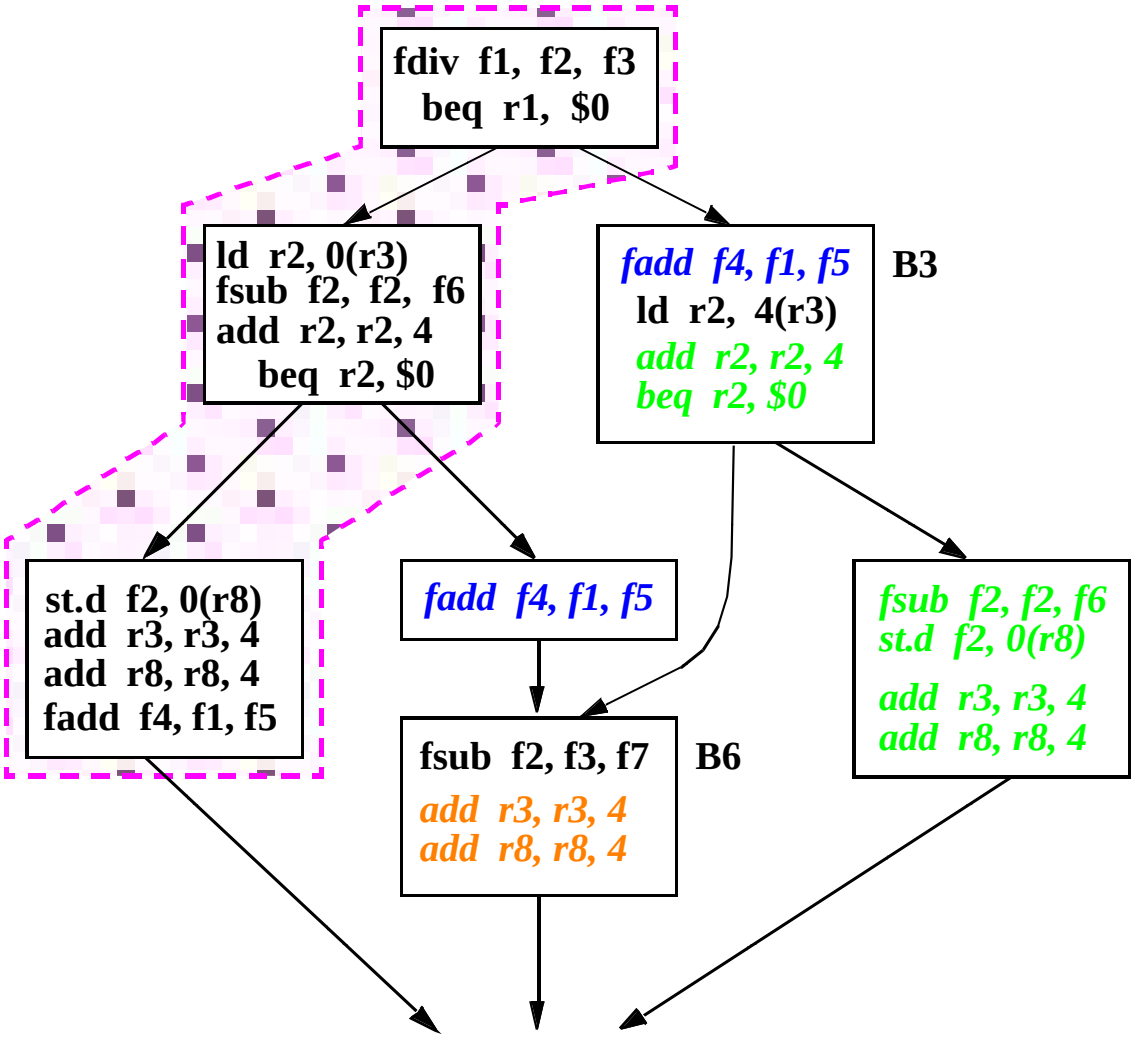
Trace Scheduling Example (III)



Trace Scheduling Example (IV)



Trace Scheduling Example (V)



Trace Scheduling Tradeoffs

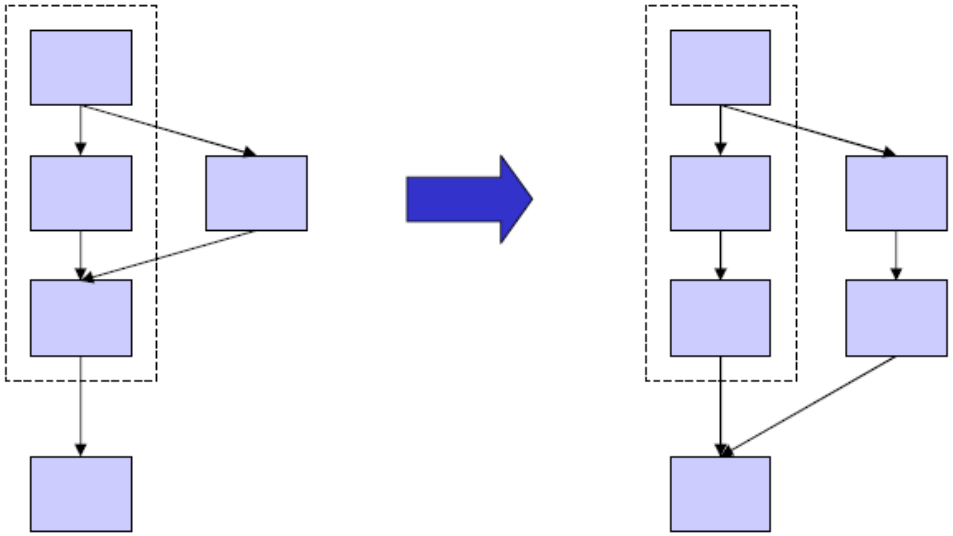
- Advantages
  + Enables the finding of more independent instructions → fewer NOPs in a VLIW instruction

- 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

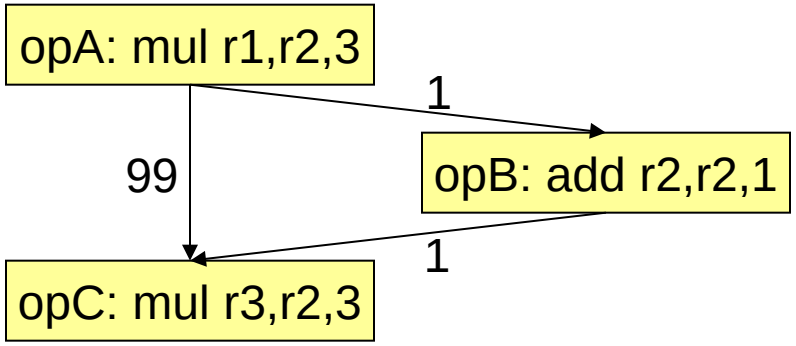
- Trace: multiple entry, multiple exit block
- Superblock: single-entry, multiple exit block
  - A trace with side entrances are eliminated
  - Infrequent paths do not interfere with the frequent path

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

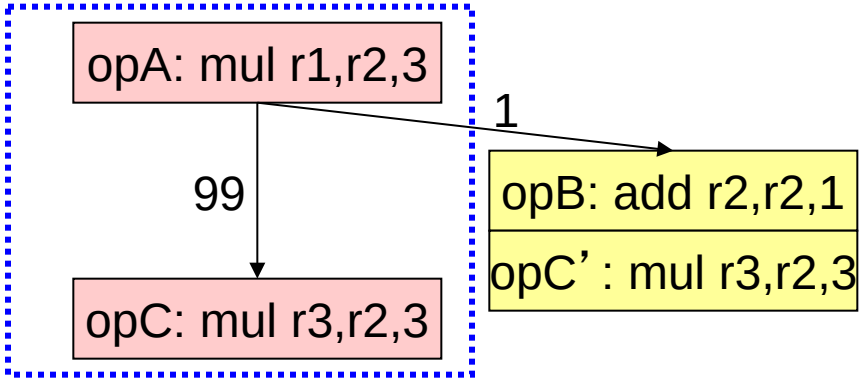


Hwu+, “The Superblock: An Effective Technique for VLIW and superscalar compilation,” J of SC 1991.

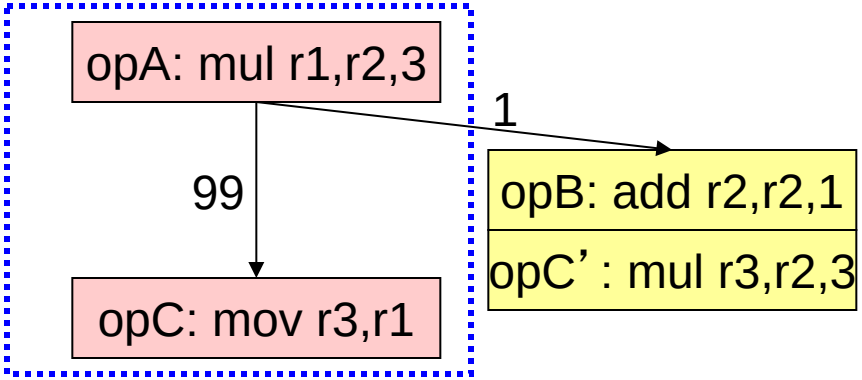
Can You Do This with a Trace?



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 scheduled block size

- Disadvantages

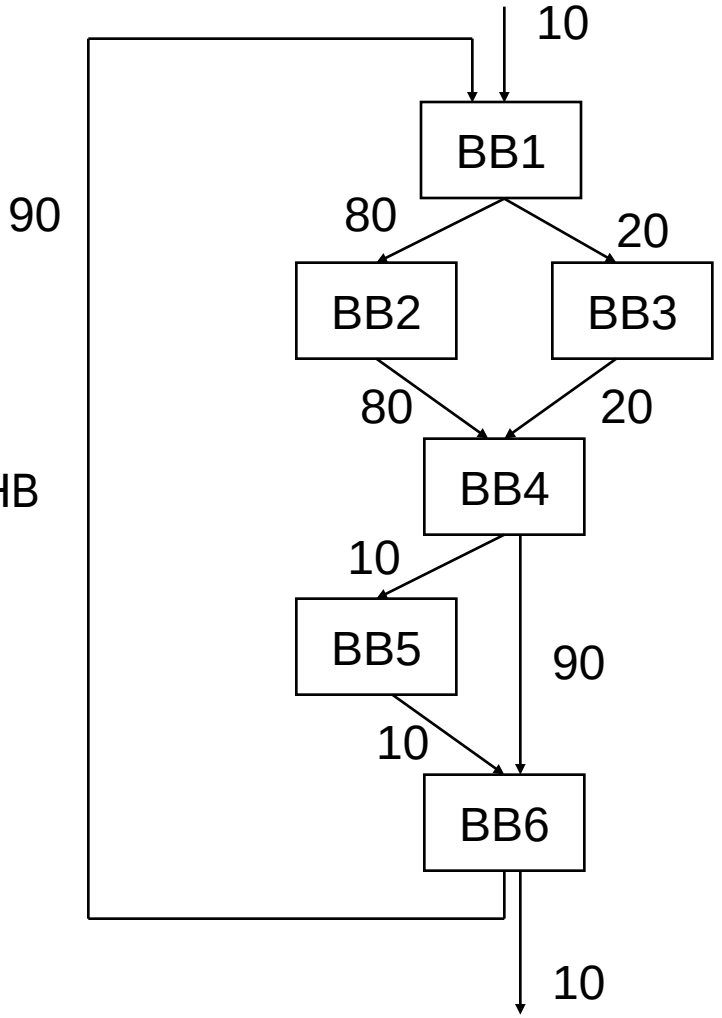
  -- Requires predicated execution support

  -- All disadvantages of predicated execution

Hyperblock Formation (I)

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

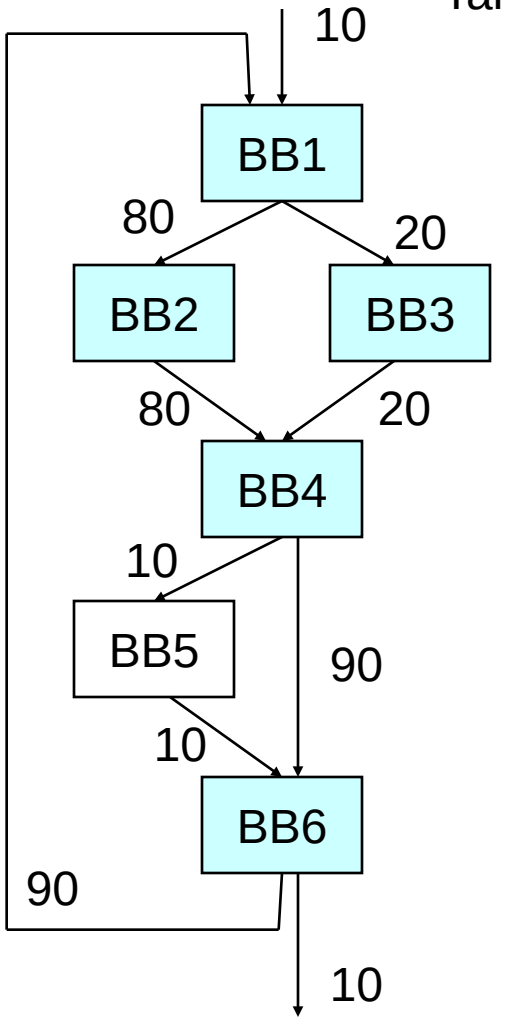
- Block selection
  - Select subset of BBs for inclusion in HB
  - Difficult problem
  - Weighted cost/benefit function
    - Height overhead
    - Resource overhead
    - Dependency overhead
    - Branch elimination benefit
    - Weighted by frequency

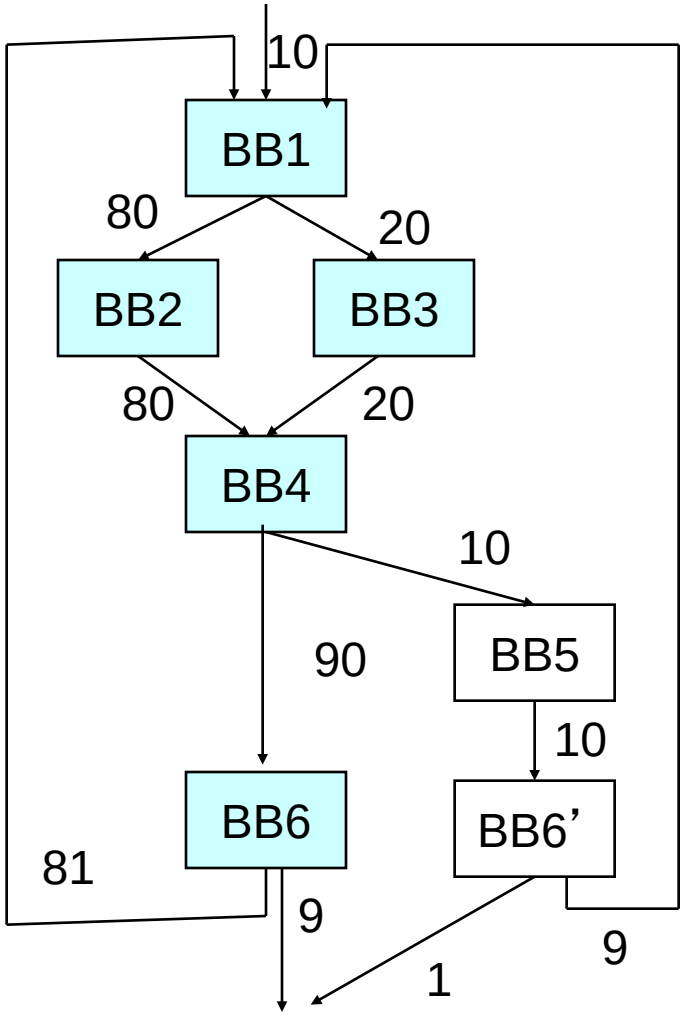


- Mahlke et al., "Effective Compiler Support for Predicated Execution Using the Hyperblock," MICRO 1992.

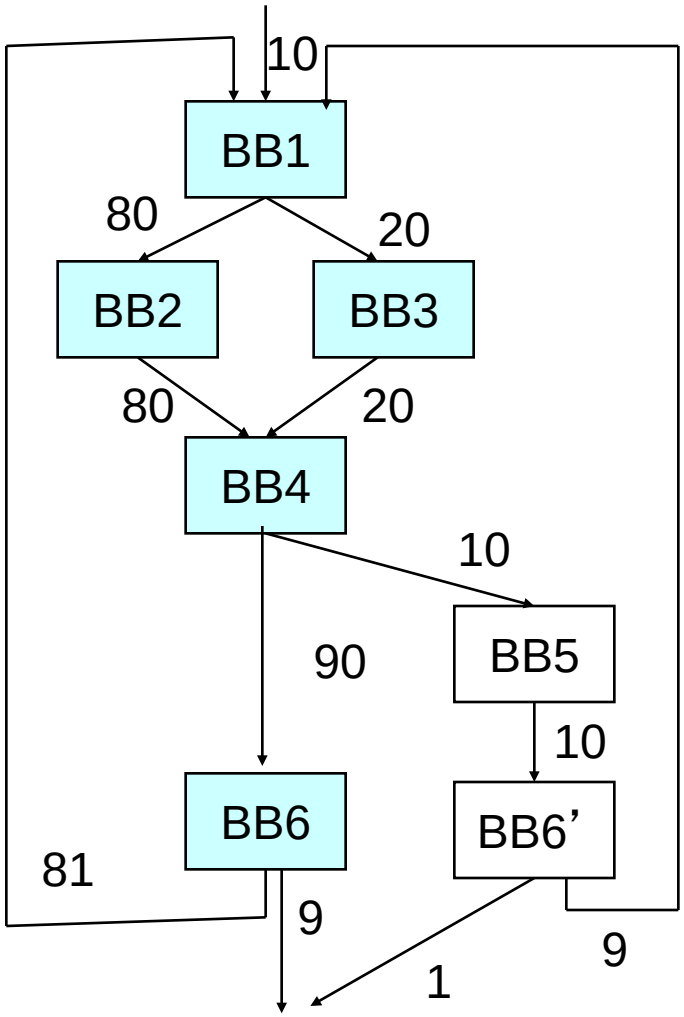
Hyperblock Formation (II)

Tail duplication same as with Superblock formation

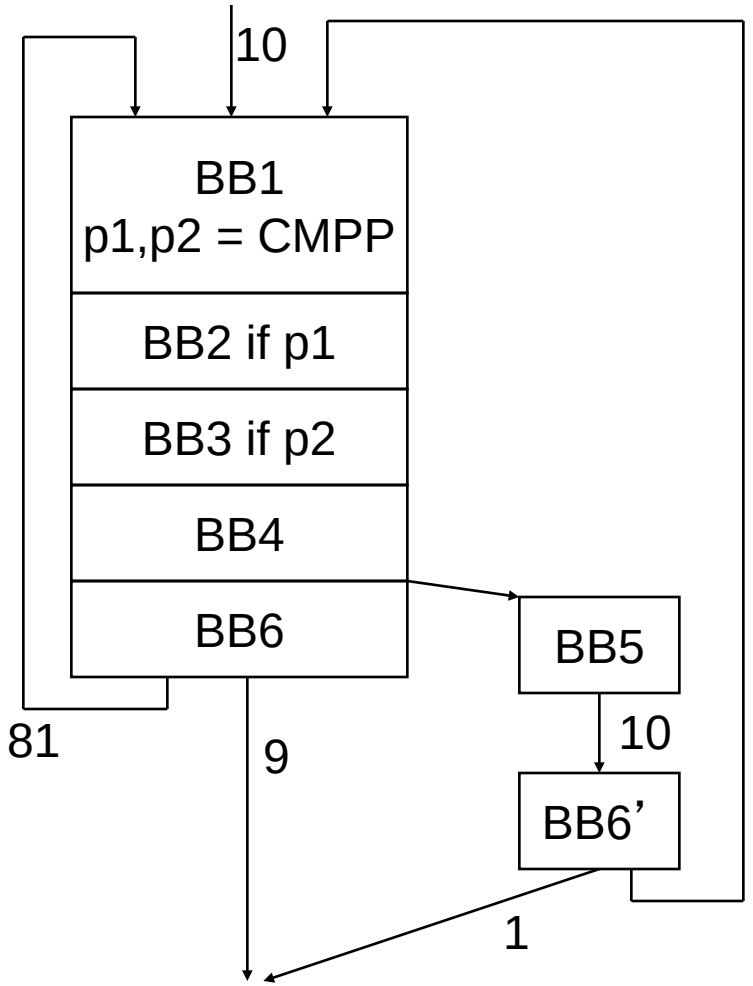




Hyperblock Formation (III)



If-convert (predicate) intra-hyperblock branches

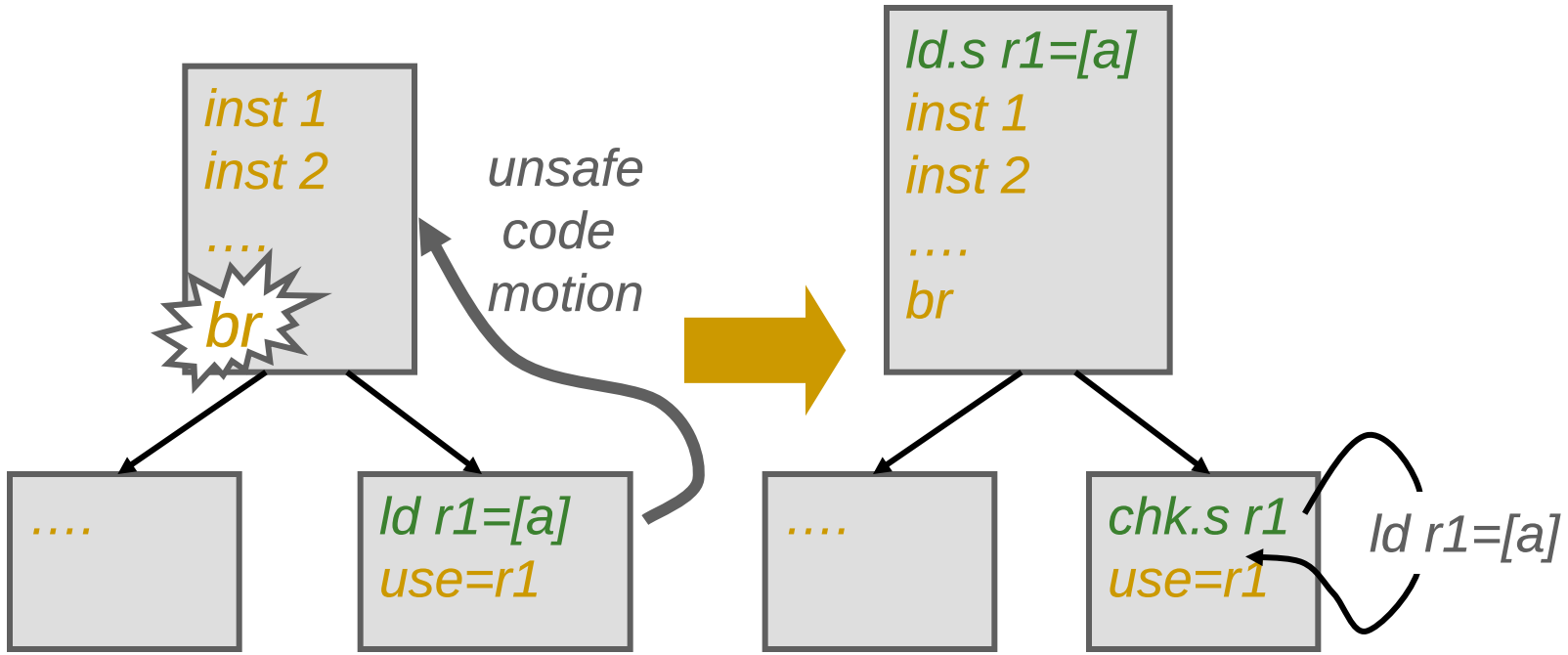


Can We Do Better?

- 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)

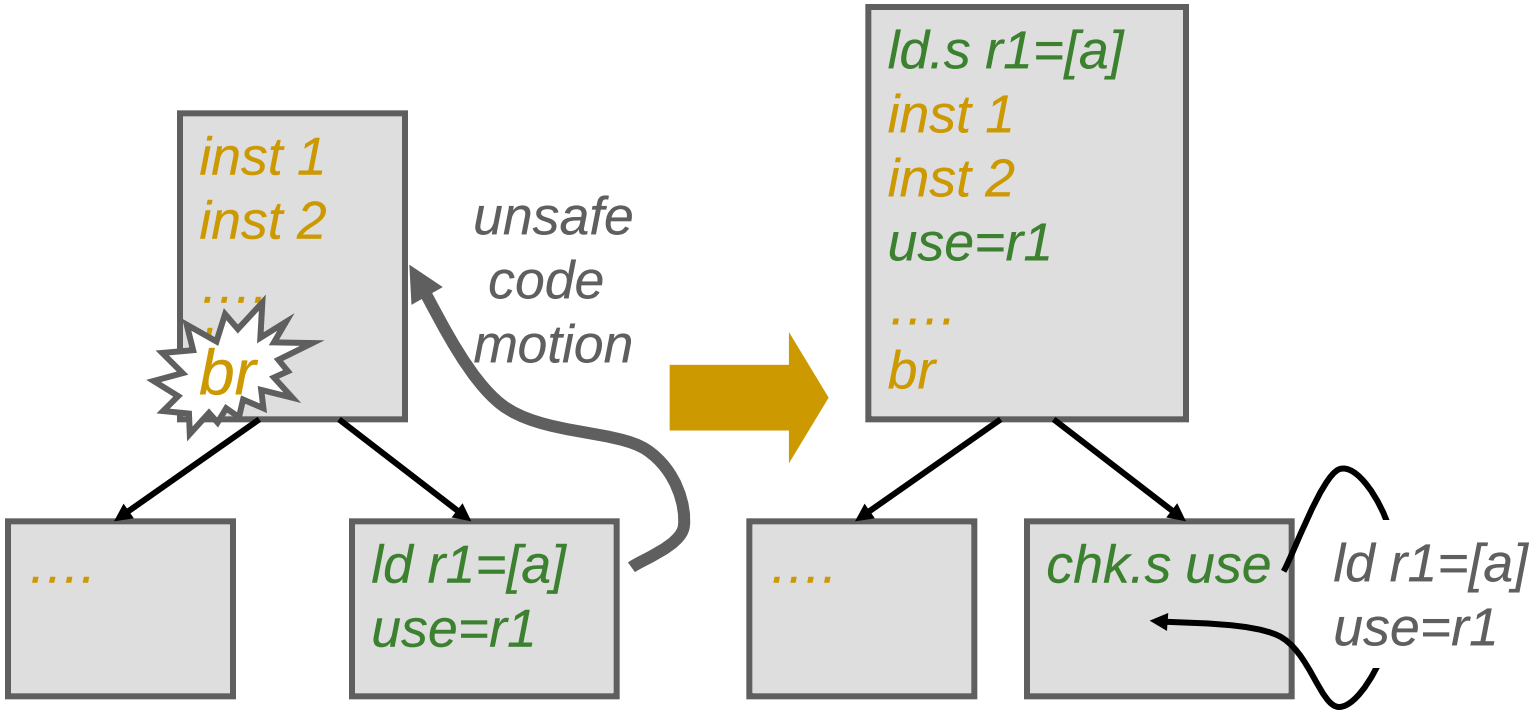
Non-Faulting Loads and Exception Propagation



- *ld.s* fetches *speculatively* from memory

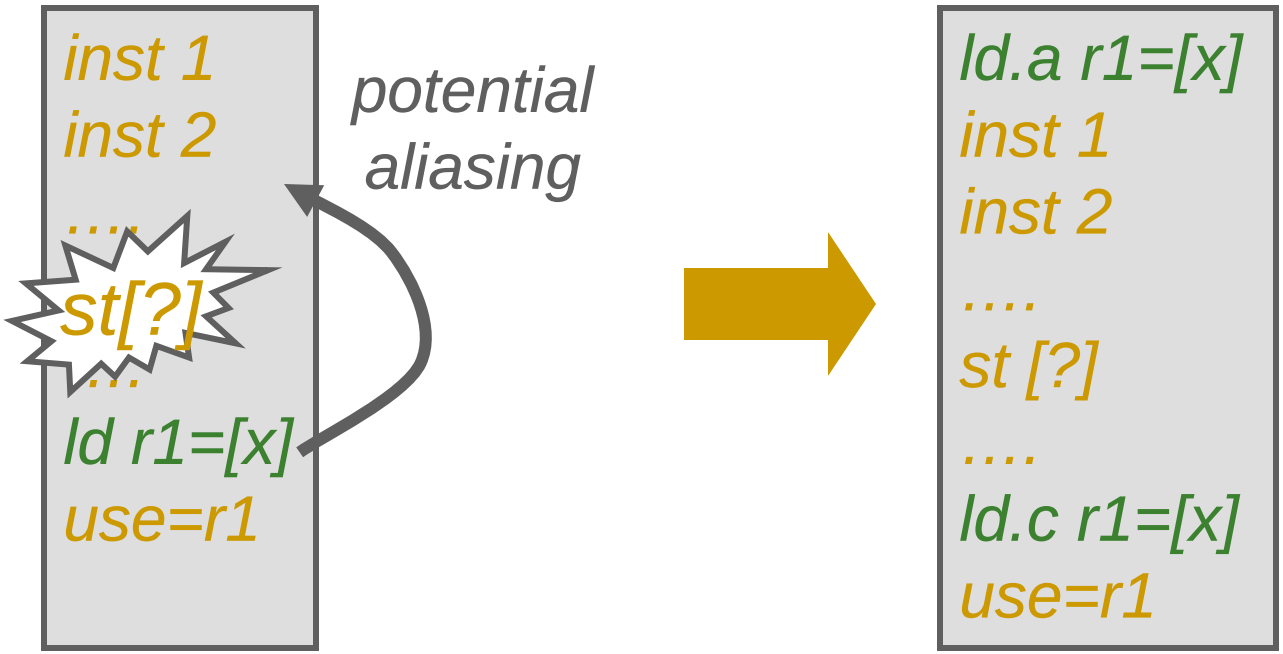
  i.e. any exception due to *ld.s* is suppressed
- If *ld.s r1* did not cause an exception then *chk.s r1* is a NOP, else a branch is taken (to execute some compensation code)

Non-Faulting Loads and Exception Propagation in IA-64



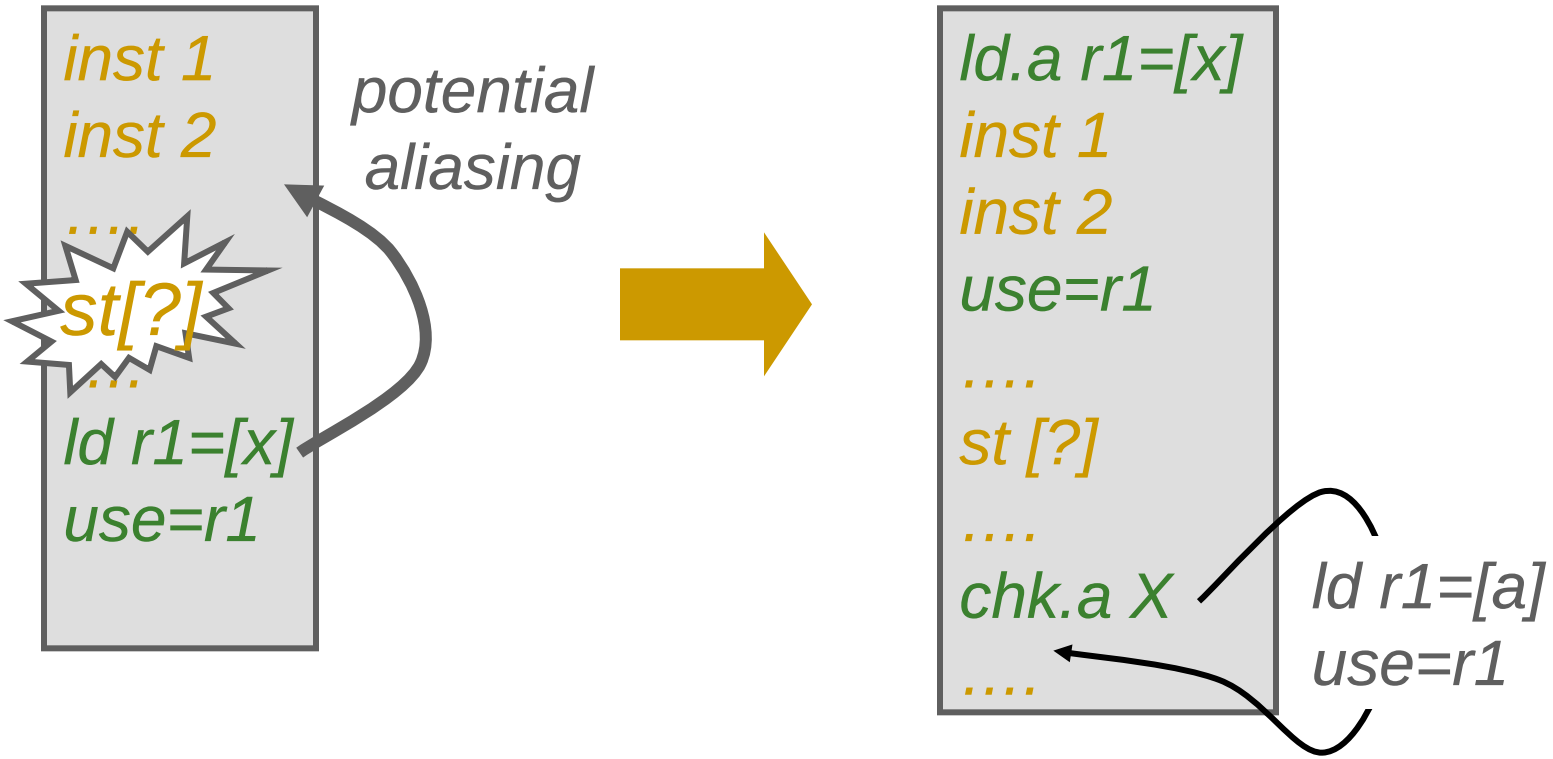
- Load data can be speculatively consumed prior to check
- "speculation" status is propagated with speculated data
- Any instruction that uses a speculative result also becomes speculative itself (i.e. suppressed exceptions)
- *chk.s* checks the entire dataflow sequence for exceptions

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)

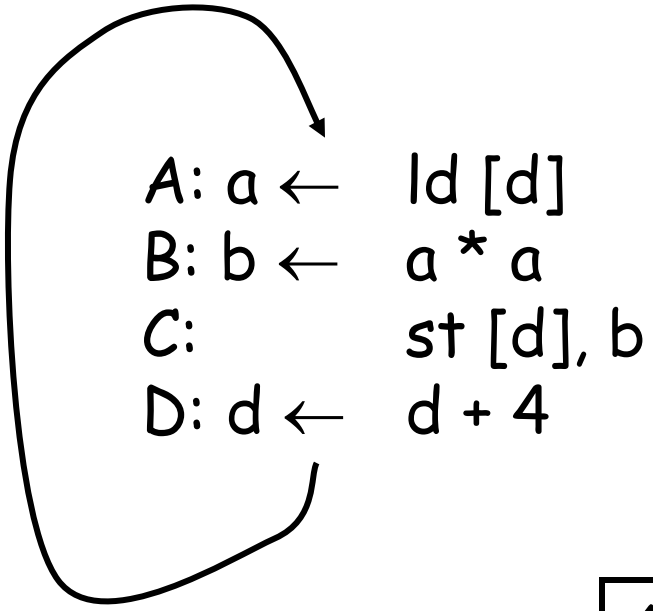
-- Increases code size

Software Pipelining

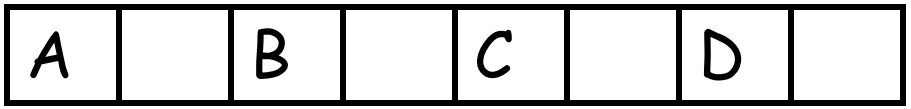
- Software pipelining is an instruction scheduling technique that reorders the instructions in a loop.
  - Possibly moving instructions from one iteration to the previous or the next iteration.
  - Very large improvements in running time are possible.
- The first serious approach to software pipelining 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
```

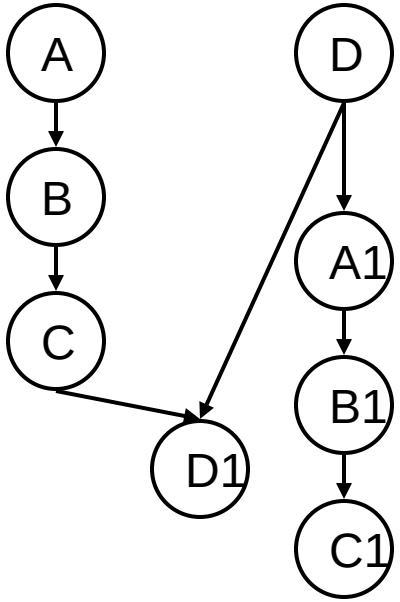
A		B		C		D		A1		B1		C1		D1	

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
```

Schedule

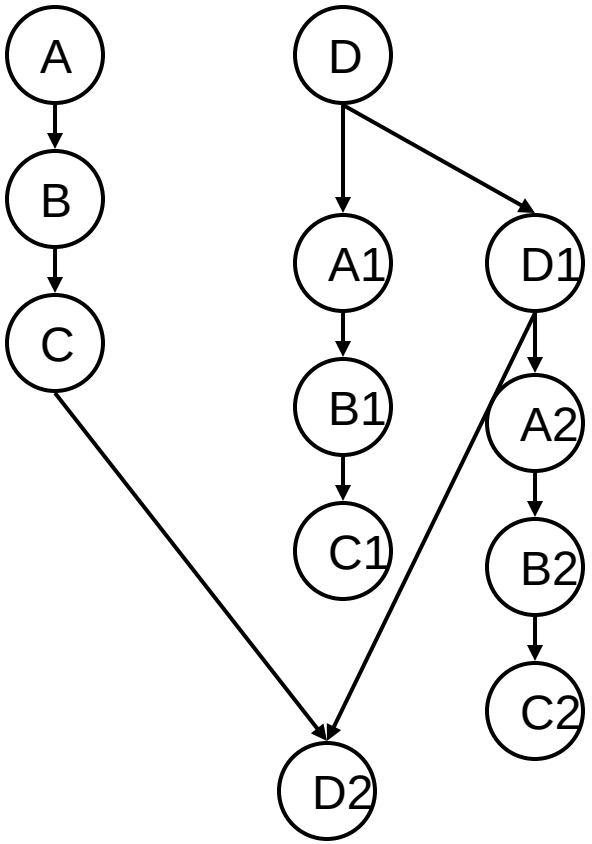
```
A:  a ←   ld [d]
B:  b ←   a * a
C:        st [d], b
D:  d1 ←  d + 4
A1: a1 ←  ld [d1]
B1: b1 ←  a1 * a1
C1:       st [d1], b1
D1: d ←   d1 + 4
```



A		B		C		D1	
D		A1		B1		C1	

Unroll Some More

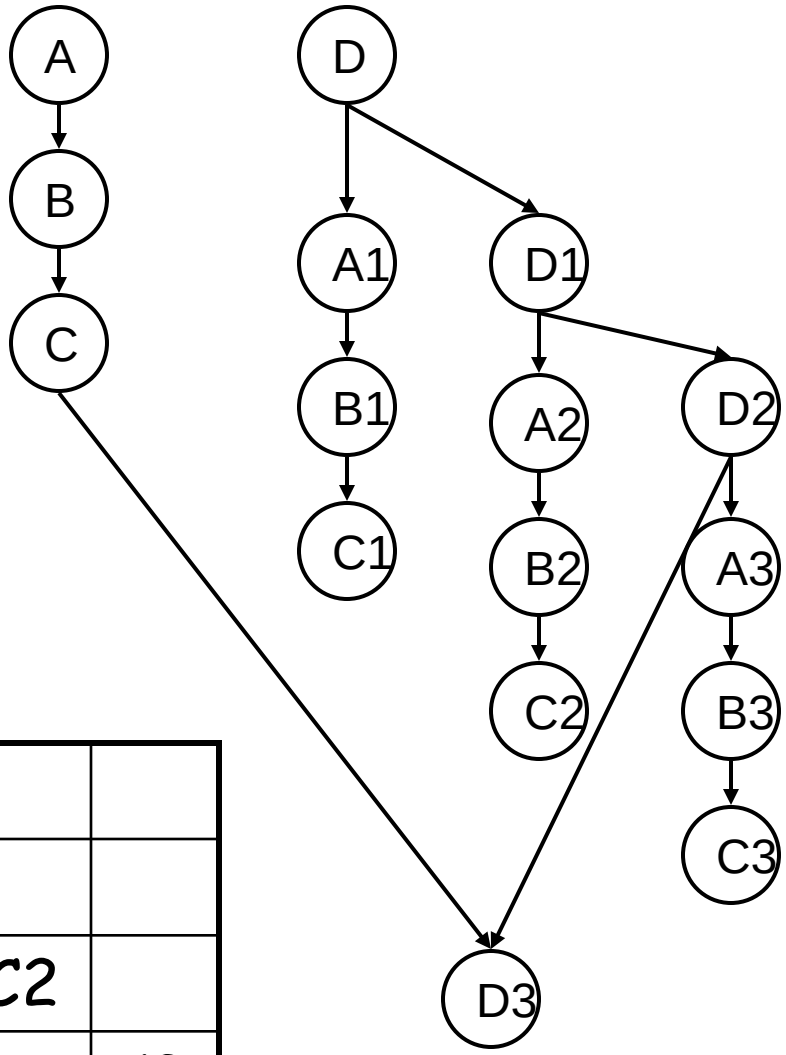
```
A:  a ←  ld [d]
B:  b ←  a * a
C:       st [d], b
D:  d1 ← d + 4
A1: a1 ← ld [d1]
B1: b1 ← a1 * a1
C1:      st [d1], b1
D1: d2 ← d1 + 4
A2: a2 ← ld [d2]
B2: b2 ← a2 * a2
C2:      st [d2], b2
D2: d ←  d2 + 4
```



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

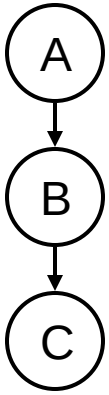
Unroll Some More

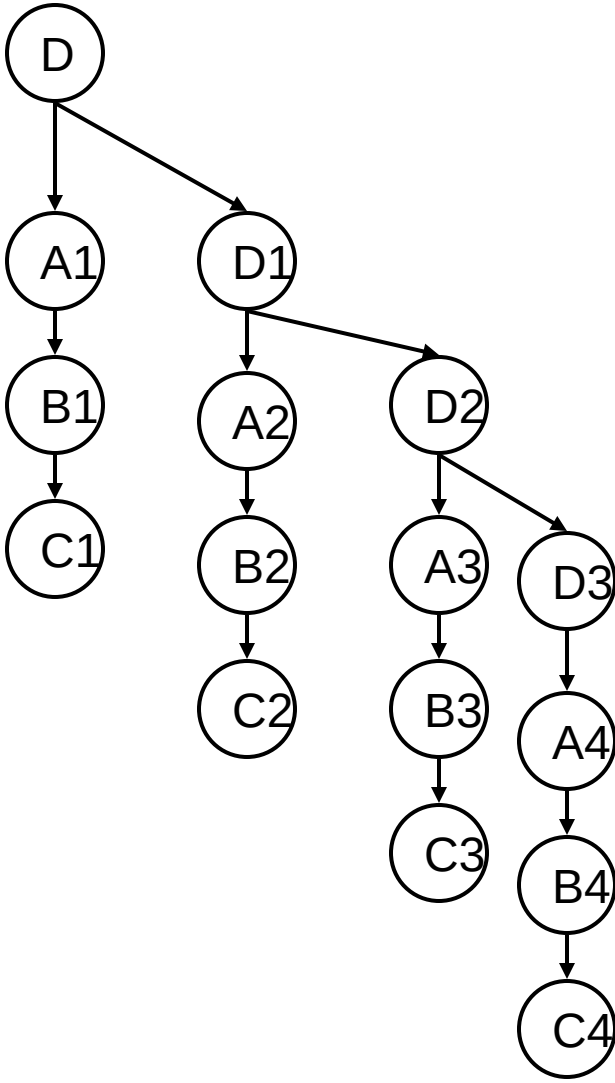
```
A:   a ←    ld [d]
B:   b ←    a * a
C:          st [d], b
D:   d1 ←   d + 4
A1:  a1 ←   ld [d1]
B1:  b1 ←   a1 * a1
C1:         st [d1], b1
D1:  d2 ←   d1 + 4
A2:  a2 ←   ld [d2]
B2:  b2 ←   a2 * a2
C2:         st [d2], b2
D2:  d ←    d2 + 4
```



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

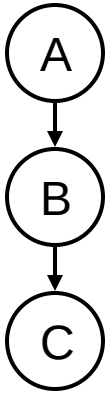
One More Time

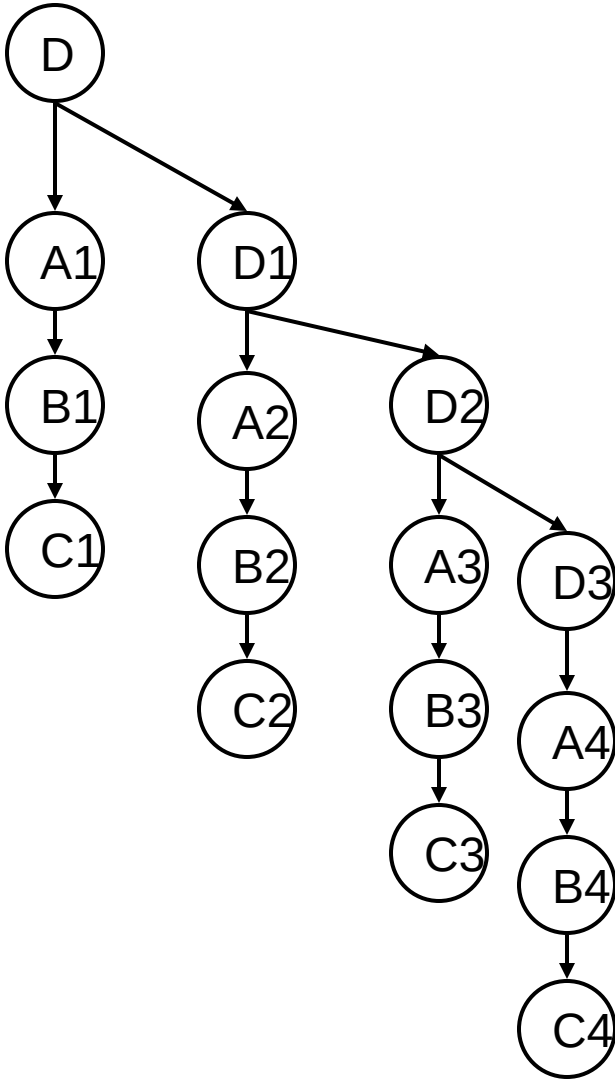




A		B		C		D4			
D		A1		B1		C1			
	D1		A2		B2		C2		
		D2		A3		B3		C3	
			D3		A4		B4		C4

Can Rearrange

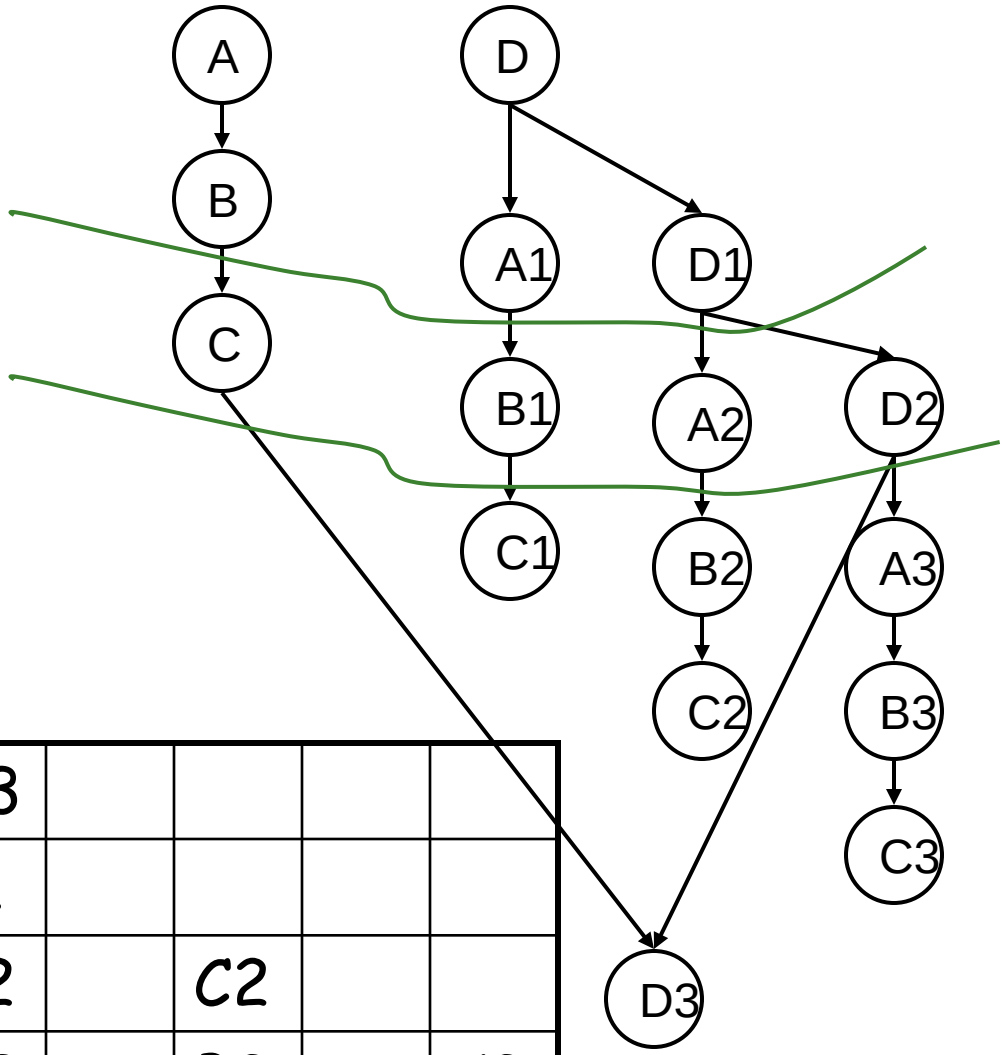




A		B		C		D4			
D		A1		B1		C1			
	D1 →		A2		B2		C2		
		D2 →		A3		B3		C3	
			D3		A4		B4		C4

Rearrange

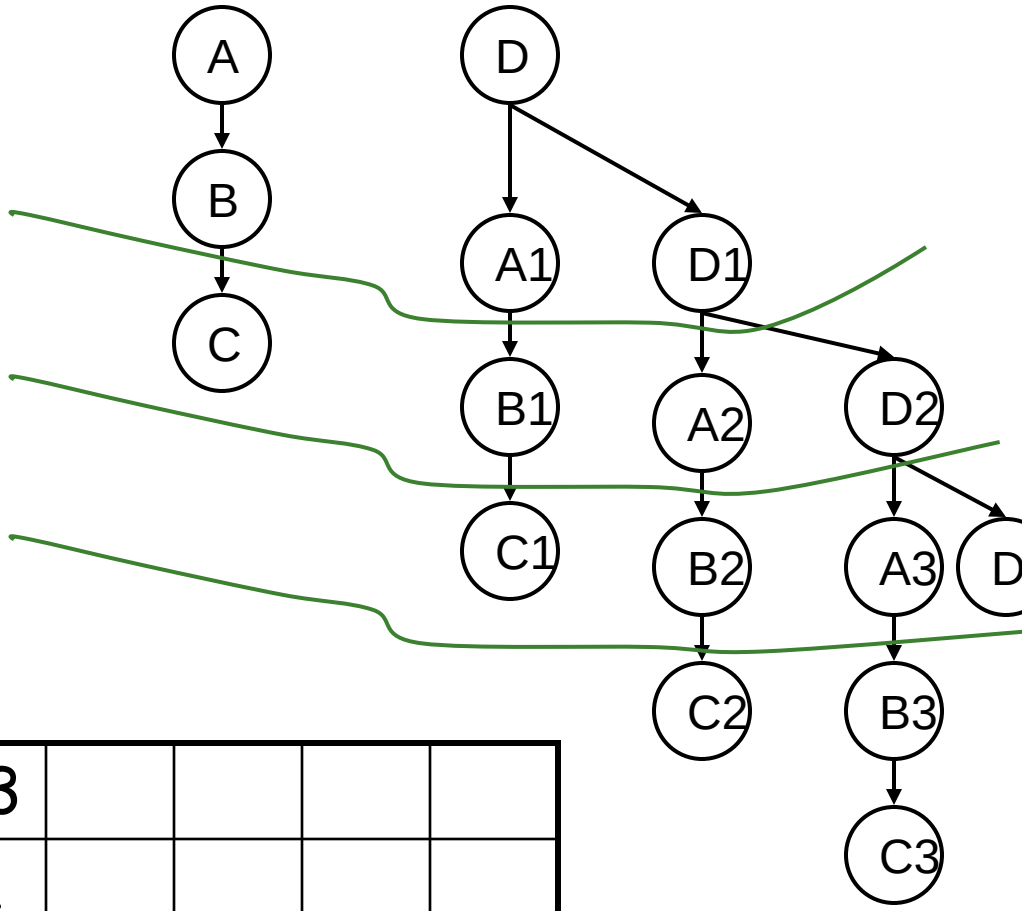
```
A:   a ←    ld [d]
B:   b ←    a * a
C:          st [d], b
D:   d1 ←   d + 4
A1:  a1 ←   ld [d1]
B1:  b1 ←   a1 * a1
C1:         st [d1], b1
D1:  d2 ←   d1 + 4
A2:  a2 ←   ld [d2]
B2:  b2 ←   a2 * a2
C2:         st [d2], b2
D2:  d ←    d2 + 4
```



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

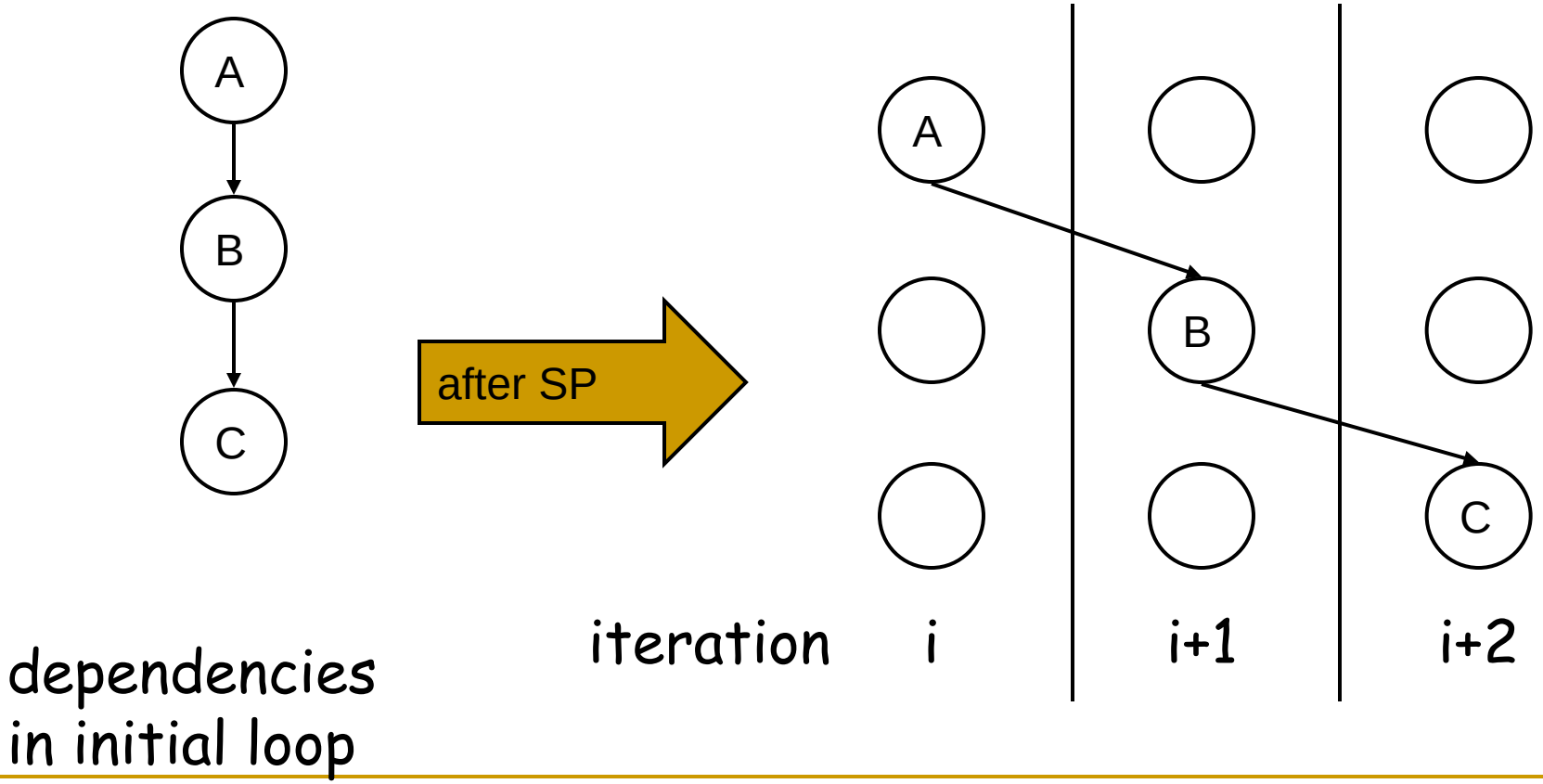
Body

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

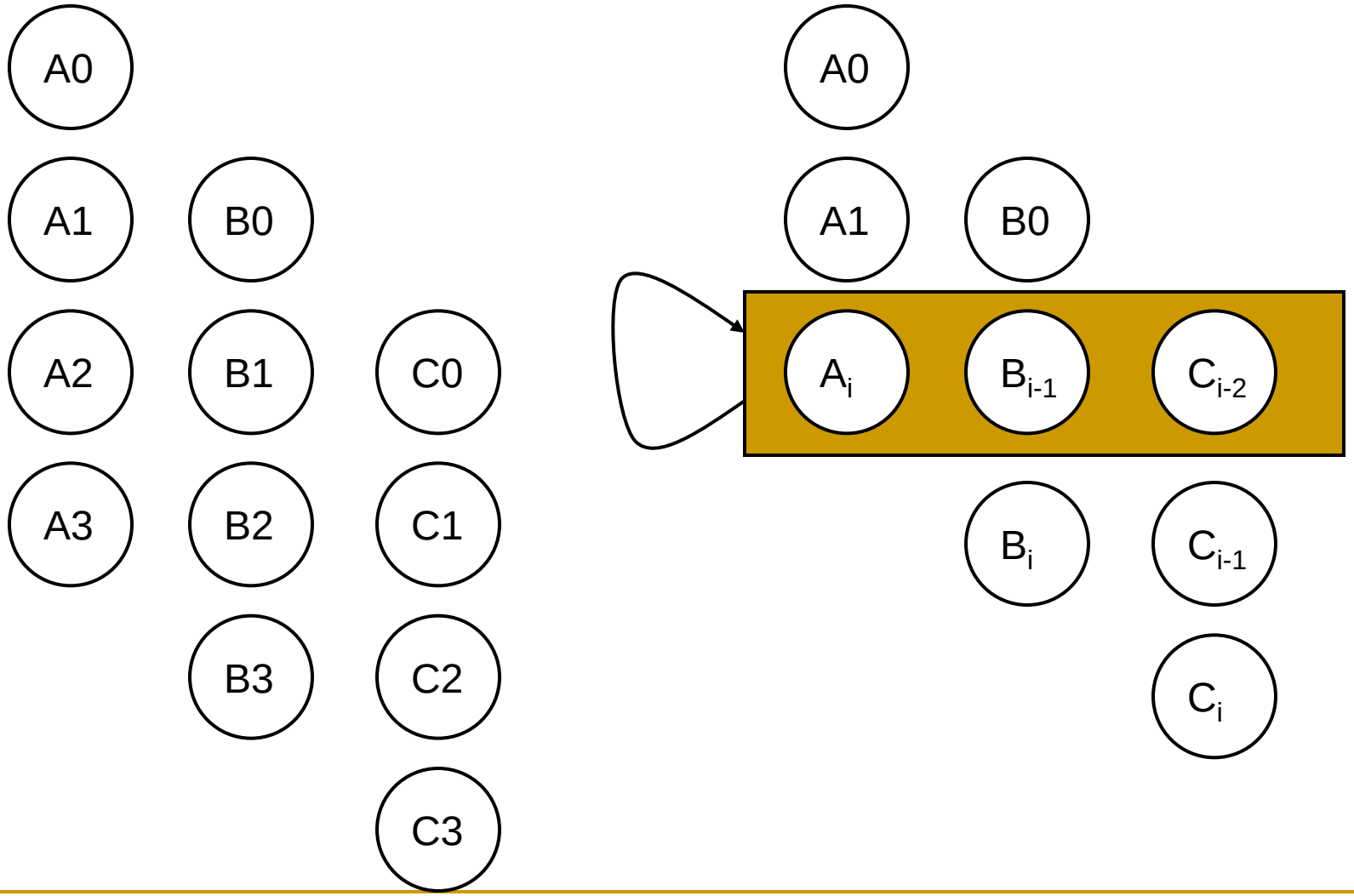


Goal of SP

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

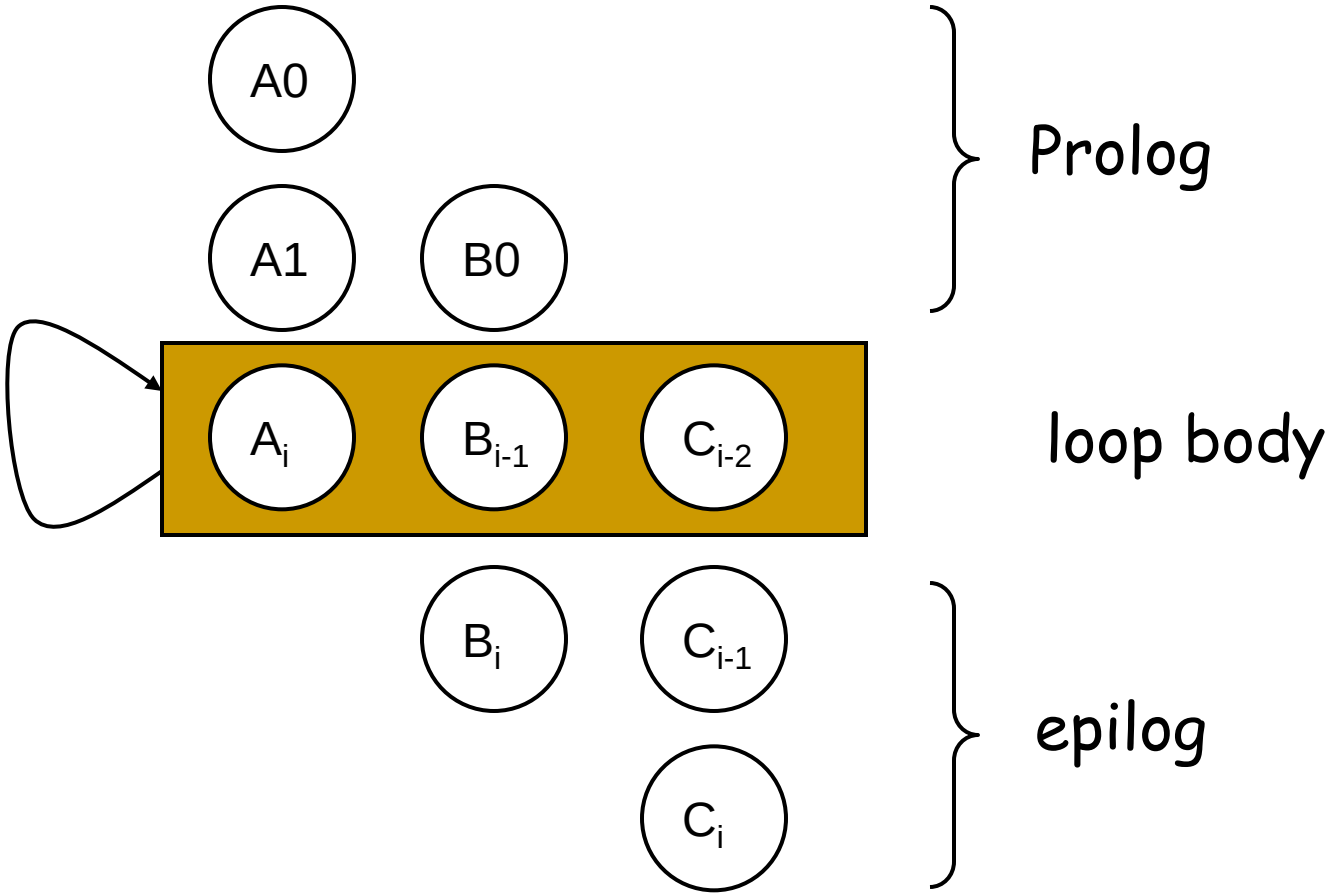
Example

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++)
{
    A_i
    B_i
    C_i
}
```

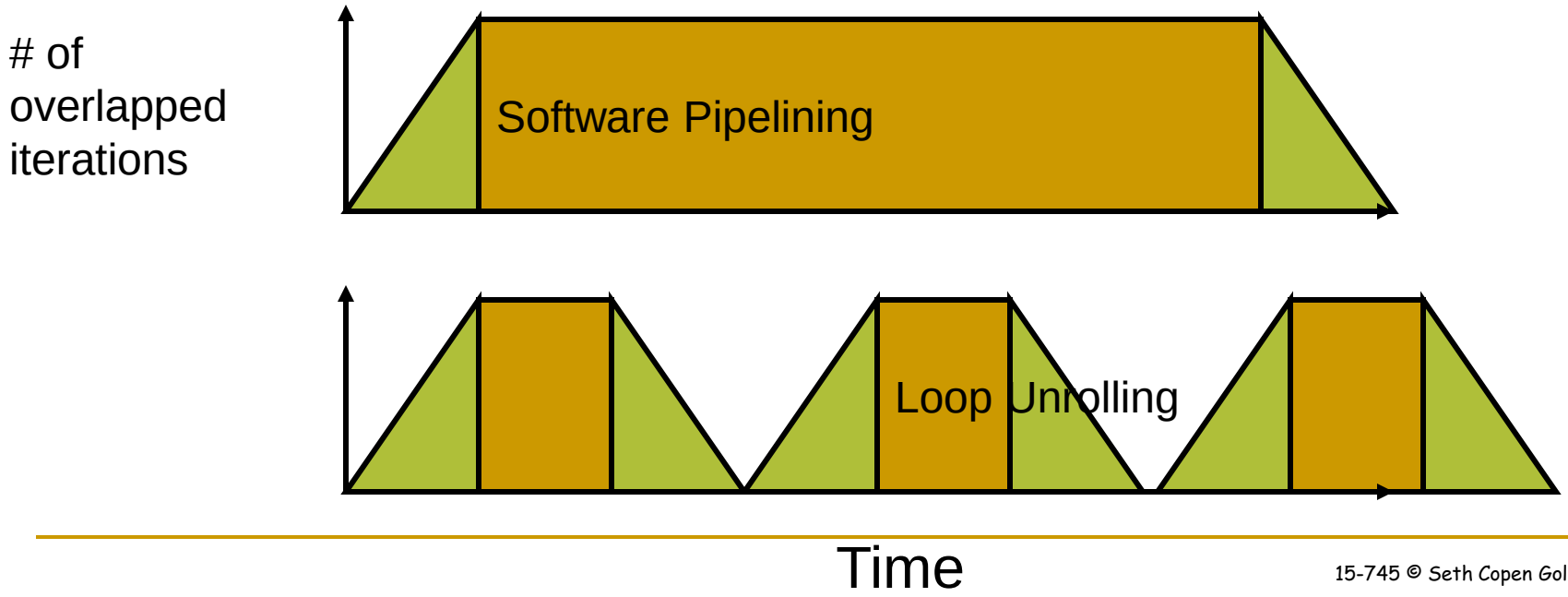
```
i=0
if (i >= N) goto done
A_0
B_0
if (i+1 == N) goto last
i=1
A_1
if (i+2 == N) goto epilog
i=2
```

```
loop:
    A_i
    B_i-1
    C_i-2
    i++
    if (i < N) goto loop
epilog:
    B_i
    C_i-1
last:
    c_i
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?

Small Aside

- Scalar Replacement & Dependencies

Scalar Replacement: Example

```
DO I = 1, N
   DO J = 1, M
       A(I) = A(I) + B(J)
   ENDDO
ENDDO
```

- A(I) can be left in a register throughout the inner loop
- Superscalar + cache will get most of this, but not allocate A(I) to register

```
DO I = 1, N
   T = A(I)
   DO J = 1, M
       T = T + B(J)
   ENDDO
   A(I) = T
ENDDO
```

- All loads and stores to A in the inner loop have been saved
- High chance of T being allocated a register by compiler

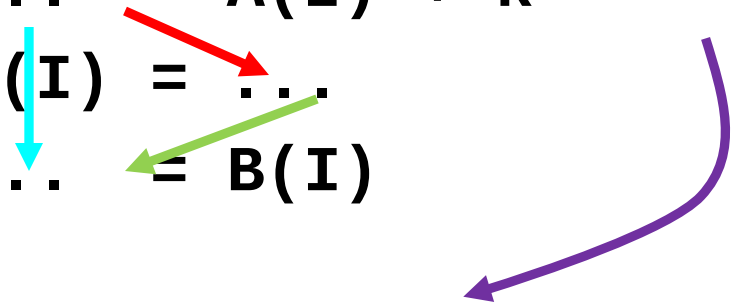
Scalar Replacement

- Convert array reference to scalar reference
- Approach is to use dependences to achieve these memory hierarchy transformations

Dependence and Memory Hierarchy

- True or Flow - save loads and cache miss
- Anti - save cache miss?
- Output - save stores
- Input - save loads

```
A(I) = ... + B(I)
...  = A(I) + k
A(I) = ...
...  = B(I)
```



Dependence and Memory Hierarchy

- Loop Carried dependences - Consistent dependences most useful for memory management purposes
- Consistent dependences - dependences with constant dependence distance

Using Dependences

- In the reduction example

```
DO I = 1, N
   DO J = 1, M
      A(I) = A(I) + B(J)
   ENDDO
ENDDO
```

```
DO I = 1, N
   T = A(I)
   DO J = 1, M
      T = T + B(J)
   ENDDO
   A(I) = T
ENDDO
```

- True dependence - replace the references to A in the inner loop by scalar T
- Output dependence - store can be moved outside the inner loop
- Antidependence - load can be moved before the inner loop

Scalar Replacement

- Example: Scalar Replacement in case of *loop independent dependence*

```
DO I = 1, N
   A(I) = B(I) + C
   X(I) = A(I)*Q
ENDDO
```

```
DO I = 1, N
   t = B(I) + C
   A(I) = t
   X(I) = t*Q
ENDDO
```

- One less load for each iteration for reference to A

Scalar Replacement

- Example: Scalar Replacement in case of *loop carried dependence spanning single iteration*

```
DO I = 1, N
   A(I) = B(I-1)
   B(I) = A(I) +
C(I)
ENDDO
```

```
tB = B(0)
DO I = 1, N
   tA = tB
   A(I) = tA
   tB = tA + C(I)
   B(I) = tB
ENDDO
```

- One less load for each iter for ref to B which had a loop carried true dependence of 1 iter
- Also one less load per iter for reference to A

Scalar Replacement

- Example: Scalar Replacement in case of *loop carried dependence spanning multiple iterations*

```
DO I = 1, N
   A(I) = B(I-1) + B(I+1)
ENDDO
```

```
t1 = B(0)
t2 = B(1)
DO I = 1, N
   t3 = B(I+1)
   A(I) = t1 + t3
   t1 = t2
   t2 = t3
ENDDO
```

- One less load for each iter for ref to B which had a loop carried input dependence of 2 iters
- Invariants maintained were `t1=B(I-1); t2=B(I); t3=B(I+1)`

Aiken/Nicolau Scheduling

Step 1

Perform *scalar replacement* to eliminate memory references where possible.

```
for i:=1 to N do
    a := j ⊕ V[i-1]
    b := a ⊕ f
    c := e ⊕ j
    d := f ⊕ c
    e := b ⊕ d
    f := U[i]
 g: V[i] := b
 h: W[i] := d
    j := X[i]
```

```
for i:=1 to N do
    a := j ⊕ b
    b := a ⊕ f
    c := e ⊕ j
    d := f ⊕ c
    e := b ⊕ d
    f := U[i]
 g: V[i] := b
 h: W[i] := d
    j := X[i]
```

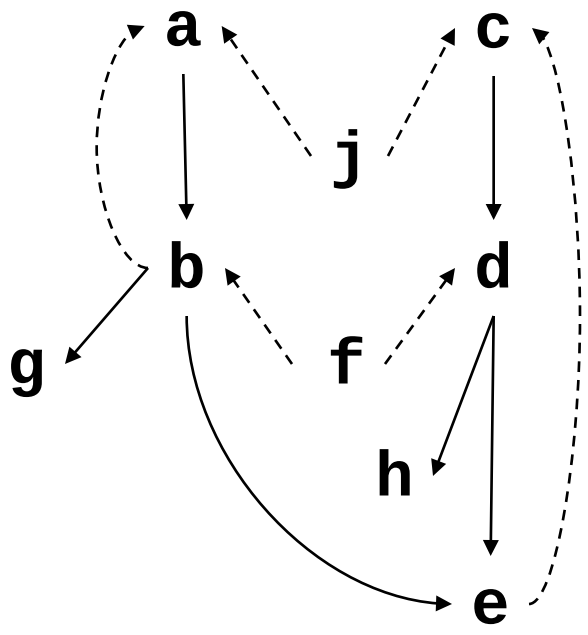
Aiken/Nicolau Scheduling

Step 2

Unroll the loop and compute the data-dependence graph (DDG).

DDG for rolled loop:

```
for i:=1 to N do
    a := j ⊕ b
    b := a ⊕ f
    c := e ⊕ j
    d := f ⊕ c
    e := b ⊕ d
    f := U[i]
 g: V[i] := b
 h: W[i] := d
    j := X[i]
```

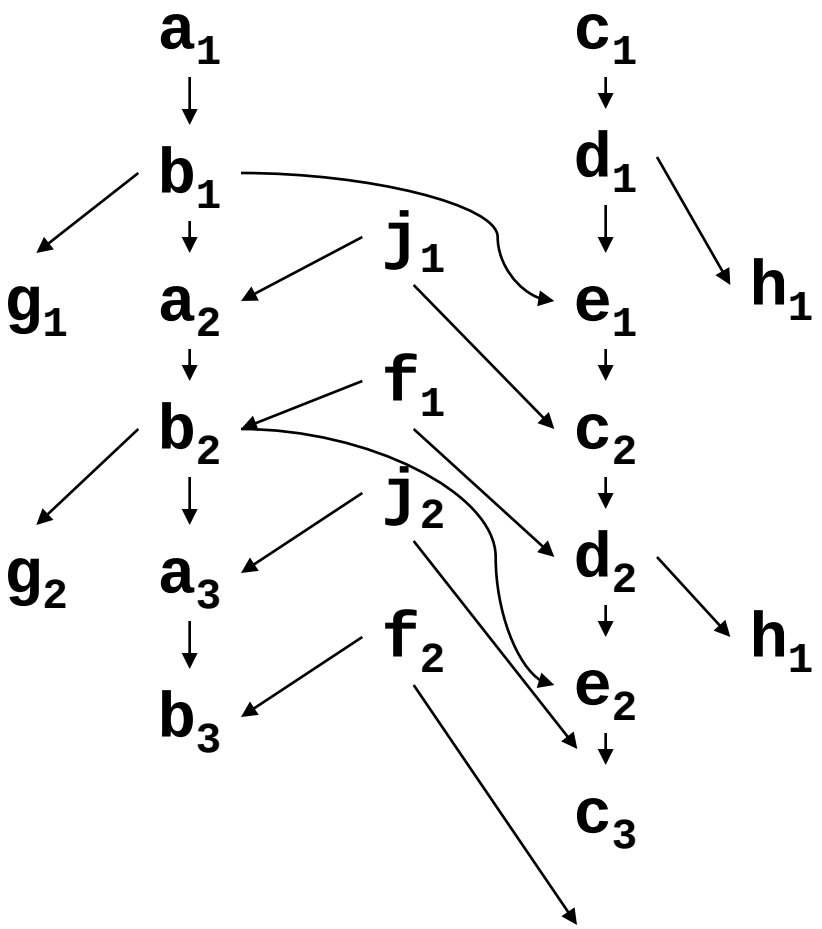


Aiken/Nicolau Scheduling

Step 2, cont'd

DDG for unrolled loop:

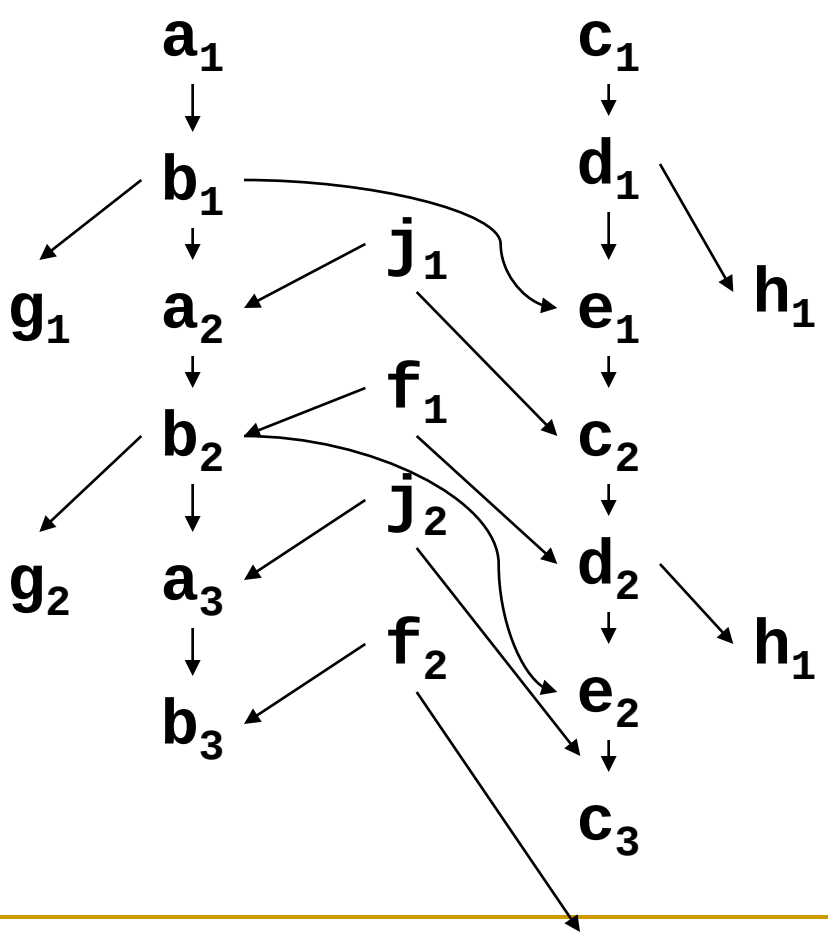
```
for i:=1 to N do
    a := j ⊕ b
    b := a ⊕ f
    c := e ⊕ j
    d := f ⊕ c
    e := b ⊕ d
    f := U[i]
 g: V[i] := b
 h: W[i] := d
    j := X[i]
```



Aiken/Nicolau Scheduling

Step 3

Build a tableau of iteration number vs cycle time.



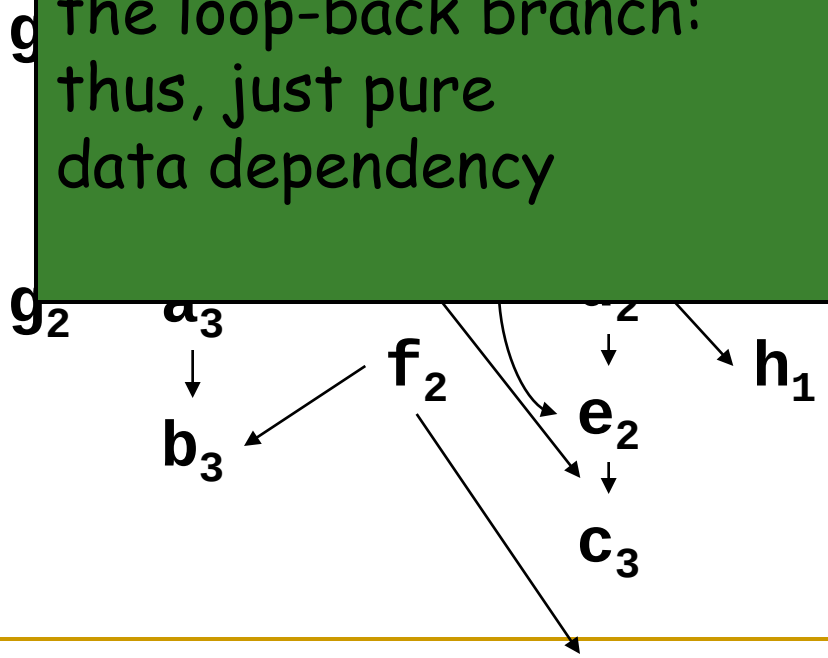
cycle	iteration 1	2	3	4	5	6
1	acfj	fj	fj	fj	fj	fj
2	bd					
3	egh	a				
4		cb				
5		dg	a			
6		eh	b			
7			cg	a		
8			d	b		
9			eh	g	a	
10				c	b	
11				d	g	a
12				eh		b
13					c	g
14					d	
15					eh	

Aiken/Nicolau Scheduling

Step 3

basically, you're emulating a superscalar with infinite resources, infinite register renaming, always predicting the loop-back branch: thus, just pure data dependency

on number vs cycle time.



cycle \ iteration	1	2	3	4	5	6
1	acfj	fj	fj	fj	fj	fj
2	bd					
3	egh	a				
4		cb				
5		dg	a			
6		eh	b			
7			cg	a		
8			d	b		
9			eh	g	a	
10				c	b	
11				d	g	a
12				eh		b
13					c	g
14					d	
15					eh	

Aiken/Nicolau Scheduling

Step 4

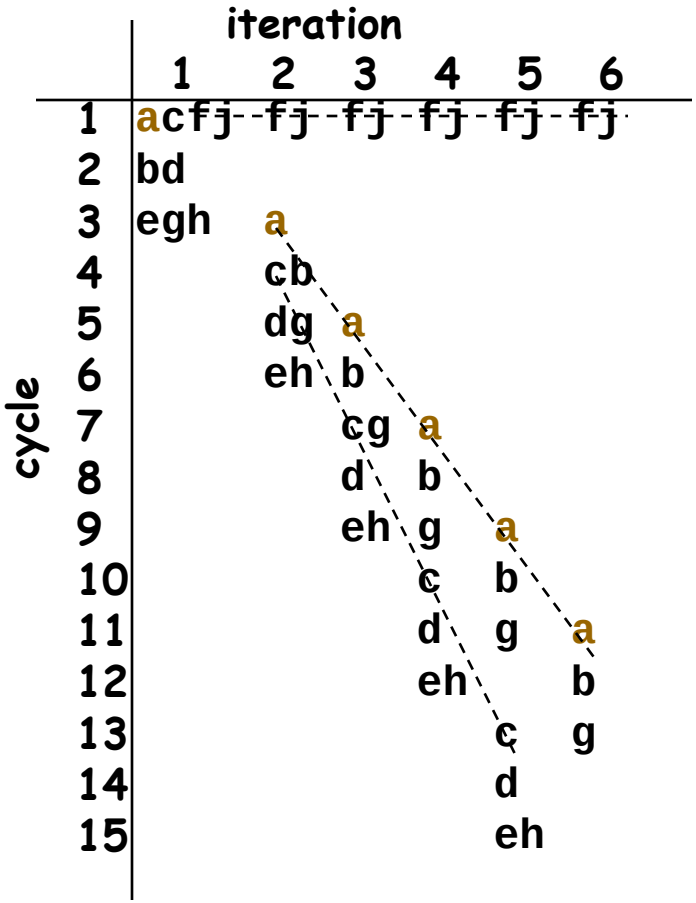
Find repeating patterns of instructions.

cycle	iteration 1	2	3	4	5	6
1	acfj	fj	fj	fj	fj	fj
2	bd					
3	egh	a				
4		cb				
5		dg	a			
6		eh	b			
7			cg	a		
8			d	b		
9			eh	g	a	
10				c	b	
11				d	g	a
12				eh		b
13					c	g
14					d	
15					eh	

Aiken/Nicolau Scheduling

Step 4

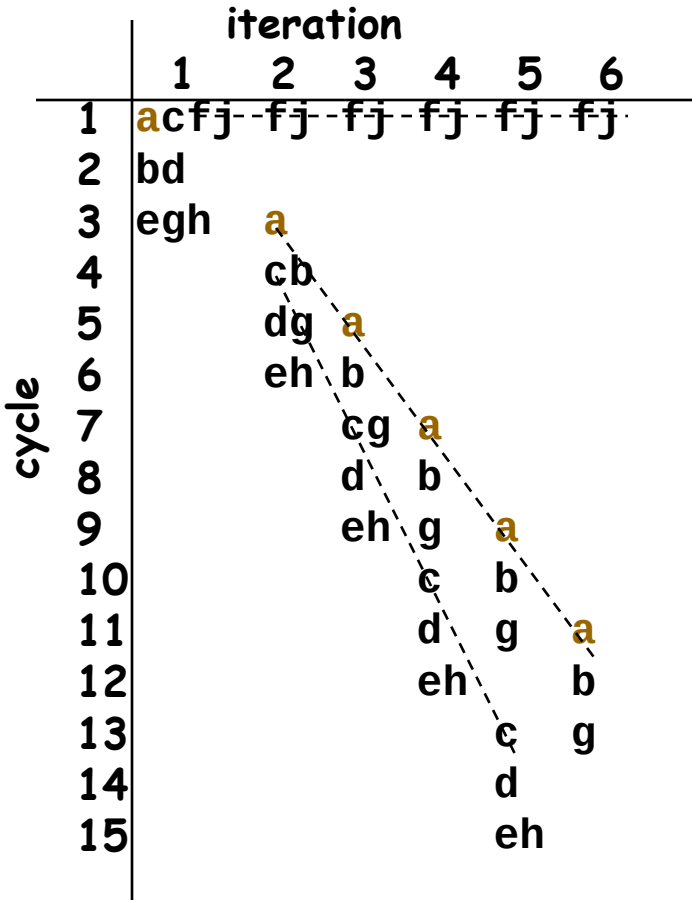
Find repeating patterns of instructions.



Aiken/Nicolau Scheduling

Step 4

Find repeating patterns of instructions.

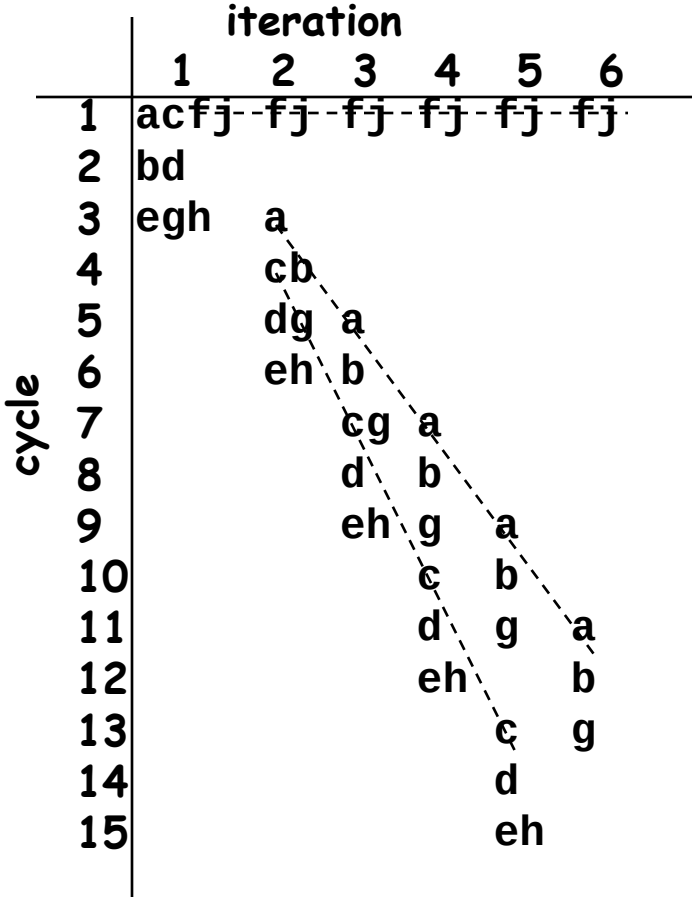


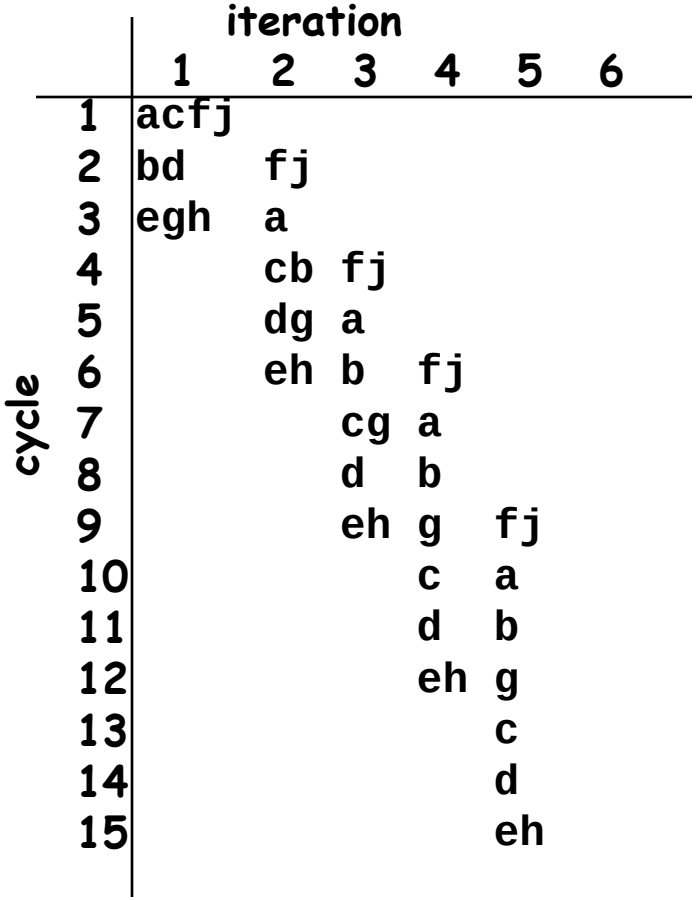
Go back and relate slopes to DDG

Aiken/Nicolau Scheduling

Step 5

"Coalesce" the slopes.





Aiken/Nicolau Scheduling

Step 6

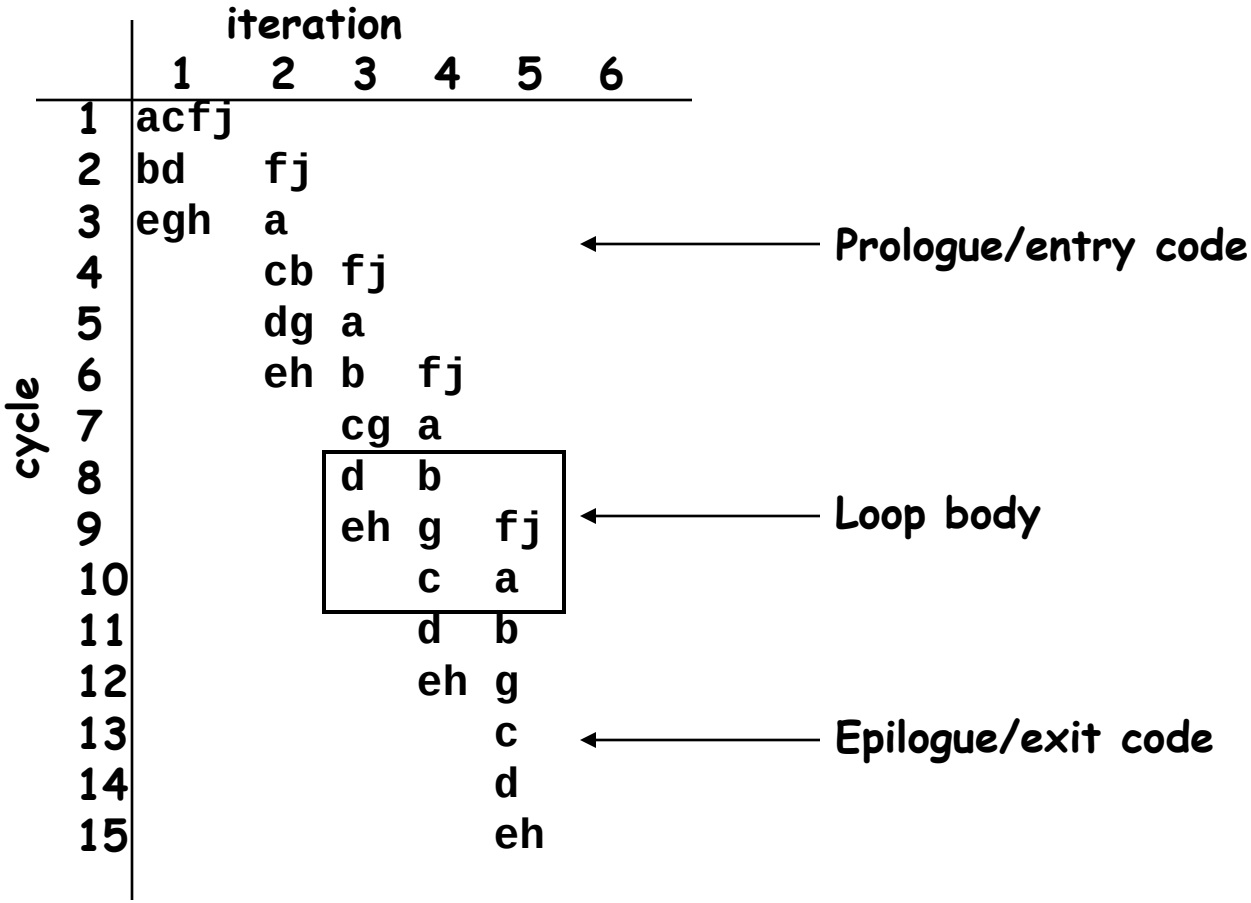
Find the loop body and "reroll" the loop.

cycle	iteration 1	2	3	4	5	6
1	acfj					
2	bd	fj				
3	egh	a				
4		cb	fj			
5		dg	a			
6		eh	b	fj		
7			cg	a		
8			d	b		
9			eh	g	fj	
10				c	a	
11				d	b	
12				eh	g	
13					c	
14					d	
15					eh	

Aiken/Nicolau Scheduling

Step 6

Find the loop body and "reroll" the loop.



Aiken/Nicolau Scheduling

Step 7

Generate code.

(Assume VLIW-like machine for this example. The instructions on each line should be issued in parallel.)

```
   a1 := j0 ⊕ b0     c1 := e0 ⊕ j0     f1 := U[1]      j1 := X[1]
   b1 := a1 ⊕ f0     d1 := f0 ⊕ c1     f2 := U[2]      j2 := X[2]
   e1 := b1 ⊕ d1     V[1] := b1        W[1] := d1      a2 := j1 ⊕ b1
   c2 := e1 ⊕ j1     b2 := a2 ⊕ f1     f3 := U[3]      j3 := X[3]
   d2 := f1 ⊕ c2     V[2] := b2        a3 := j2 ⊕ b2
   e2 := b2 ⊕ d2     W[2] := d2        b3 := a3 ⊕ f2   f4 := U[4]     j4 := X[4]
   c3 := e2 ⊕ j2     V[3] := b3        a4 := j3 ⊕ b3    i := 3
L:
   d_i := f_i-1 ⊕ c_i     b_i+1 := a_i ⊕ f_i
   e_i := b_i ⊕ d_i       W[i] := d_i       V[i+1] := b_i+1   f_i+2 := U[I+2]   j_i+2 := X[i+2]
   c_i+1 := e_i ⊕ j_i     a_i+2 := j_i+1 ⊕ b_i+1  i := i+1    if i<N-2 goto L

   d_N-1 := f_N-2 ⊕ c_N-1  b_N := a_N ⊕ f_N-1
   e_N-1 := b_N-1 ⊕ d_N-1  W[N-1] := d_N-1   v[N] := b_N
   c_N := e_N-1 ⊕ j_N-1
   d_N := f_N-1 + c_N
   e_N := b_N ⊕ d_N        w[N] := d_N
```

Aiken/Nicolau Scheduling

Step 8

- Since several versions of a variable (e.g., j_i and j_{i+1}) might be live simultaneously, we need to add new temps and moves

```
   a1 := j0 ⊕ b0     c1 := e0 ⊕ j0     f1 := U[1]      j1 := X[1]
   b1 := a1 ⊕ f0     d1 := f0 ⊕ c1     f2 := U[2]      j2 := X[2]
   e1 := b1 ⊕ d1     V[1] := b1        W[1] := d1      a2 := j1 ⊕ b1
   c2 := e1 ⊕ j1     b2 := a2 ⊕ f1     f3 := U[3]      j3 := X[3]
   d2 := f1 ⊕ c2     V[2] := b2        a3 := j2 ⊕ b2
   e2 := b2 ⊕ d2     W[2] := d2        b3 := a3 ⊕ f2   f4 := U[4]    j4 := X[4]
   c3 := e2 ⊕ j2     V[3] := b3        a4 := j3 ⊕ b3    i := 3
L:
   d_i := f_i-1 ⊕ c_i     b_i+1 := a_i ⊕ f_i
   e_i := b_i ⊕ d_i       W[i] := d_i       V[i+1] := b_i+1  f_i+2 := U[I+2]  j_i+2 := X[i+2]
   c_i+1 := e_i ⊕ j_i     a_i+2 := j_i+1 ⊕ b_i+1 i := i+1    if i<N-2 goto L

   d_N-1 := f_N-2 ⊕ c_N-1 b_N := a_N ⊕ f_N-1
   e_N-1 := b_N-1 ⊕ d_N-1 W[N-1] := d_N-1   v[N] := b_N
   c_N := e_N-1 ⊕ j_N-1
   d_N := f_N-1 + c_N
   e_N := b_N ⊕ d_N       w[N] := d_N
```

Aiken/Nicolau Scheduling

Step 8

- Since several versions of a variable (e.g., j_i and j_{i+1}) might be live simultaneously, we need to add new temps and moves

```
   a1 := j0 ⊕ b0    c1 := e0 ⊕ j0    f1 := U[1]     j1 := X[1]
   b1 := a1 ⊕ f0    d1 := f0 ⊕ c1    f'' := U[2]     j2 := X[2]
   e1 := b1 ⊕ d1    V[1] := b1       W[1] := d1     a2 := j1 ⊕ b1
   c2 := e1 ⊕ j1    b2 := a2 ⊕ f1    f' := U[3]     j' := X[3]
   d2 := f1 ⊕ c2    V[2] := b2       a3 := j2 ⊕ b2
   e2 := b2 ⊕ d2    W[2] := d2       b3 := a3 ⊕ f''  f4 := U[4]    j4 := X[4]
   c3 := e2 ⊕ j2    V[3] := b3       a4 := j' ⊕ b3   i := 3
L:
   d_i := f'' ⊕ c_i    b_{i+1} := a' ⊕ f'   b' := b; a'=a; f''=f'; f'=f; j''=j'; j'=j
   e_i := b' ⊕ d_i     W[i] := d_i          V[i+1] := b_{i+1}  f_{i+2} := U[I+2]  j_{i+2} := X[i+2]
   c_{i+1} := e_i ⊕ j'    a_{i+2} := j'' ⊕ b_{i+1}  i := i+1      if i<N-2 goto L

   d_{N-1} := f_{N-2} ⊕ c_{N-1}  b_N := a_N ⊕ f_{N-1}
   e_{N-1} := b_{N-1} ⊕ d_{N-1}  W[N-1] := d_{N-1}   v[N] := b_N
   c_N := e_{N-1} ⊕ j_{N-1}
   d_N := f_{N-1} + c_N
   e_N := b_N ⊕ d_N      w[N] := d_N
```

Next Step in SP

- AN88 did not deal with resource constraints.
- Modulo Scheduling is a SP algorithm that does.
- It schedules the loop based on
  - resource constraints
  - precedence constraints
- Basically, it's list scheduling that takes into account resource conflicts from overlapping iterations
- Look at original motivation: Systolic Arrays

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
  - Different people work on the same car
  - Many cars are assembled simultaneously
  - Can be two-dimensional

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

Systolic Architectures

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

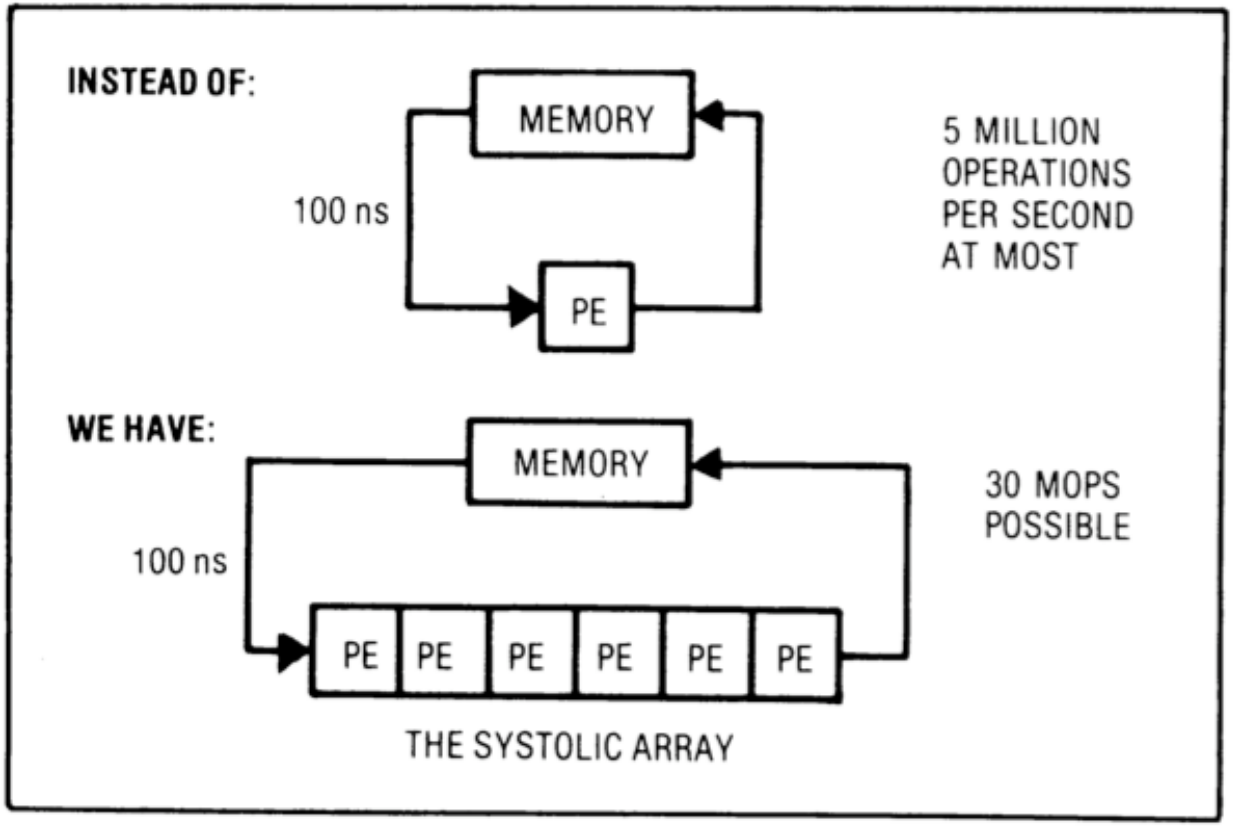


Figure 1. Basic principle of a systolic system.

Memory: heart
PEs: cells

Memory pulses data through cells

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

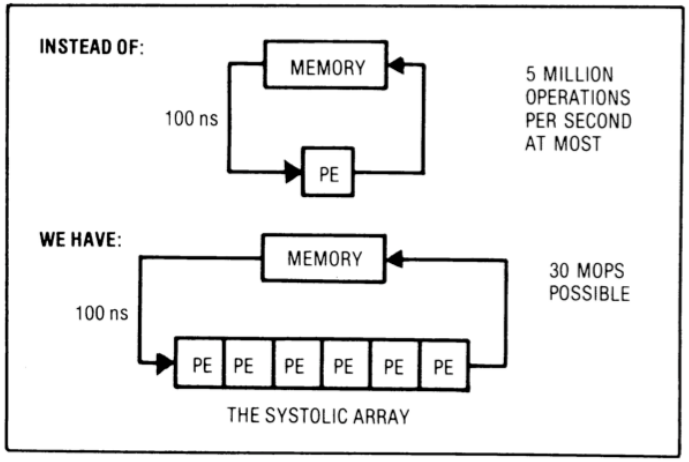


Figure 1. Basic principle of a systolic system.

- 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)

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

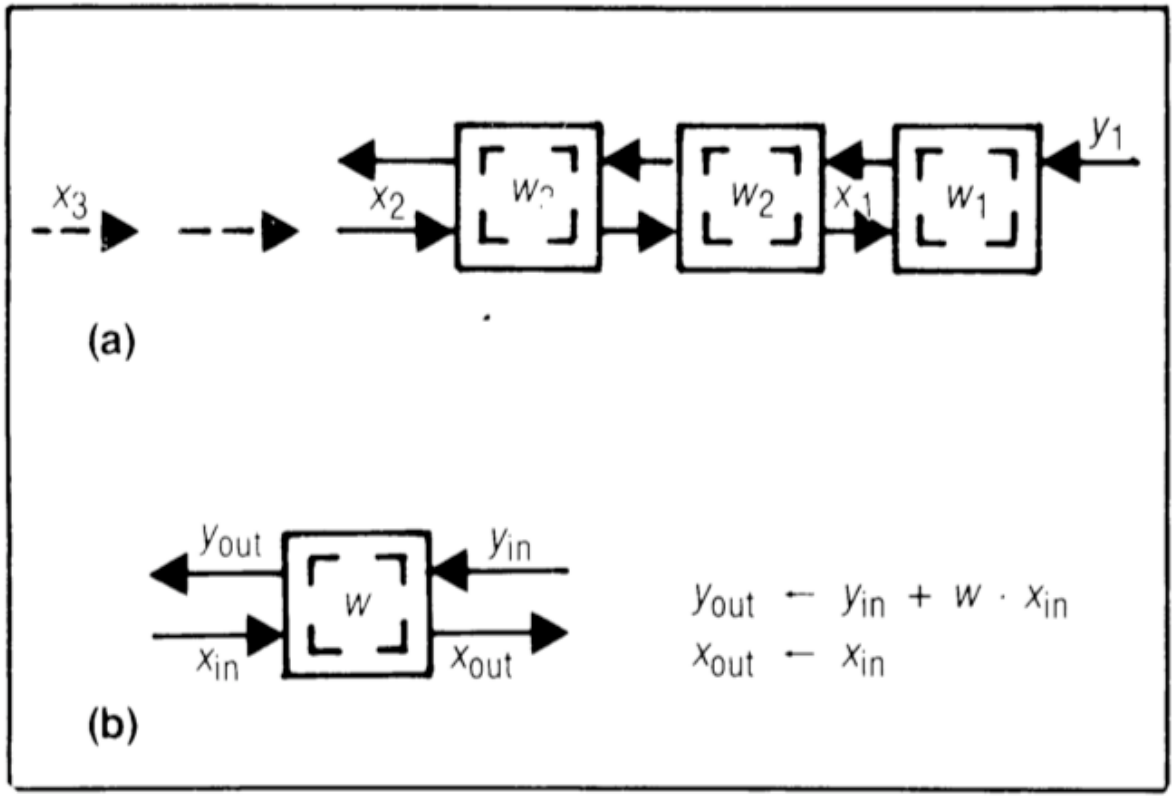


Figure 8. Design W1: systolic convolution array (a) and cell (b) where w_i's stay and x_i's and y_i's move systolically in opposite directions.

Systolic Computation Example: Convolution

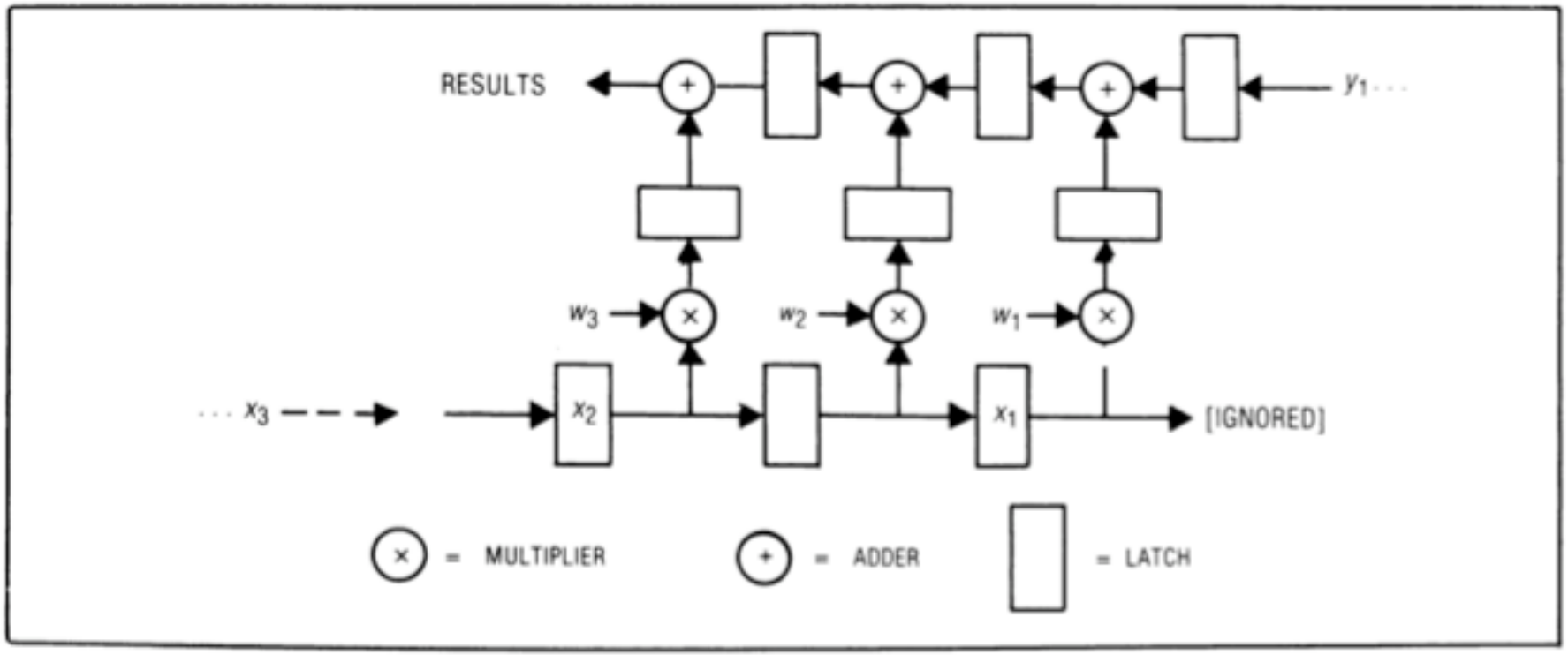


Figure 10. Overlapping the executions of multiply and add in design W1.

to allow overlapping of add/mul executions

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

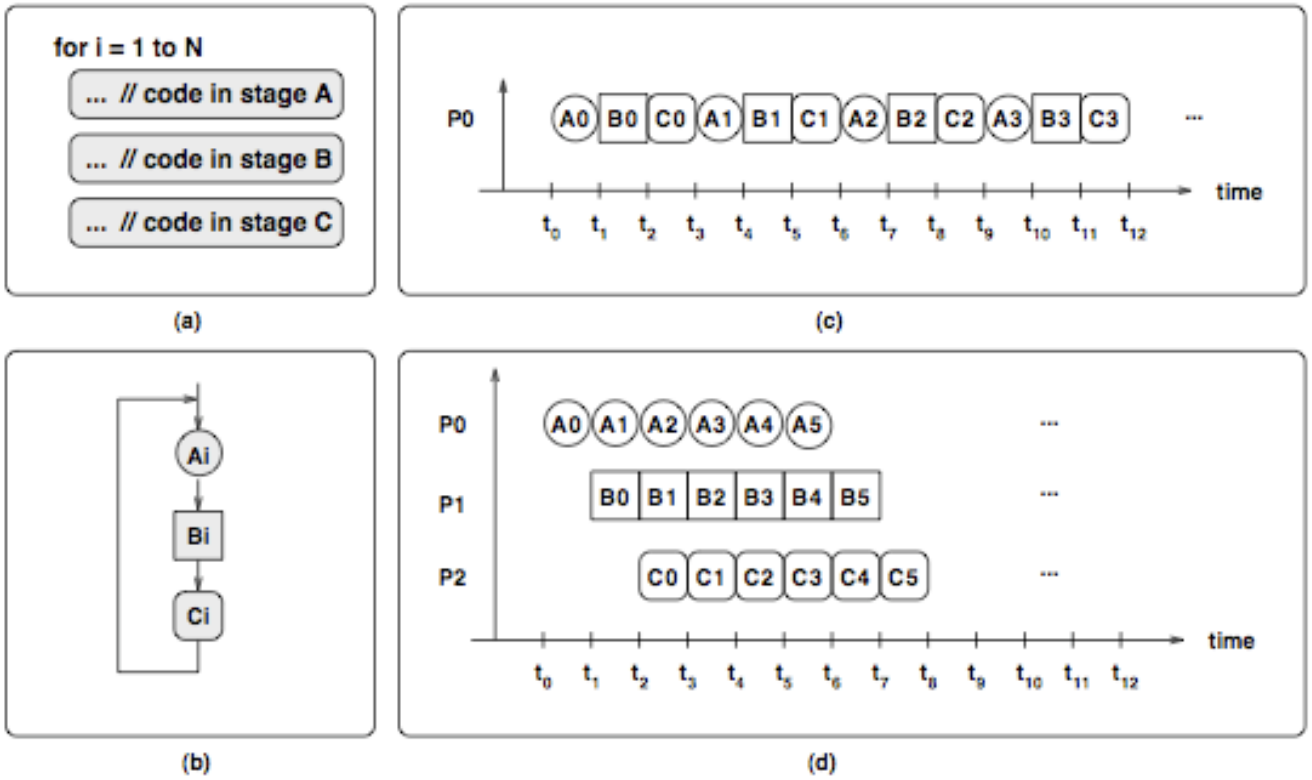


Figure 1. (a) The code of a loop, (b) Each iteration is split into 3 pipeline stages: A, B, and C. Iteration i comprises Ai, Bi, Ci. (c) Sequential execution of 4 iterations. (d) Parallel execution of 6 iterations using pipeline parallelism on a three-core machine. Each stage executes on one core.

File Compression Example

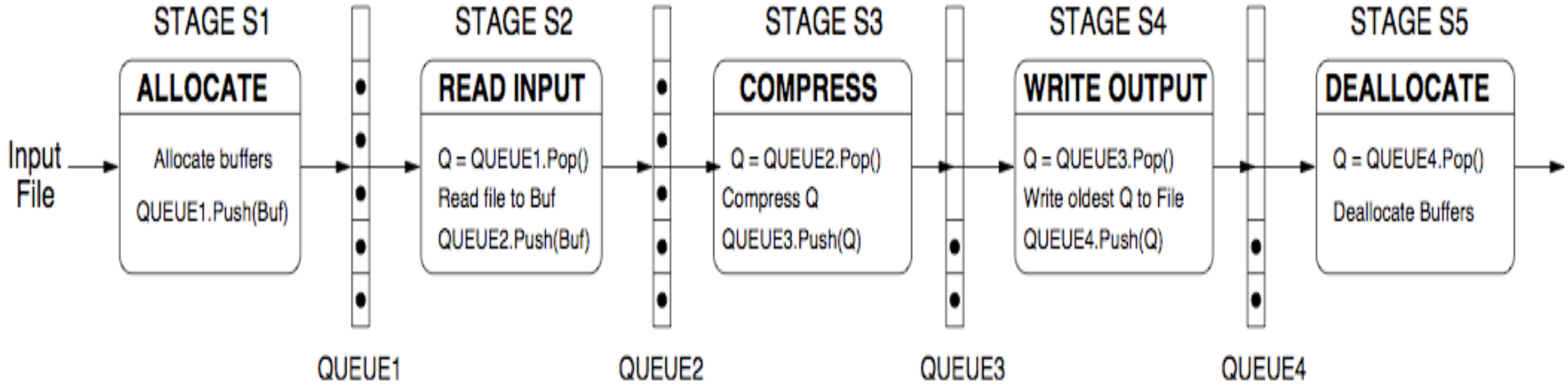


Figure 3. File compression algorithm executed using pipeline parallelism

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

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

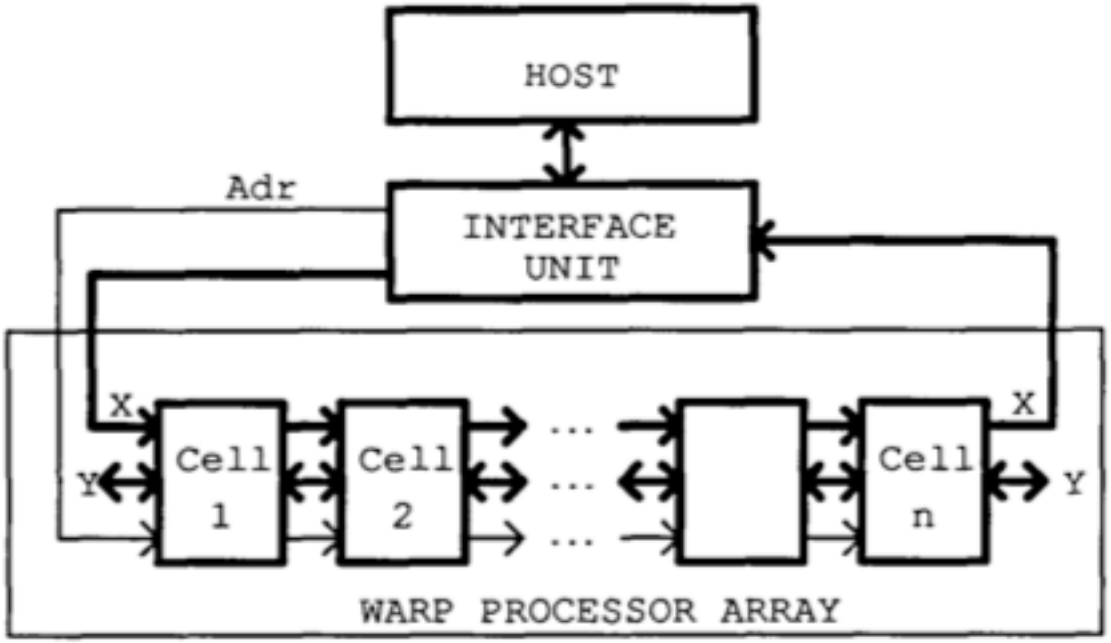
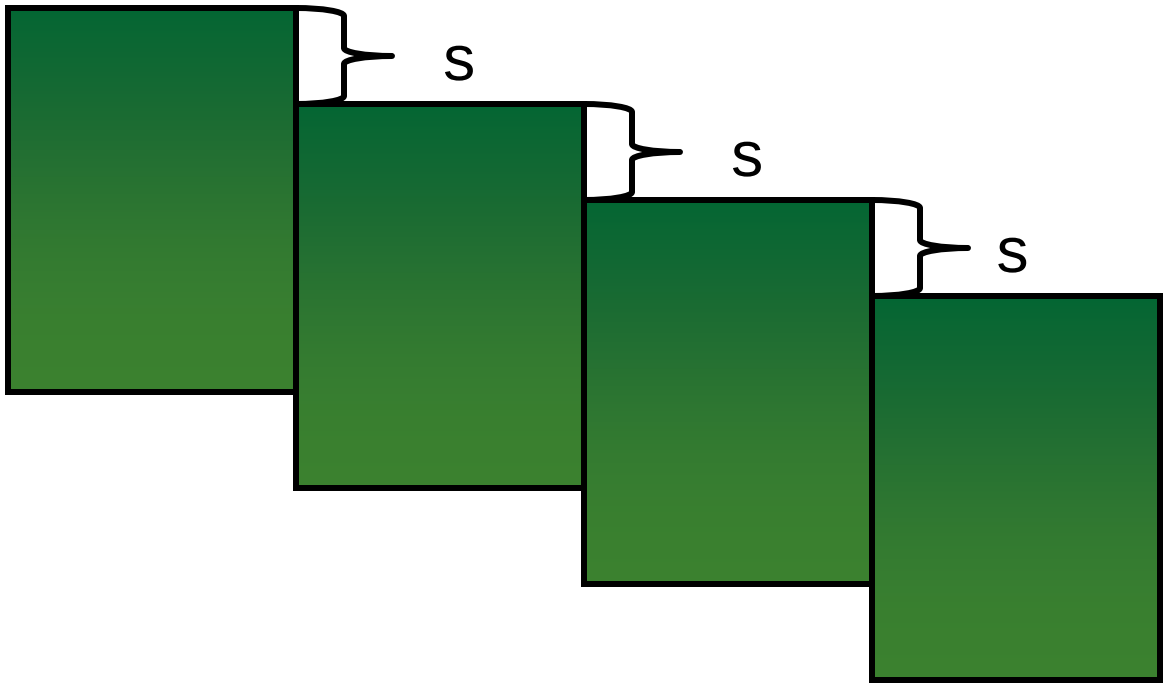


Figure 1: Warp system overview

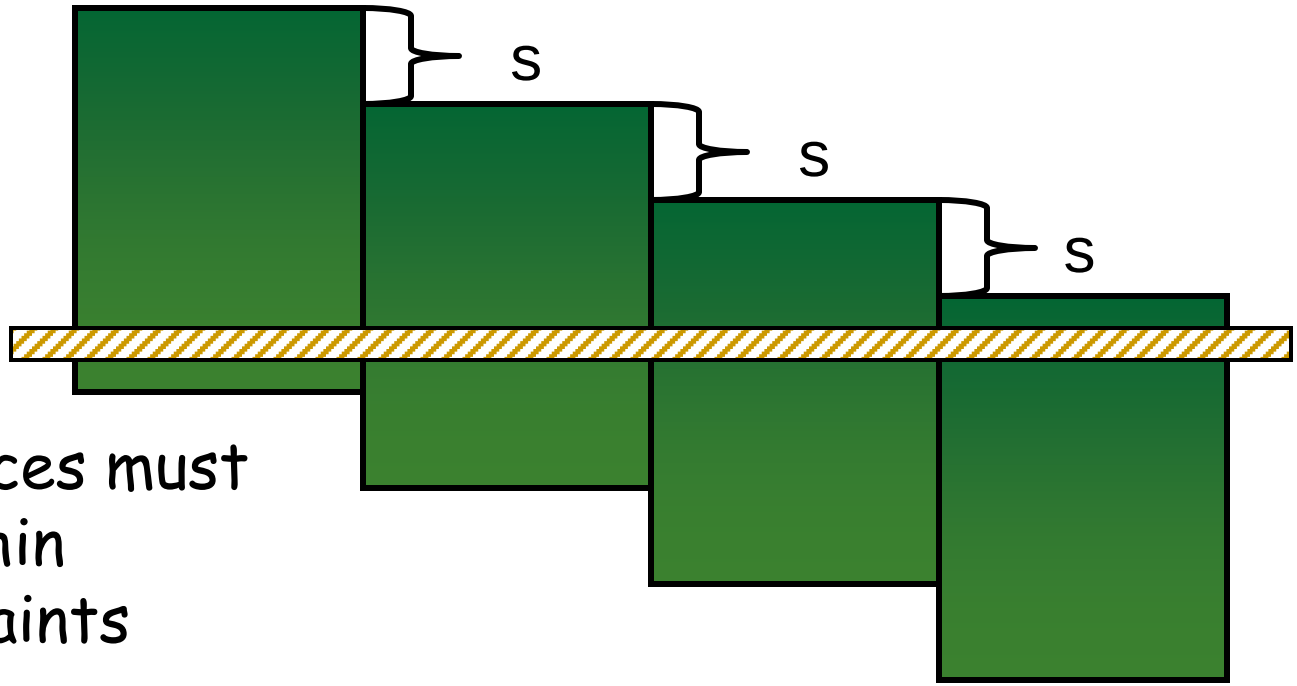
Software Pipelining Goal

- Find the same schedule for each iteration.
- Stagger by iteration initiation interval, *s*
- Goal: minimize *s.*



Software Pipelining Goal

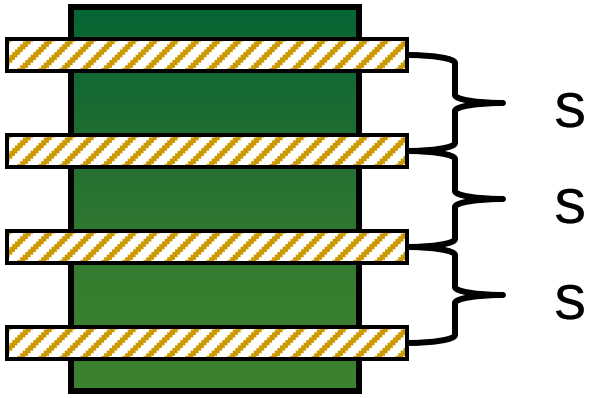
- Find the same schedule for each iteration.
- Stagger by iteration initiation interval, *s*
- Goal: minimize *s.*



resources must be within constraints

Software Pipelining Goal

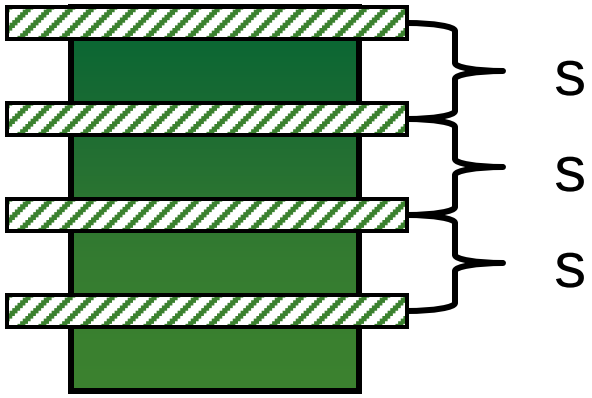
- Find the same schedule for each iteration.
- Stagger by iteration initiation interval, *s*
- Goal: minimize *s.*



resources must
be within
constraints

Software Pipelining Goal

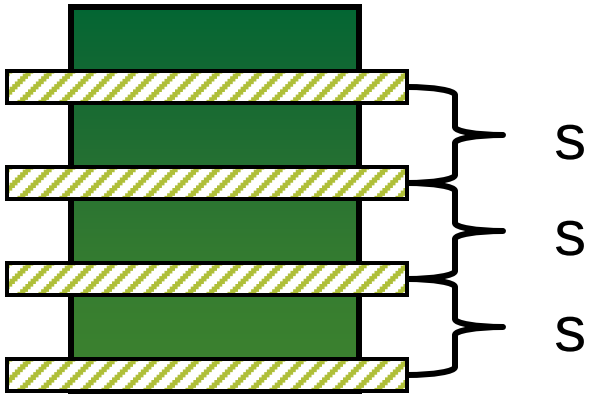
- Find the same schedule for each iteration.
- Stagger by iteration initiation interval, *s*
- Goal: minimize *s.*



resources must
be within
constraints

Software Pipelining Goal

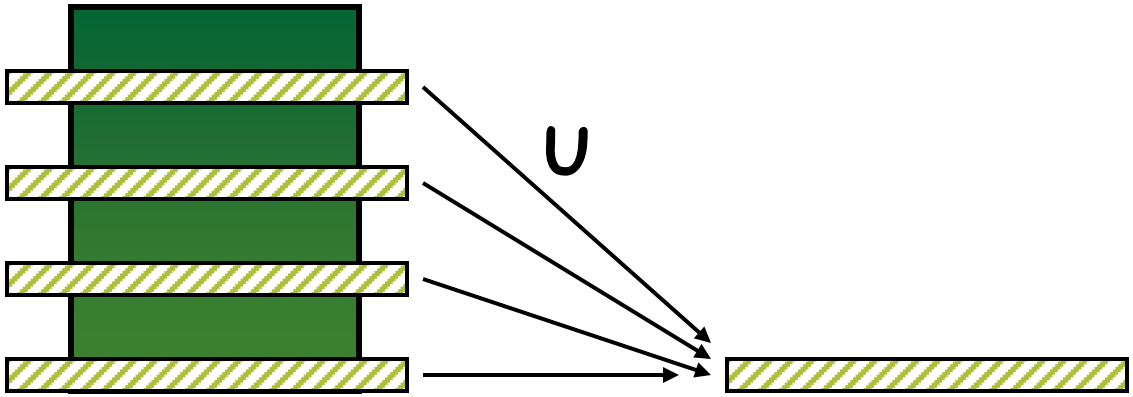
- Find the same schedule for each iteration.
- Stagger by iteration initiation interval, *s*
- Goal: minimize *s.*



resources must
be within
constraints

Software Pipelining Goal

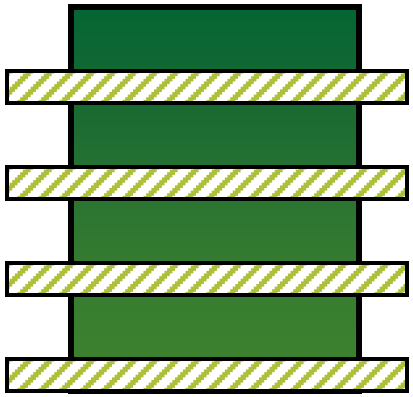
- Find the same schedule for each iteration.
- Stagger by iteration initiation interval, *s*
- Goal: minimize *s.*



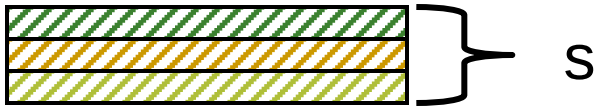
resources must
be within
constraints

Software Pipelining Goal

- Find the same schedule for each iteration.
- Stagger by iteration initiation interval, *s*
- Goal: minimize *s.*



resources must be within constraints



modulo resource table

Precedence Constraints

- Review: for acyclic scheduling, constraint is just the required delay between two ops u, v: <d(u,v)>

- For an edge, u$\rightarrow$v, we must have

  $\sigma(v)-\sigma(u) \geq d(u,v)$

Precedence Constraints

- Cyclic: constraint becomes a tuple: <p,d>
  - p is the minimum iteration delay
    (or the loop carried dependence distance)
  - d is the delay
- For an edge, u$\rightarrow$v, we must have

  $\sigma(v)-\sigma(u) \geq d(u,v)-s*p(u,v)$
- $p \geq 0$
- If data dependence is
  - within an iteration, p=0
  - loop-carried across p iter boundaries, p>0

Iterative Approach

- Finding minimum S that satisfies the constraints is NP-Complete.
- Heuristic:
  - Find lower and upper bounds for S
  - foreach s from lower to upper bound?
    - Schedule graph.
    - If succeed, done
    - Otherwise try again (with next higher s)
- Thus: “Iterative Modulo Scheduling” Rau, et.al.

Iterative Approach

- Heuristic:
  - Find lower and upper bounds for S
  - foreach s from lower to upper bound
    - Schedule graph.
    - If succeed, done
    - Otherwise try again (with next higher s)

- So the key difference:
  - AN88 does not assume S when scheduling
  - IMS must assume an S for each scheduling attempt to understand resource conflicts

Lower Bounds

- Resource Constraints: S_R (also called II_{res})
  maximum over all resources of # of uses divided by # available...

- Precedence Constraints: S_E (also called II_{rec})
  max over all cycles: d(c)/p(c)

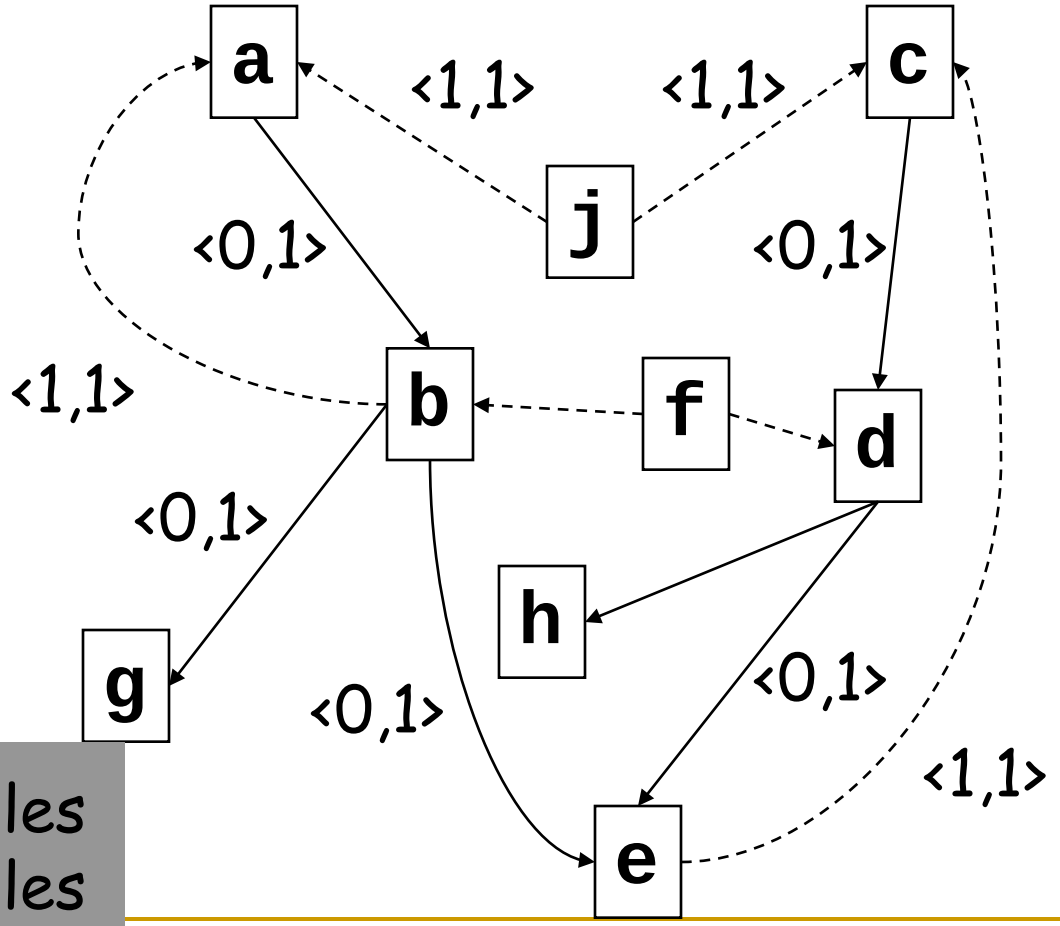
In practice, one is easy, other is hard.

Tim's secret approach: just use S_R as lower bound, then do binary search for best S

Lower Bound on s

- Assume 1 ALU and 1 MU
- Assume latency Op or load is 1 cycle

```
for i:=1 to N do
    a := j ⊕ b
    b := a ⊕ f
    c := e ⊕ j
    d := f ⊕ c
    e := b ⊕ d
    f := U[i]
 g: V[i] := b
 h: W[i] := d
    j := X[i]
```



Resources => 5 cycles
Dependencies => 3 cycles

Scheduling data structures

To schedule for initiation interval s:

- Create a resource table with s rows and R columns
- Create a vector, σ, of length N for n instructions in the loop
  - $\sigma[n]$ = the time at which n is scheduled, or NONE
- Prioritize instructions by some heuristic
  - critical path (or cycle)
  - resource critical

Scheduling algorithm

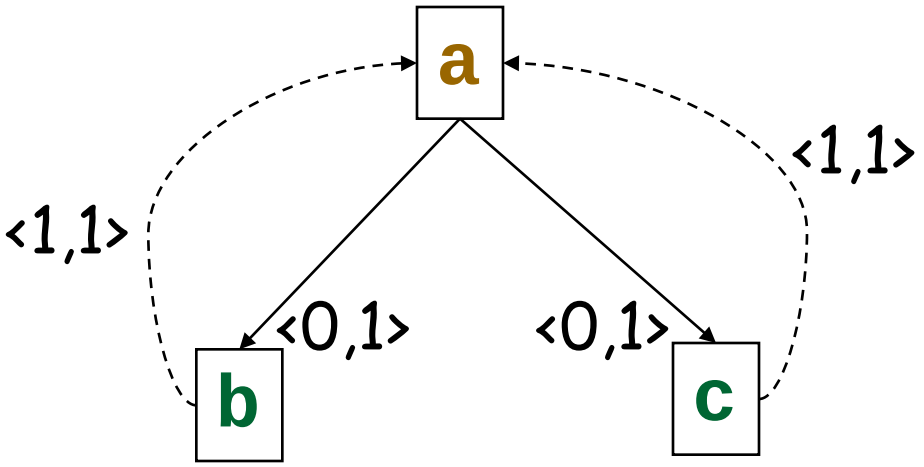
- Pick an instruction, n
- Calculate earliest time due to dependence constraints
  For all x=pred(n),
  earliest = max(earliest, σ(x)+d(x,n)-s·p(x,n))
- try and schedule n from earliest to (earliest+s-1)
  s.t. resource constraints are obeyed.
  - possible twist: deschedule a conflicting node to make way for n, maybe randomly, like sim anneal
- If we fail, then this schedule is faulty
  (i.e. give up on this s)

Scheduling algorithm – cont.

- We now schedule n at earliest, I.e., $\sigma(n)$ = earliest
- Fix up schedule
  - Successors, x, of n must be scheduled s.t.
    $\sigma(x) >= \sigma(n)+d(n,x)-s\cdot p(n,x)$, otherwise they are removed (descheduled) and put back on worklist.
- repeat this some number of times until either
  - succeed, then register allocate
  - fail, then increase s

Simplest Example

```
for () {
   a = b+c
   b = a*a
   c = a*194
}
```

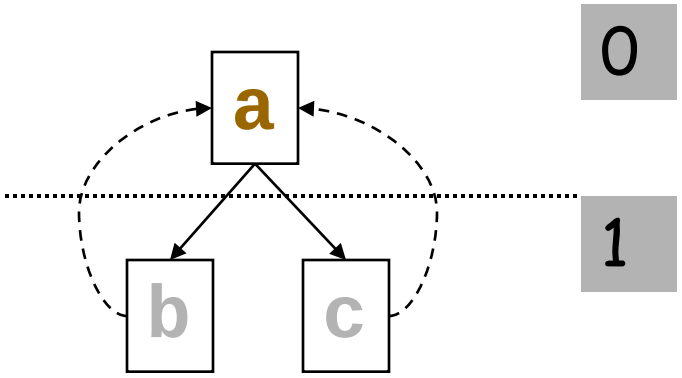


Resources: 1 1

What is IIres?
What is IIrec?

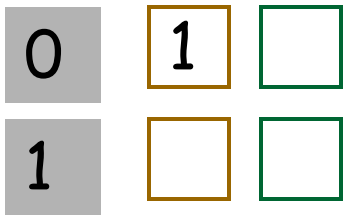
Simplest Example

```
for () {
   a = b+c
   b = a*a
   c = a*194
}
```



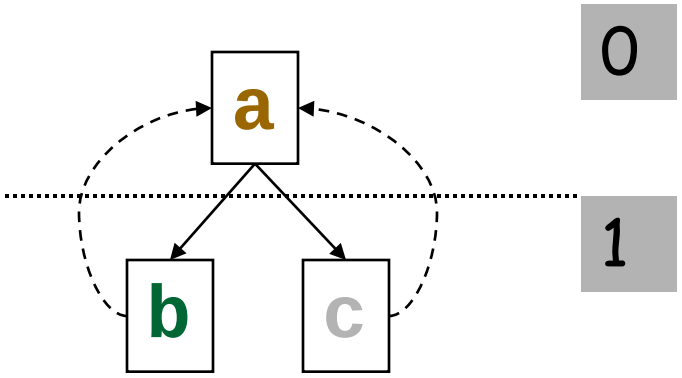
Try II = 2

Modulo Resource Table:



Simplest Example

```
for () {
   a = b+c
   b = a*a
   c = a*194
}
```



Try II = 2

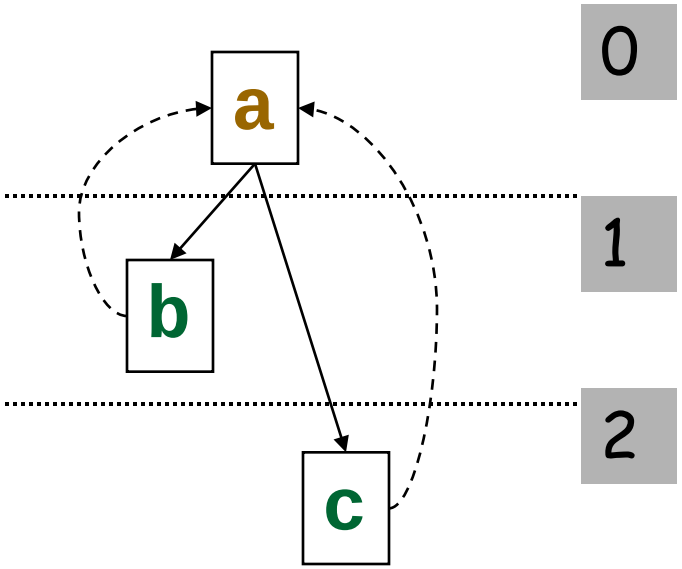
Modulo Resource Table:

0	1	
1		1

Simplest Example

```
for () {
   a = b+c
   b = a*a
   c = a*194
}
```

Try II = 2



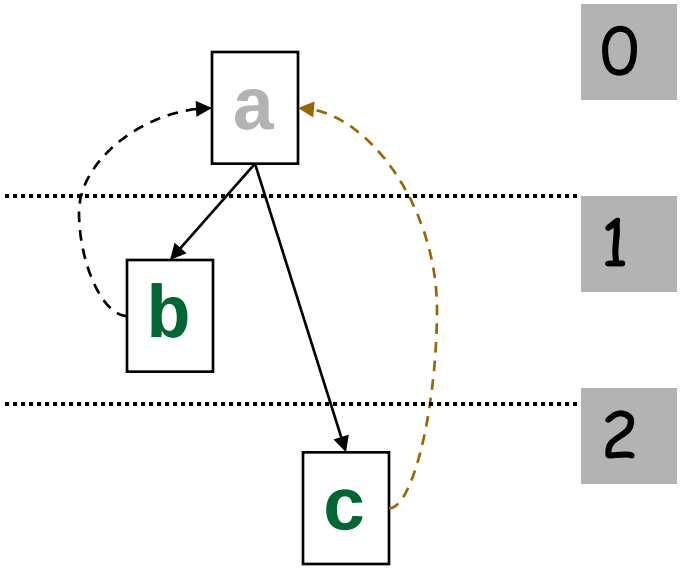
Modulo Resource Table:

0	1	1
1		1

Simplest Example

```
for () {
   a = b+c
   b = a*a
   c = a*194
}
```

Try II = 2



Modulo Resource Table:

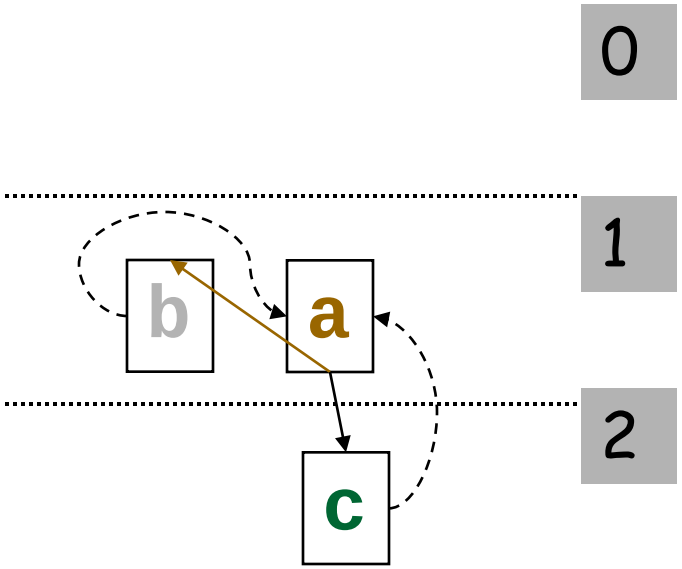
0		1
1		1

earliest a: sigma(c) + delay(c) - 2
= 2+1-2 = 1

Simplest Example

```
for () {
   a = b+c
   b = a*a
   c = a*194
}
```

Try II = 2



Modulo Resource Table:

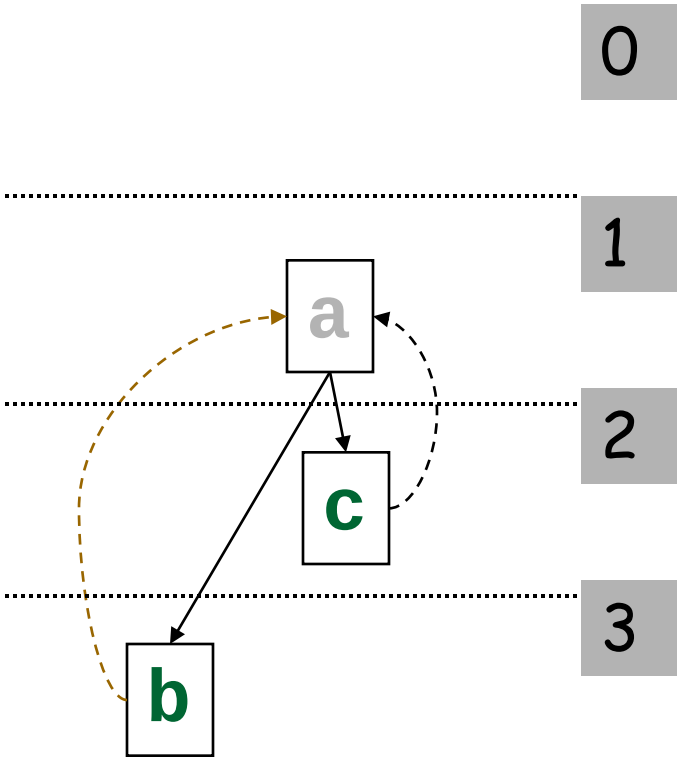
0		1
1	1	

earliest b?
scheduled b?
what next?

Simplest Example

```
for () {
   a = b+c
   b = a*a
   c = a*194
}
```

Try II = 2



Modulo Resource Table:

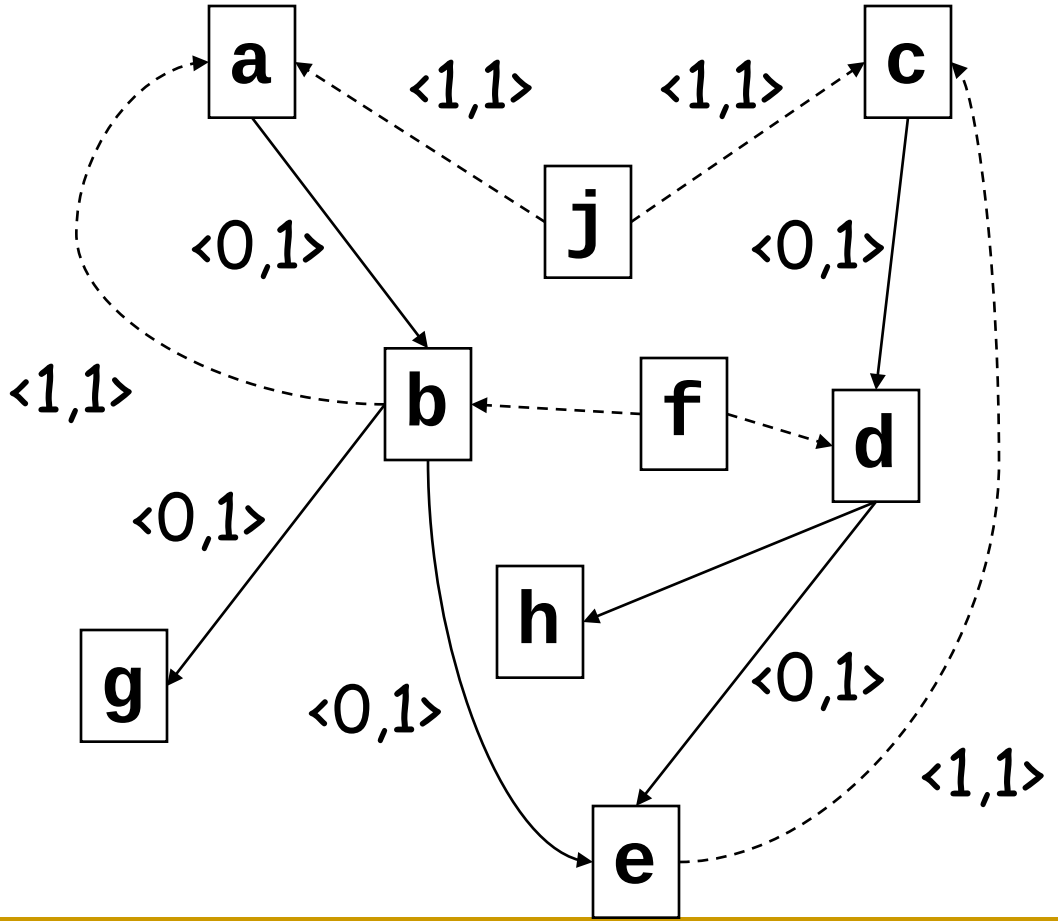
0		1
1		1

Lesson: lower bound may not be achievable

Example

```
for i:=1 to N do
    a := j ⊕ b
    b := a ⊕ f
    c := e ⊕ j
    d := f ⊕ c
    e := b ⊕ d
    f := U[i]
 g: V[i] := b
 h: W[i] := d
    j := X[i]
```

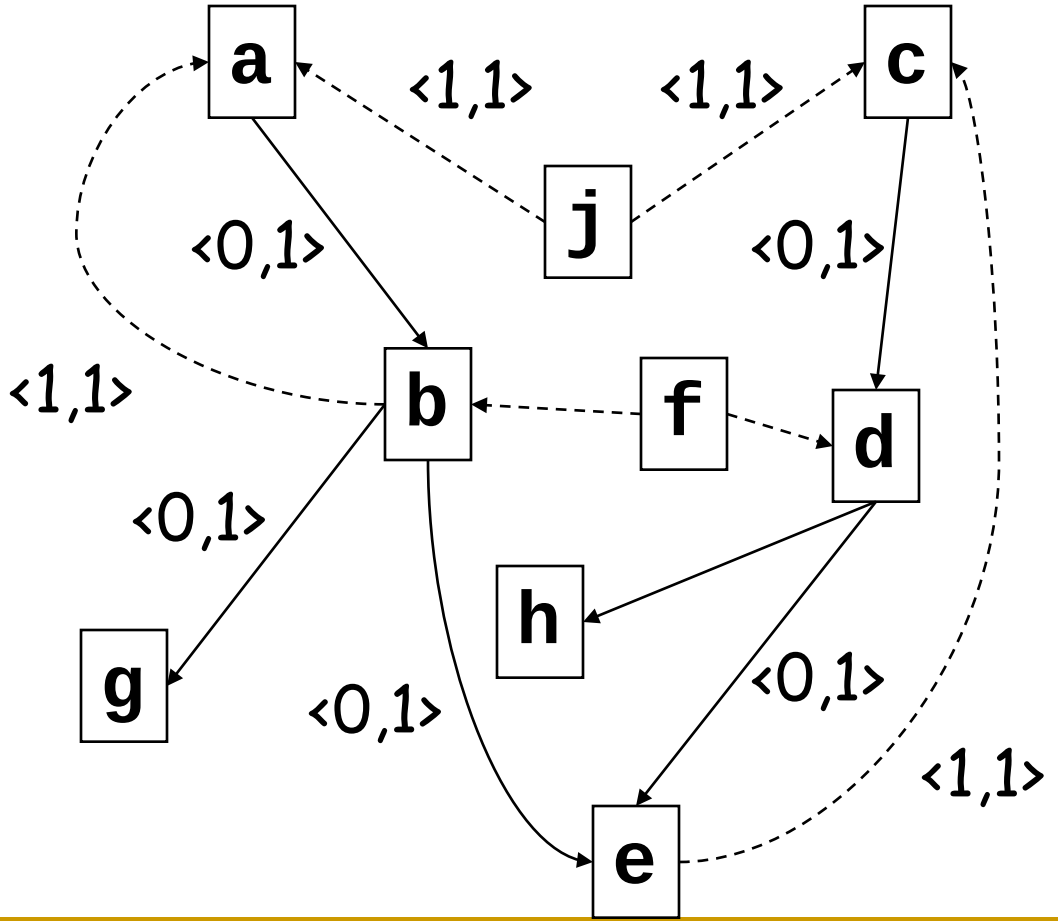
Priorities: ?



Example

```
for i:=1 to N do
    a := j ⊕ b
    b := a ⊕ f
    c := e ⊕ j
    d := f ⊕ c
    e := b ⊕ d
    f := U[i]
 g: V[i] := b
 h: W[i] := d
    j := X[i]
```

Priorities: c,d,e,a,b,f,j,g,h



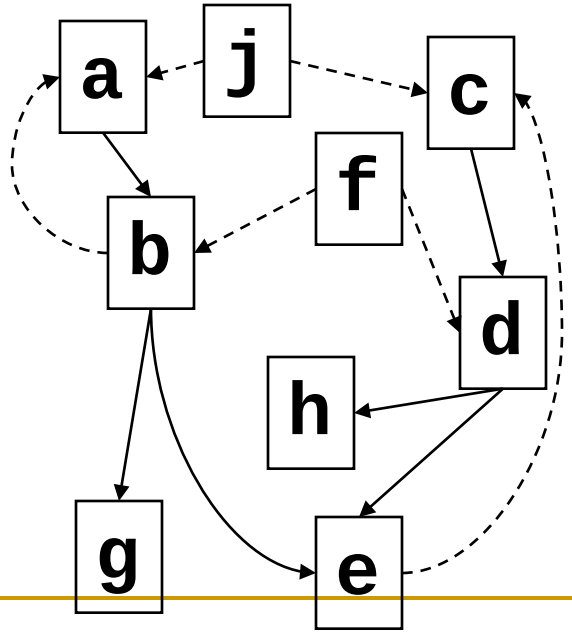
```
for i:=1 to N do
    a := j ⊕ b
    b := a ⊕ f
    c := e ⊕ j
    d := f ⊕ c
    e := b ⊕ d
    f := U[i]
g:  V[i] := b
h:  W[i] := d
    j := X[i]
```

s=5

instr	σ
a	
b	
c	
d	
e	
f	
g	
h	
j	

ALU	MU

Priorities: c,d,e,a,b,f,j,g,h



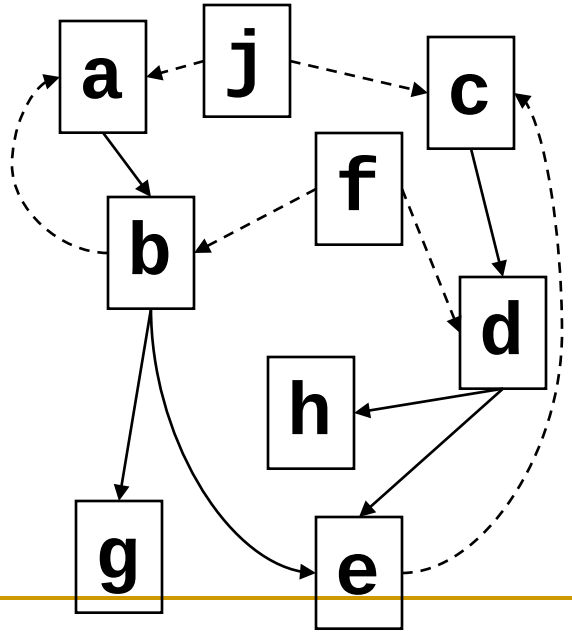
```
for i:=1 to N do
    a := j ⊕ b
    b := a ⊕ f
    c := e ⊕ j
    d := f ⊕ c
    e := b ⊕ d
    f := U[i]
g:  V[i] := b
h:  W[i] := d
    j := X[i]
```

s=5

instr	σ
a	
b	
c	0
d	1
e	2
f	
g	
h	
j	

ALU	MU
c	
d	
e	

Priorities: a,b,f,j,g,h



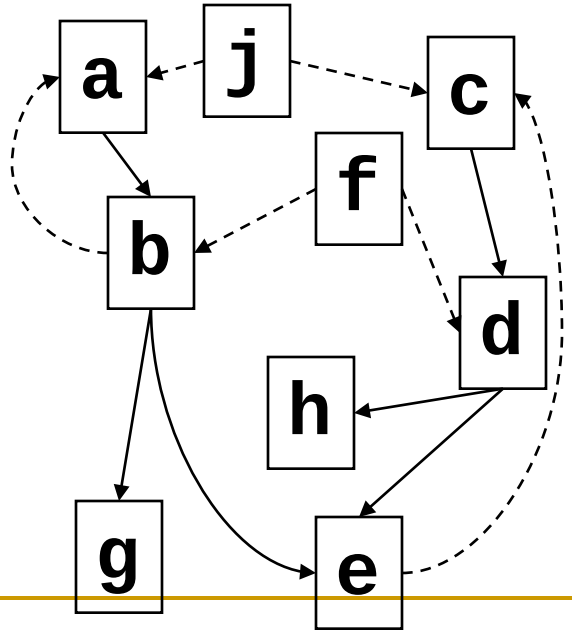
```
for i:=1 to N do
    a := j ⊕ b
    b := a ⊕ f
    c := e ⊕ j
    d := f ⊕ c
    e := b ⊕ d
    f := U[i]
g:  V[i] := b
h:  W[i] := d
    j := X[i]
```

s=5

instr	σ
a	3
b	
c	0
d	1
e	2
f	
g	
h	
j	

ALU	MU
c	
d	
e	
a	

Priorities: b,f,j,g,h



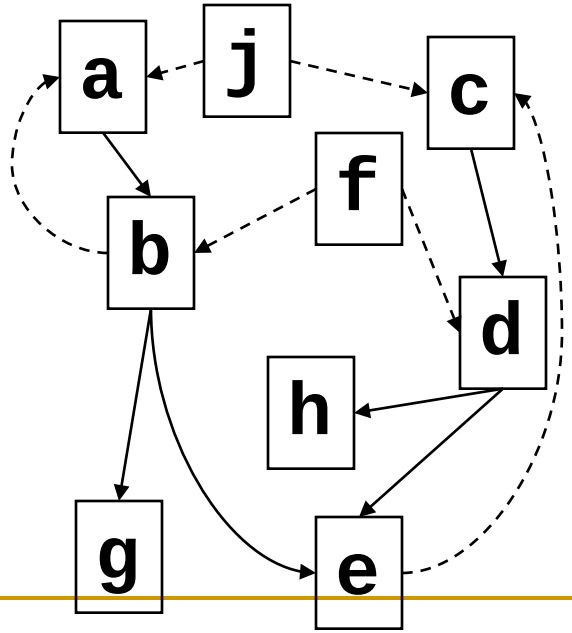
```
for i:=1 to N do
    a := j ⊕ b
    b := a ⊕ f
    c := e ⊕ j
    d := f ⊕ c
    e := b ⊕ d
    f := U[i]
g:  V[i] := b
h:  W[i] := d
    j := X[i]
```

s=5

instr	σ
a	3
b	4
c	0
d	1
e	2
f	
g	
h	
j	

ALU	MU
c	
d	
e	
a	
b	

Priorities: b,f,j,g,h



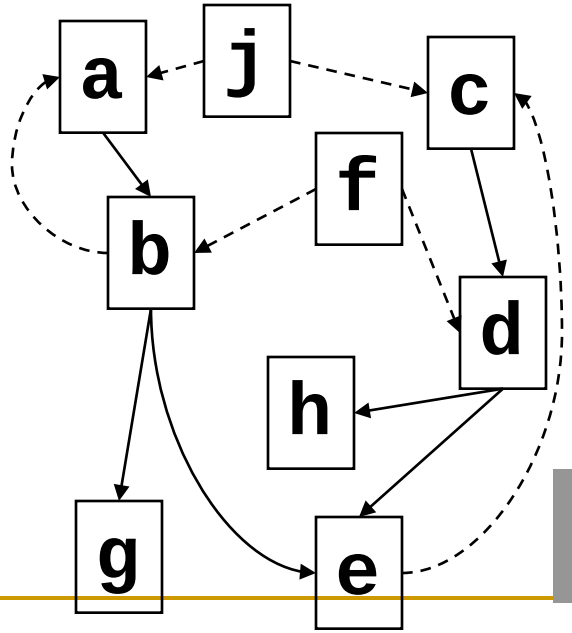
```
for i:=1 to N do
    a := j ⊕ b
    b := a ⊕ f
    c := e ⊕ j
    d := f ⊕ c
    e := b ⊕ d
    f := U[i]
g:  V[i] := b
h:  W[i] := d
    j := X[i]
```

s=5

instr	σ
a	3
b	4
c	0
d	1
e	
f	
g	
h	
j	

ALU	MU
c	
d	
a	
b	

Priorities: e,f,j,g,h



b causes b->e edge violation

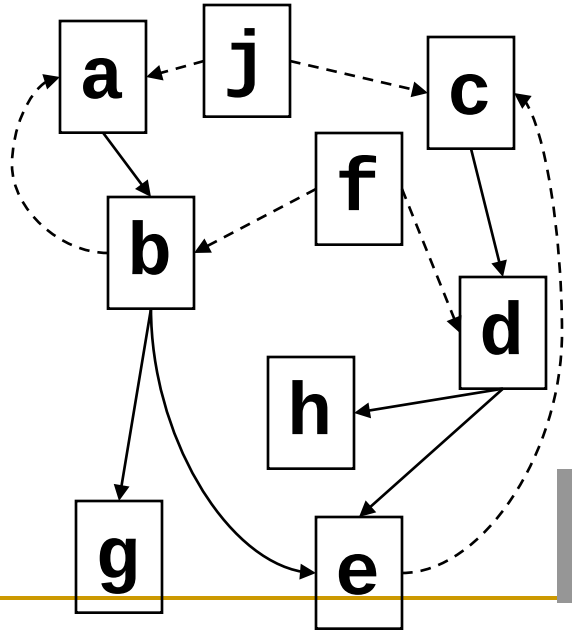
```
for i:=1 to N do
    a := j ⊕ b
    b := a ⊕ f
    c := e ⊕ j
    d := f ⊕ c
    e := b ⊕ d
    f := U[i]
g:  V[i] := b
h:  W[i] := d
    j := X[i]
```

s=5

instr	σ
a	3
b	4
c	0
d	1
e	7
f	
g	
h	
j	

ALU	MU
c	
d	
e	
a	
b	

Priorities: e,f,j,g,h



e causes e->c edge violation

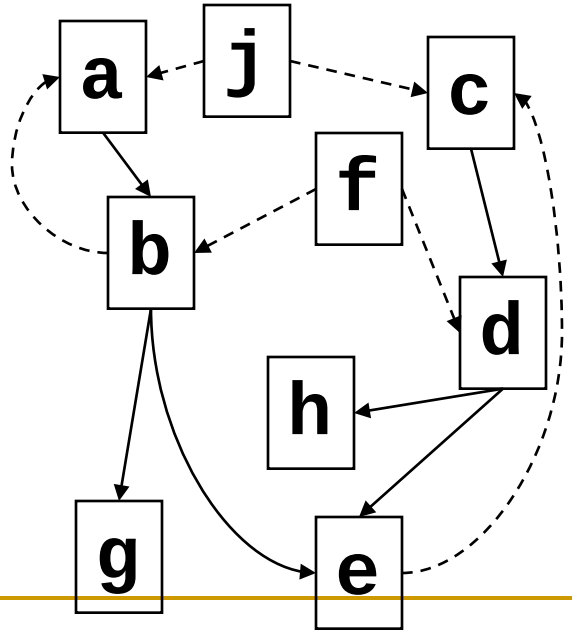
```
for i:=1 to N do
    a := j ⊕ b
    b := a ⊕ f
    c := e ⊕ j
    d := f ⊕ c
    e := b ⊕ d
    f := U[i]
g:  V[i] := b
h:  W[i] := d
    j := X[i]
```

s=5

instr	σ
a	3
b	4
c	5
d	6
e	7
f	0
g	
h	
j	

ALU	MU
c	f
d	
e	
a	
b	

Priorities: f,j,g,h



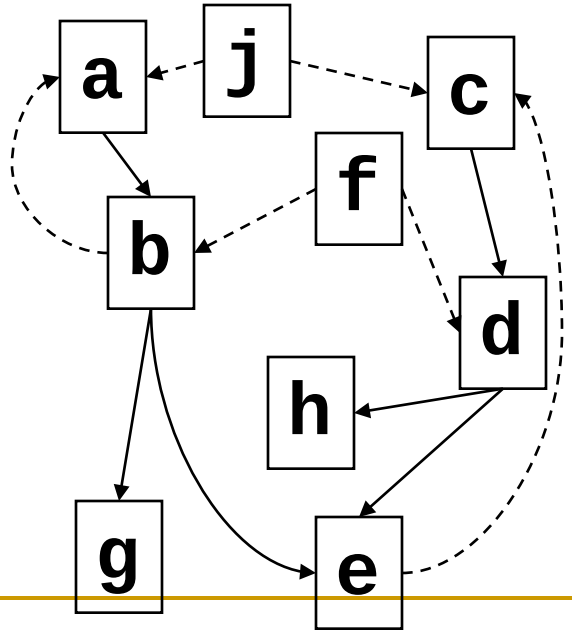
```
for i:=1 to N do
    a := j ⊕ b
    b := a ⊕ f
    c := e ⊕ j
    d := f ⊕ c
    e := b ⊕ d
    f := U[i]
g:  V[i] := b
h:  W[i] := d
    j := X[i]
```

s=5

instr	σ
a	3
b	4
c	5
d	6
e	7
f	0
g	
h	
j	1

ALU	MU
c	f
d	j
e	
a	
b	

Priorities: j, g, h



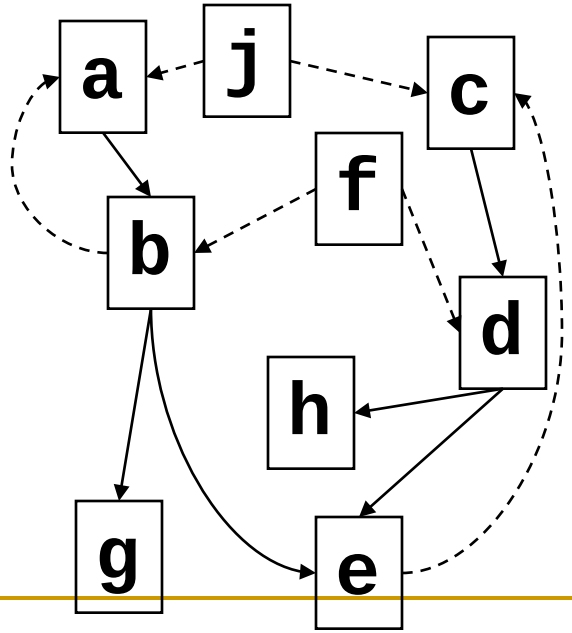
```
for i:=1 to N do
    a := j ⊕ b
    b := a ⊕ f
    c := e ⊕ j
    d := f ⊕ c
    e := b ⊕ d
    f := U[i]
g:  V[i] := b
h:  W[i] := d
    j := X[i]
```

s=5

instr	σ
a	3
b	4
c	5
d	6
e	7
f	0
g	7
h	8
j	1

ALU	MU
c	f
d	j
e	g
a	h
b	

Priorities:g,h



Creating the Loop

- Create the body from the schedule.
- Determine which iteration an instruction falls into
  - Mark its sources and dest as belonging to that iteration.
  - Add Moves to update registers
- Prolog fills in gaps at beginning
  - For each move we will have an instruction in prolog, and we fill in dependent instructions
- Epilog fills in gaps at end

instr	σ
a	3
b	4
c	5
d	6
e	7
f	0
g	7
h	8
j	1

```
f0 = U[0];
j0 = X[0];

FOR i = 0 to N
    f1 := U[i+1]
    j1 := X[i+1]
    nop
    a := j0 ? b
    b := a ? f0
    c := e ? j0
    d := f0 ? c
    e := b ? d                  g: V[i] := b
 h: W[i] := d
    f0 = f1
    j0 = j1
```

Conditionals

- What about internal control structure, I.e., conditionals
- Three approaches
  - Schedule both sides and use conditional moves
  - Schedule each side, then make the body of the conditional a macro op with appropriate resource vector
  - Trace schedule the loop

What to take away

- Architecture includes compiler!
- Dependence analysis is very important (including alias analysis)
- Software pipelining crucial for statically scheduled, but also very useful for dynamically scheduled