

Static Scheduling & VLIW

15-740

Prof. Nathan Beckmann

(Original slides by Onur Mutlu, edited by Seth Goldstein)

Carnegie Mellon University

Reprise of dynamic scheduling

- Out-of-order (OOO) processors have key benefits
 - Overlap long-latency operations by...
 - Scheduling instructions over large window (**speculation**)
 - Responding to variable latencies (**dynamism**)
- This lets OOO processors keep the backend busy and sustain high ILP
- ...But these mechanisms are complex & expensive
 - $O(N^2)$ overheads in superscalar structures
 - Complex scheduling logic & error conditions
 - E.g., detecting and rolling back speculative loads due to memory-memory RAW hazard

**DO WE REALLY NEED ALL
THIS COMPLEX HARDWARE?**

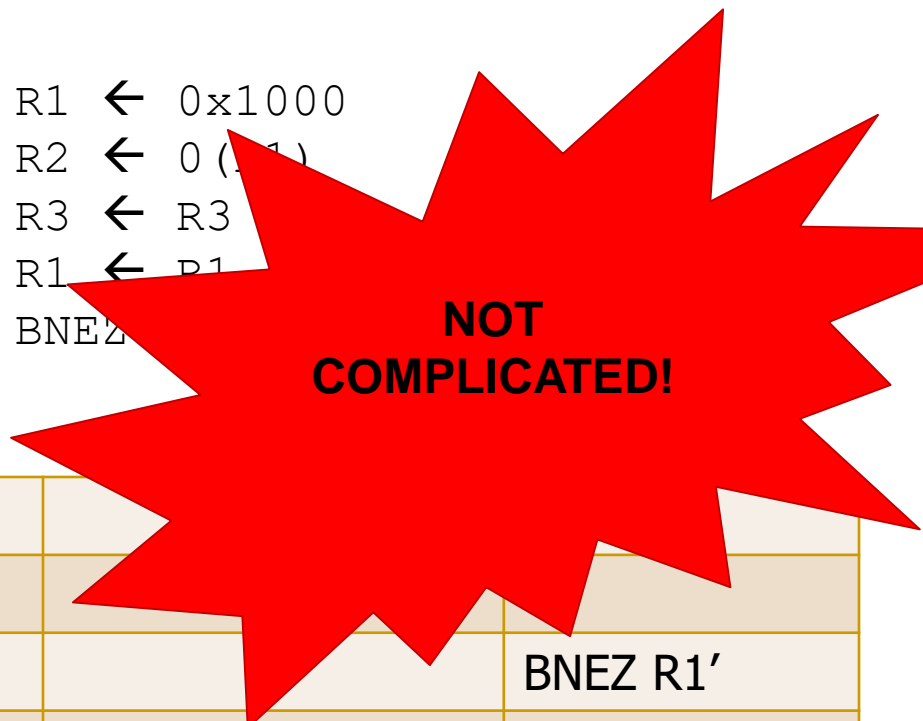
**HOW FAR CAN WE GET
WITHOUT IT?**

What would OOO do?

- 4-wide superscalar
 - LDs take 2 cycles, fully pipelined
 - Adds take 1 cycle

```

R1 ← 0x1000
LOOP: R2 ← 0 (R1)
      R3 ← R3
      R1 ← R1
      BNEZ
    
```



0	$R1 \leftarrow 1000$			
1	$R1' \leftarrow R1 - 4$	$R2 \leftarrow (R1)$		
2	$R1'' \leftarrow R1' - 4$	$R2' \leftarrow (R1')$		BNEZ R1'
3	$R1''' \leftarrow R1'' - 4$	$R2'' \leftarrow (R1'')$	$R3 \leftarrow R3 + R2$	BNEZ R1''
4	$R1'''' \leftarrow R1''' - 4$	$R2''' \leftarrow (R1''')$	$R3' \leftarrow R3' + R2'$	BNEZ R1'''
5	$R1''''' \leftarrow R1'''' - 4$	$R2'''' \leftarrow (R1''''')$	$R3'' \leftarrow R3'' + R2''$	BNEZ R1''''
6	$R1'''''' \leftarrow R1''''' - 4$	$R2''''' \leftarrow (R1''''')$	$R3''' \leftarrow R3''' + R2'''$	BNEZ R1'''''
...	$R1''''''' \leftarrow R1'''''' - 4$	$R2'''''' \leftarrow (R1''''''')$	$R3'''' \leftarrow R3'''' + R2''''$	BNEZ R1''''''

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

Very long instruction word - VLIW

- Compiler does the scheduling statically
- 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.)

VLIW example – 2-cycle loads

■ RISC code

```
MUL    R1, R3, 3
LD     R4, 0(R1)
ADD    R2, R2, R4
SUB    R3, R3, 1
BNEZ   R3, -4
```

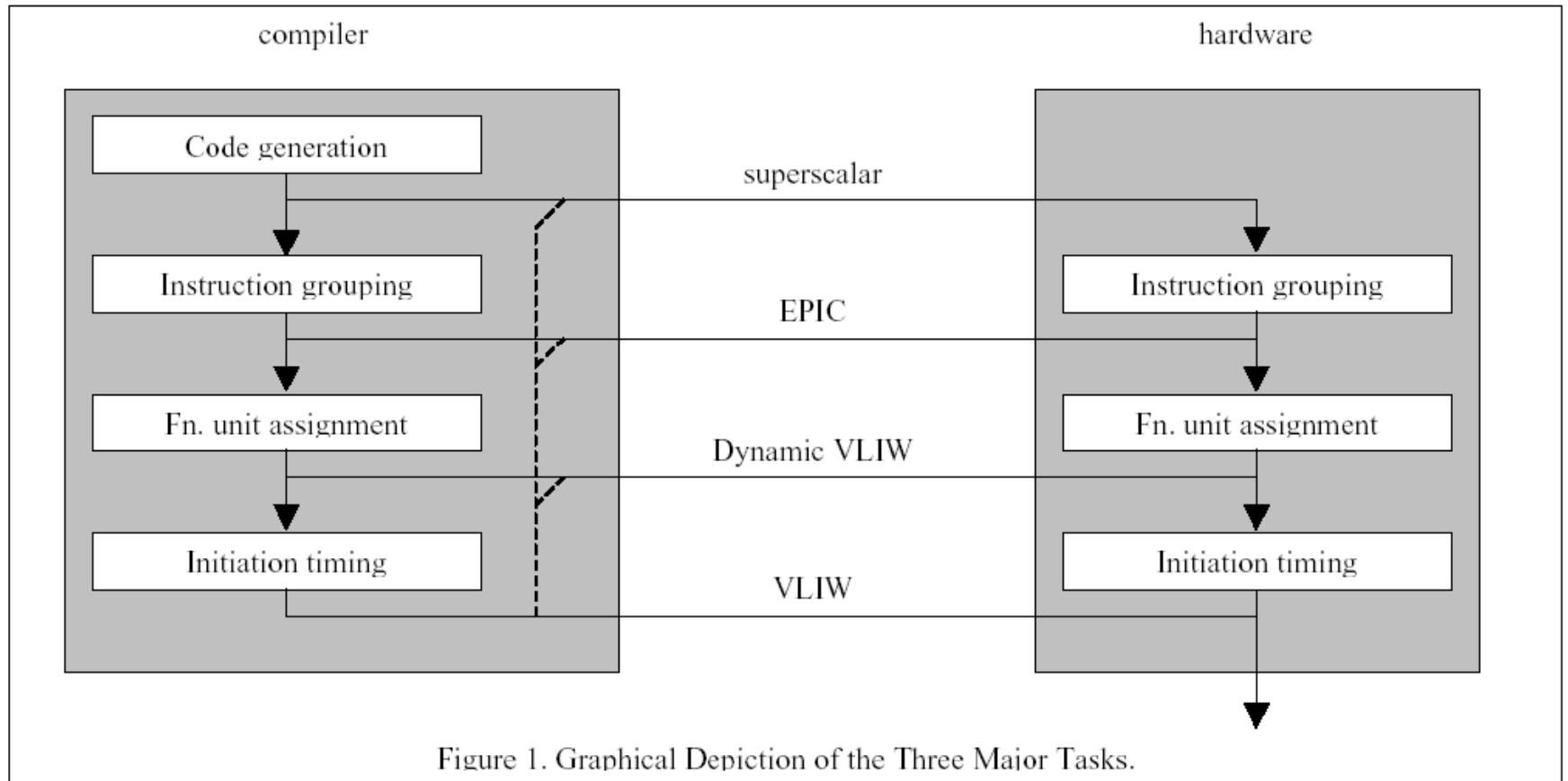
■ VLIW code

```
MUL R1, R3, 3      SUB R3, R3, 1
LD  R4, 0(R1)     NOP
NOP               NOP
ADD R2, R2, R4    BNEZ R3, -4
```


VLIW relies on compiler for ILP

- Idea: expose microarchitecture to compiler
 - Available functional units & their latencies
- Compiler finds a good static schedule of independent instructions
 - All dependencies checked statically
- Hardware blindly executes instructions in parallel
 - No $O(N^2)$ checks
- Limitations: Can the compiler really do this? What about control hazards, memory dependencies?

Comparison between SS ↔ VLIW



From Mark Smotherman, "[Understanding EPIC Architectures and Implementations](#)"

Comparison: CISC, RISC, VLIW

ARCHITECTURE
CHARACTERISTIC

CISC

RISC

VLIW

VLIW is a natural extension of RISC ideas to superscalar

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

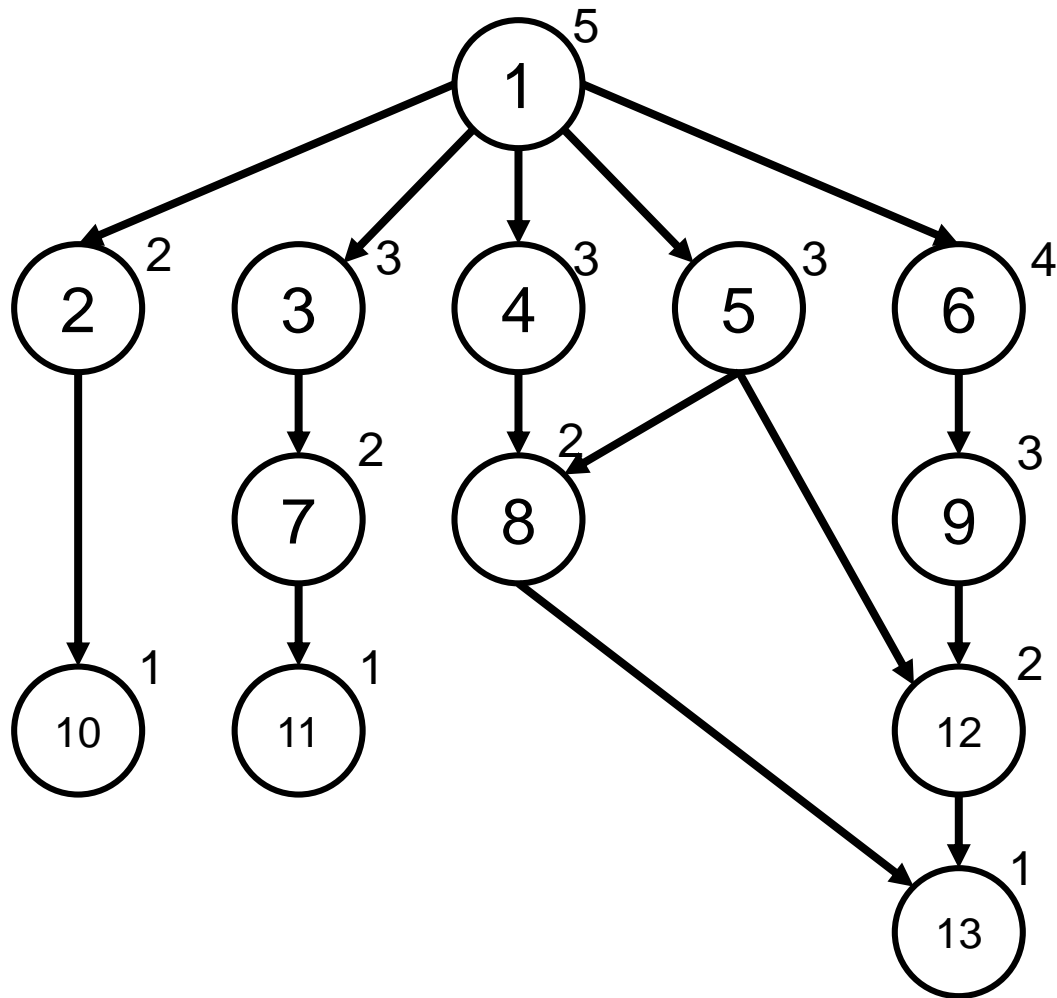
List Scheduling: For 1 basic block

- Idea: Assign **priority** to each instruction
- Initialize ready list that holds all ready instructions
- Choose one ready instruction ***I*** from ready list with the highest priority
- Insert ***I*** into schedule
 - Ensuring resources are available (structural hazards)
- Add those instructions whose precedence constraints are now satisfied into the ready list

Instruction Prioritization Heuristics

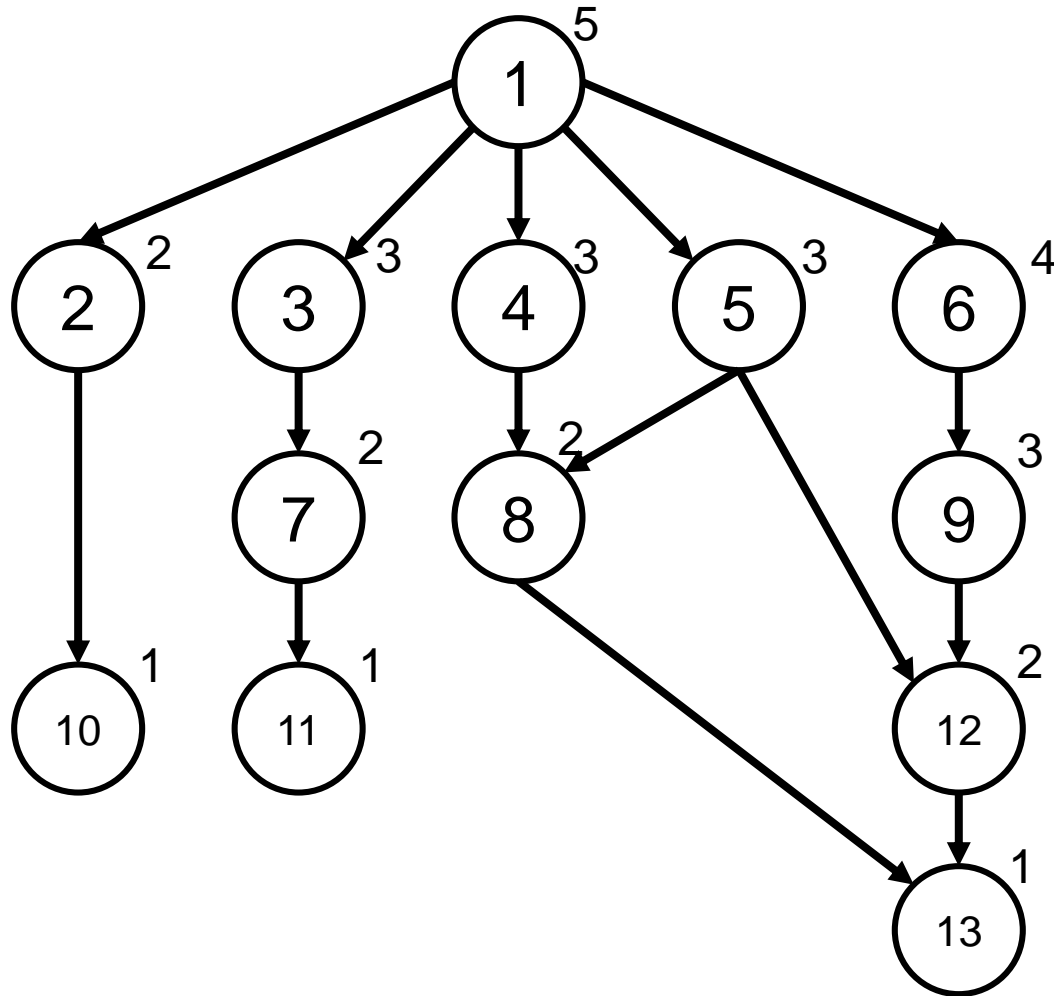
- Number of descendants in precedence graph
- Maximum latency from root node of precedence graph
- Length of operation latency
- Ranking of paths based on importance
- Some combination of above

VLIW List Scheduling



4-wide VLIW

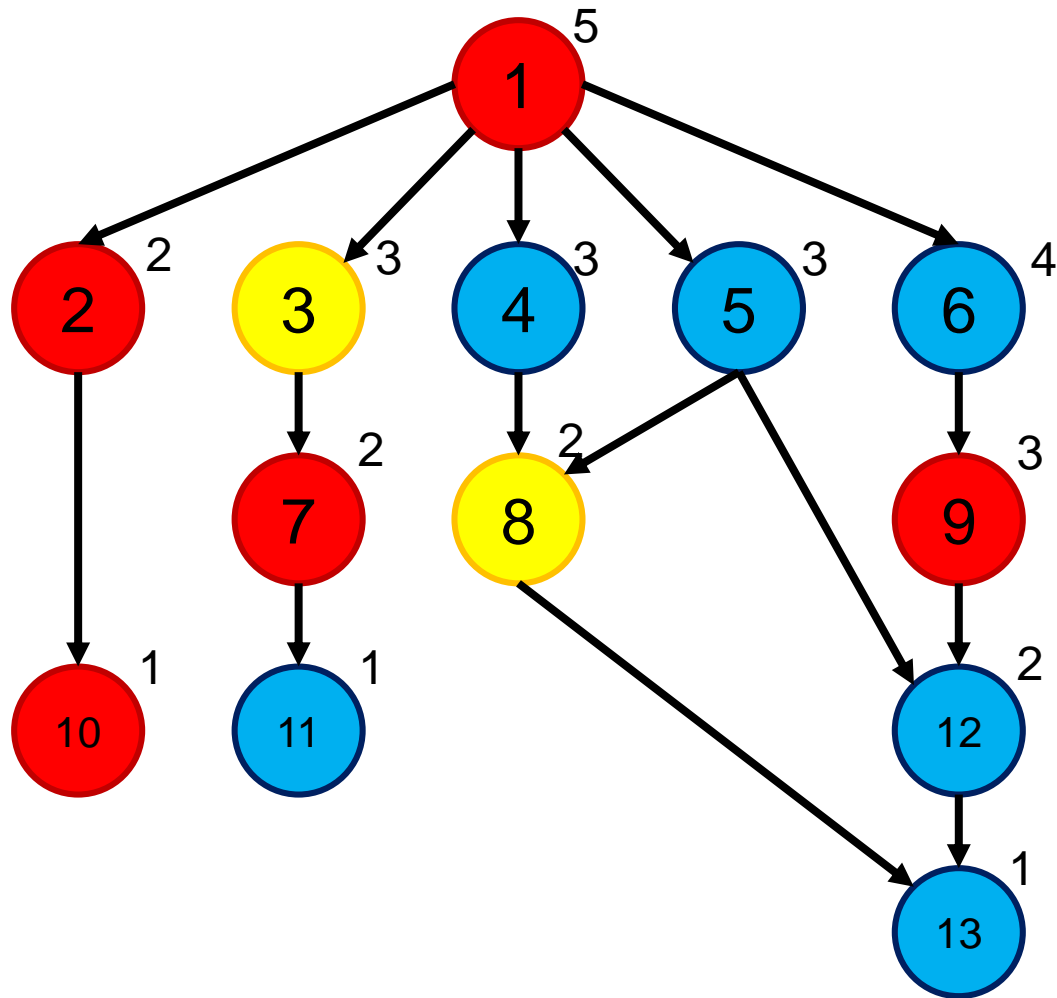
VLIW List Scheduling



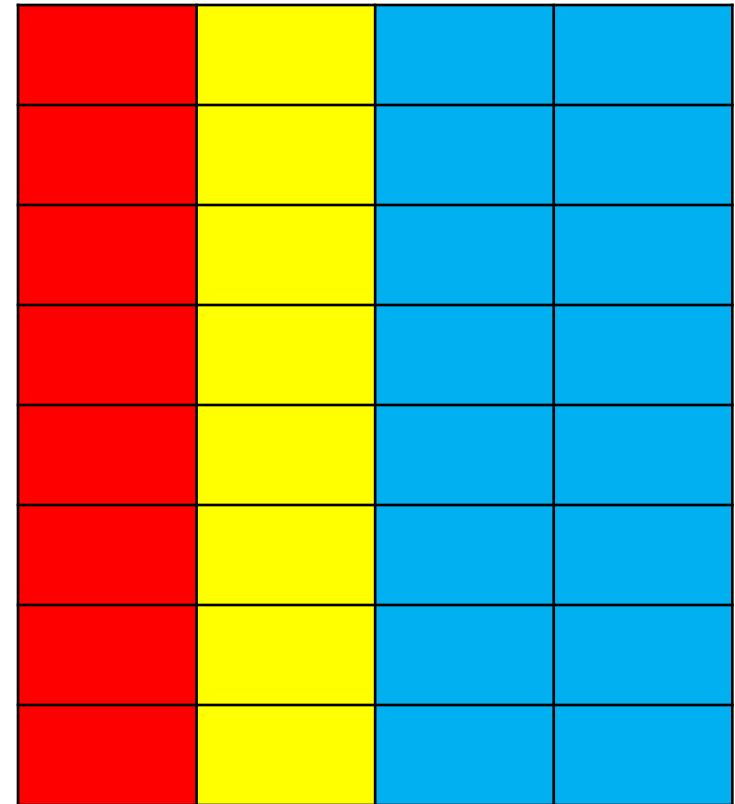
4-wide VLIW

1			
6	3	4	5
9	2	7	8
12	10	11	
13			

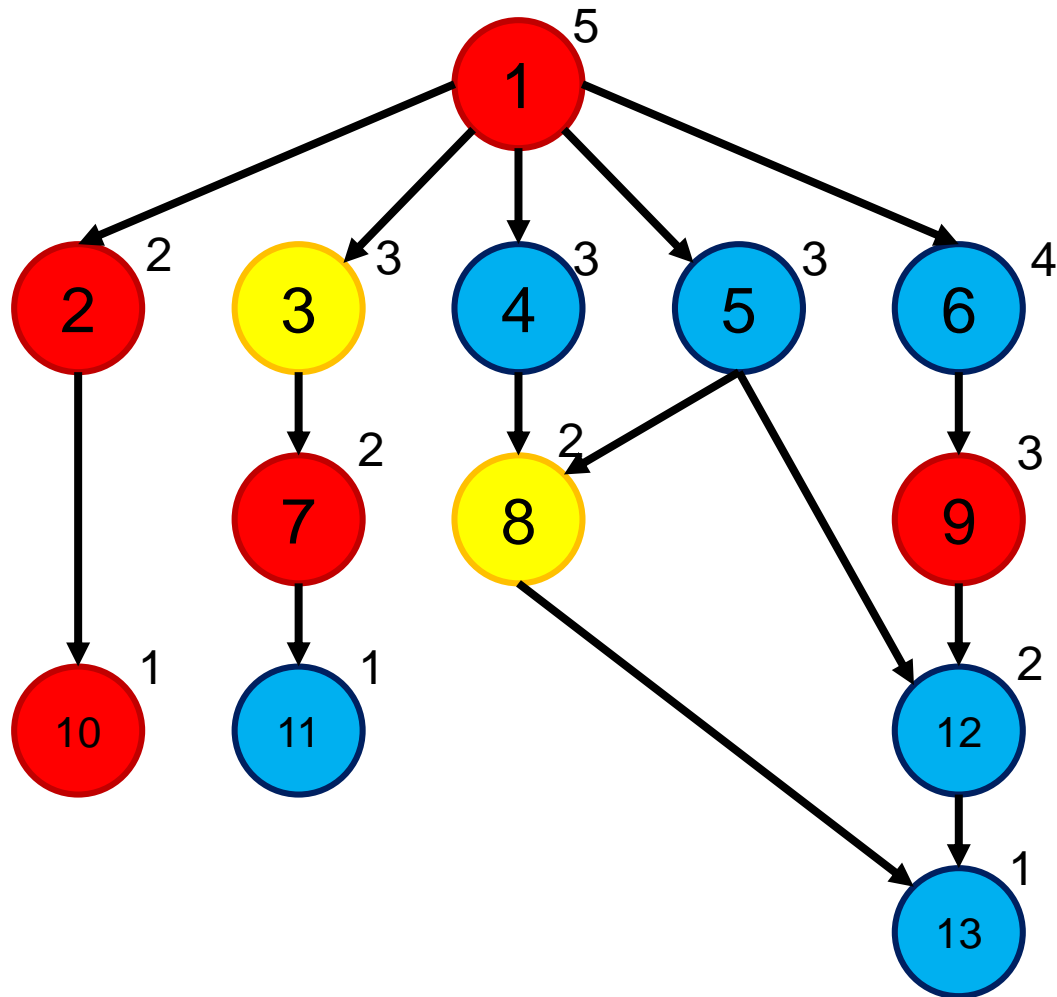
VLIW List Scheduling+Structural Hazards



4-wide VLIW



VLIW List Scheduling+Structural Hazards



4-wide VLIW

1			
2	3	6	5
9		4	
7	8	12	
10		11	13

WHAT ABOUT LOOPS?

The problem with loops

- Consider the following code:

```
for (int i = 0; i < N; i++) {  
    b[i] = b[i] * b[i];  
}
```

The problem with loops

- Consider the following code:

```
for (int i = 0; i < N; i++) {  
    b[i] = b[i] * b[i];  
}
```

- RISC assembly (LD & MUL 2-cycles)

LOOP:

```
LD R1, 0(R3)  
MUL R2, R1, R1  
ST R2, 0(R3)
```

```
ADD R3, R3, 4  
BLT R3, R4, LOOP
```

Useful work

→ 7 cycles
per iteration

Loop overhead

The problem with loops

- Consider the following code:

```
for (int i = 0; i < N; i++) {  
    b[i] = b[i] * b[i];  
}
```

- VLIW assembly (1 ALU, 1 LD/ST; 2-cycles)

LOOP:	NOP	LD R1, 0(R3)
	NOP	NOP
	MUL R2, R1, R1	NOP
	ADD R, 3 R, 3 4	NOP
	BLT R3, R4, LOOP	ST R2, -4(R3)

Loop overhead

Useful work

**→ 5 cycles
per iteration**

Amortize overheads by **unrolling** loops

- Key idea is to schedule the following code instead:

```
for (int i = 0; i < N; i+=4) {  
    b[i+0] = b[i+0] * b[i+0];  
    b[i+1] = b[i+1] * b[i+1];  
    b[i+2] = b[i+2] * b[i+2];  
    b[i+3] = b[i+3] * b[i+3];  
}
```

Loop unrolling
→ Larger scheduling block
→ Better schedule

Amortize overheads by **unrolling** loops

- VLIW assembly

LOOP:

NOP

NOP

MUL R2, R1, R1

MUL R4, R3, R3

MUL R6, R5, R5

MUL R8, R7, R7

ADD R9, R9, 16

BLT R9, R10 LOOP

LD R1, 0 (R9)

LD R3, 4 (R9)

LD R5, 8 (R9)

LD R7, 12 (R9)

ST R2, 0 (R9)

ST R4, 4 (R9)

ST R6, 8 (R9)

ST R8, -4 (R9)

→ **2 cycles**
per iteration

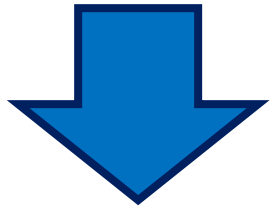
Loop overhead

Useful work

Correctness of loop unrolling

■ Is this transformation legal?

```
for (int i = 0; i < N; i++) {  
    b[i] = b[i] * b[i];  
}
```



```
for (int i = 0; i < N; i+=4) {  
    b[i+0] = b[i+0] * b[i+0];  
    b[i+1] = b[i+1] * b[i+1];  
    b[i+2] = b[i+2] * b[i+2];  
    b[i+3] = b[i+3] * b[i+3];  
}
```

Correctness of loop unrolling

- Instead, schedule the following code:

```
int i;
for (i = 0; i+3 < N; i+=4) {
    b[i+0] = b[i+0] * b[i+0];
    b[i+1] = b[i+1] * b[i+1];
    b[i+2] = b[i+2] * b[i+2];
    b[i+3] = b[i+3] * b[i+3];
}
for (; i < N; i++) {
    b[i] = b[i] * b[i];
}
```

Loop unrolling summary

- Advantages

- Reduces loop overhead
- Improves code schedule within loop
 - (eg, hiding MUL latency in example)

- Disadvantages

- Increases code size
- Less effective on loops with internal branches
 - Can use predication ... more on this later

Can we do better?

- VLIW assembly (ALU + LD/ST)

LOOP:	NOP	LD R1, 0 (R9)
	NOP	LD R3, 4 (R9)
	MUL R2, R1, R1	LD R5, 8 (R9)
	MUL R4, R3, R3	LD R7, 12 (R9)
	MUL R6, R5, R5	ST R2, 0 (R9)
	MUL R8, R7, R7	ST R4, 4 (R9)
	ADD R9, R9, 16	ST R6, 8 (R9)
	BLT R9, R10 LOOP	ST R8, -4 (R9)
	Loop overhead	Useful work

→ 2 cycles per iteration

Not in this case – LD/ST unit is at 100% utilization

Can we do better?

- VLIW assembly (3-wide)

LOOP: NOP	LD R1, 0 (R9)	NOP
NOP	LD R3, 4 (R9)	NOP
MUL R2, R1, R1	LD R5, 8 (R9)	NOP
MUL R4, R3, R3	LD R7, 12 (R9)	ST R2, 0 (R9)
MUL R6, R5, R5	NOP	ST R4, 4 (R9)
MUL R8, R7, R7	ADD R9, R9, 16	ST R6, 8 (R9)
NOP	BLT R9, R10, LOOP	ST R8, 12 (R9)

Loop overhead **Useful work**

➔ **7/4 cycles
per iteration**

Can we do better?

■ VLIW assembly (3-wide)

```
LOOP:  NOP
      NOP
      MUL R2, R1, R1
      MUL R4, R3, R3
      MUL R6, R3, R3
      MUL R8, R3, R3
      MUL R10, R3, R3
      MUL R12, R3, R3
      MUL R14, R5, R5
      MUL R16, R7, R7
      NOP
      LD R1, 0 (R31)
      LD R3, 4 (R31)
      LD R5, 8 (R31)
      LD R7, 12 (R31)
      LD R9, 12 (R31)
      LD R11, 12 (R31)
      LD R13, 12 (R31)
      LD R15, 12 (R31)
      NOP
      ADD R31, R31, 32
      BLT R31, R10, LOOP
      NOP
      NOP
      NOP
      ST R2, 0 (R31)
      ST R4, 4 (R31)
      ST R6, 8 (R31)
      ST R8, 12 (R31)
      ST R10, 16 (R31)
      ST R12, 20 (R31)
      ST R14, 24 (R31)
      ST R16, -4 (R31)
```

Loop overhead

Useful work

**→ 11/8 cycles
per iteration**

**...but code &
register bloat**

Can we do better than unrolling?

■ VLIW assembly (3-wide)

Ramp-up &
ramp-down
overhead

LOOP:	NOP	LD R1, 0 (R31)	NOP
	NOP	LD R3, 4 (R31)	NOP
	MUL R2, R1, R1	LD R5, 8 (R31)	NOP
	MUL R4, R3, R3	LD R7, 12 (R31)	ST R2, 0 (R31)
	MUL R6, R3, R3	LD R9, 12 (R31)	ST R4, 4 (R31)
	MUL R8, R3, R3	LD R11, 12 (R31)	ST R6, 8 (R31)
	MUL R10, R3, R3	LD R13, 12 (R31)	ST R8, 12 (R31)
	MUL R12, R3, R3	LD R15, 12 (R31)	ST R10, 16 (R31)
	MUL R14, R5, R5	NOP	ST R12, 20 (R31)
	MUL R16, R7, R7	ADD R31, R31, 32	ST R14, 24 (R31)
	NOP	BLT R31, R10, LOOP	ST R16, -4 (R31)

Perfect efficiency

Loop overhead

Useful work

Goal: Maintain peak efficiency, w/out ramp-up/down

Software Pipelining

- Idea: Move instructions **across iterations** of the loop
 - Very large improvements in running time are possible

- E.g., 5-wide VLIW

```
LD R1, 0(R9)  NOP                ADD R9, R9, 4
NOP           NOP                NOP
LD R1, 4(R9)  MUL R2, R1, R1      ADD R9, R9, 4
NOP           SUB R10, R10, 4     BGE R9, R10, END

LOOP:        { LD R1, 0(R9)        ← Current iteration
              MUL R2, R1, R1      ← One iteration ago
              ST R2, -8(R9)       ← Two iterations ago
              ADD R9, R9, 4
              BLT R9, R10, LOOP   }

END:        NOP                NOP                NOP
NOP         NOP                MUL R2, R1, R1      ST R2, -4(R9)
NOP         NOP                NOP                NOP
NOP         NOP                NOP                ST R2, 0(R9)
```

Software Pipelining

- Idea: Move instructions **across iterations** of the loop
 - Very large improvements in running time are possible

- E.g., 5-wide VLIW

```
LD R1, 0(R9)  NOP                                ADD R9, R9, 4
NOP           NOP                                NOP
LD R1, 4(R9)  MUL R2, R1, R1                      ADD R9, R9, 4
NOP           SUB R10, R10, 4                      BGE R9, R10, END
```

LOOP:

```
{ LD R1, 0(R9)
  MUL R2, R1, R1
  ST R2, -8(R9)
  ADD R9, R9, 4
  BLT R9, R10, LOOP }
```

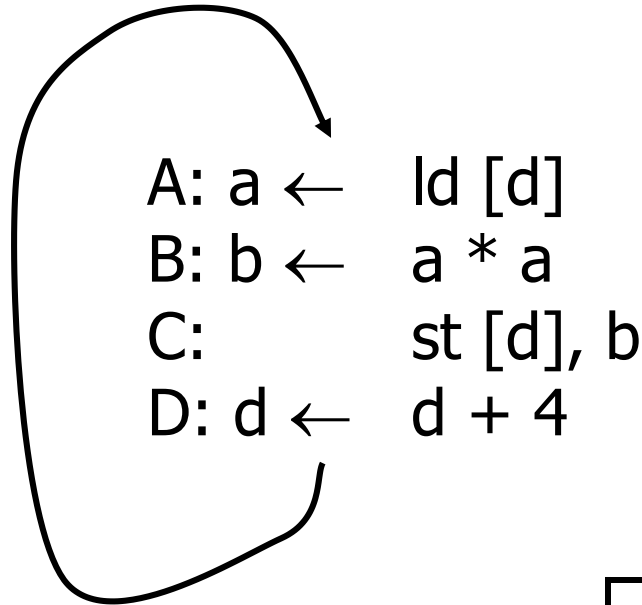
**→ 1 cycle
per iteration**

```
END:  NOP           NOP           NOP
      NOP           MUL R2, R1, R1  ST R2, -4(R9)
      NOP           NOP           NOP
      NOP           NOP           ST R2, 0(R9)
```

Perfect
efficiency

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: $a \leftarrow \text{ld } [d]$

B: $b \leftarrow a * a$

C: $\text{st } [d], b$

D: $d \leftarrow d + 4$

A1: $a \leftarrow \text{ld } [d]$

B1: $b \leftarrow a * a$

C1: $\text{st } [d], b$

D1: $d \leftarrow d + 4$



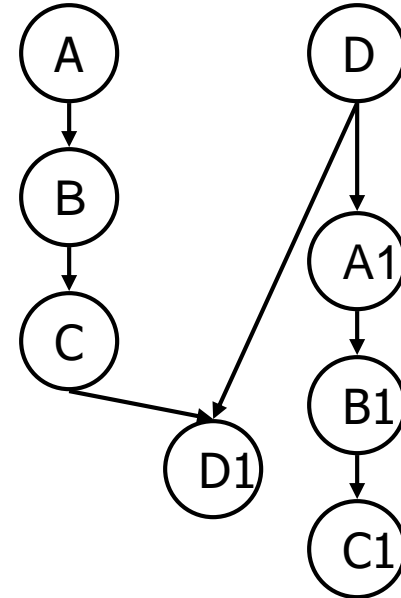
Rename variables

A: $a \leftarrow \text{ld } [d]$
B: $b \leftarrow a * a$
C: $\text{st } [d], b$
D: $d1 \leftarrow d + 4$
A1: $a1 \leftarrow \text{ld } [d1]$
B1: $b1 \leftarrow a1 * a1$
C1: $\text{st } [d1], b1$
D1: $d \leftarrow d1 + 4$



Schedule

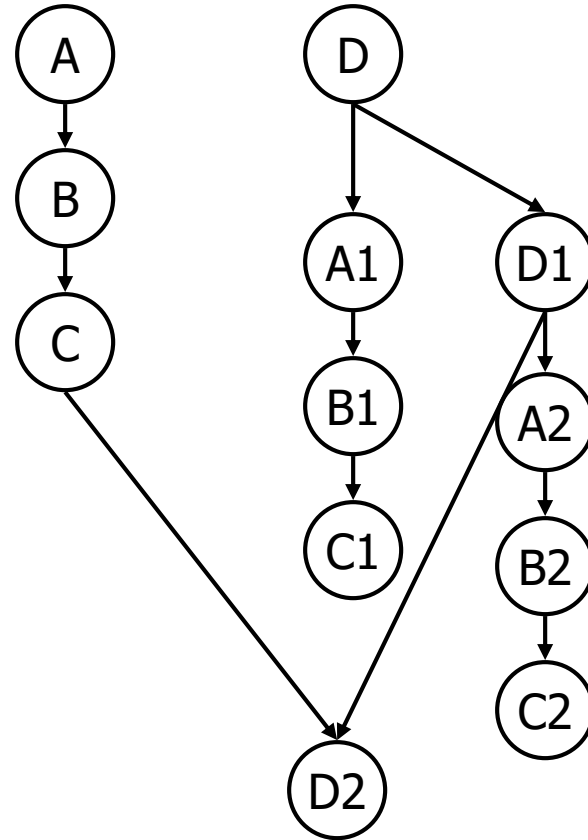
A: $a \leftarrow \text{ld}[d]$
B: $b \leftarrow a * a$
C: $\text{st}[d], b$
D: $d1 \leftarrow d + 4$
A1: $a1 \leftarrow \text{ld}[d1]$
B1: $b1 \leftarrow a1 * a1$
C1: $\text{st}[d1], b1$
D1: $d \leftarrow d1 + 4$



A		B		C		D1	
D		A1		B1		C1	

Unroll Some More

A: $a \leftarrow \text{ld}[d]$
B: $b \leftarrow a * a$
C: $\text{st}[d], b$
D: $d1 \leftarrow d + 4$
A1: $a1 \leftarrow \text{ld}[d1]$
B1: $b1 \leftarrow a1 * a1$
C1: $\text{st}[d1], b1$
D1: $d2 \leftarrow d1 + 4$
A2: $a2 \leftarrow \text{ld}[d2]$
B2: $b2 \leftarrow a2 * a2$
C2: $\text{st}[d2], b2$
D2: $d \leftarrow d2 + 4$

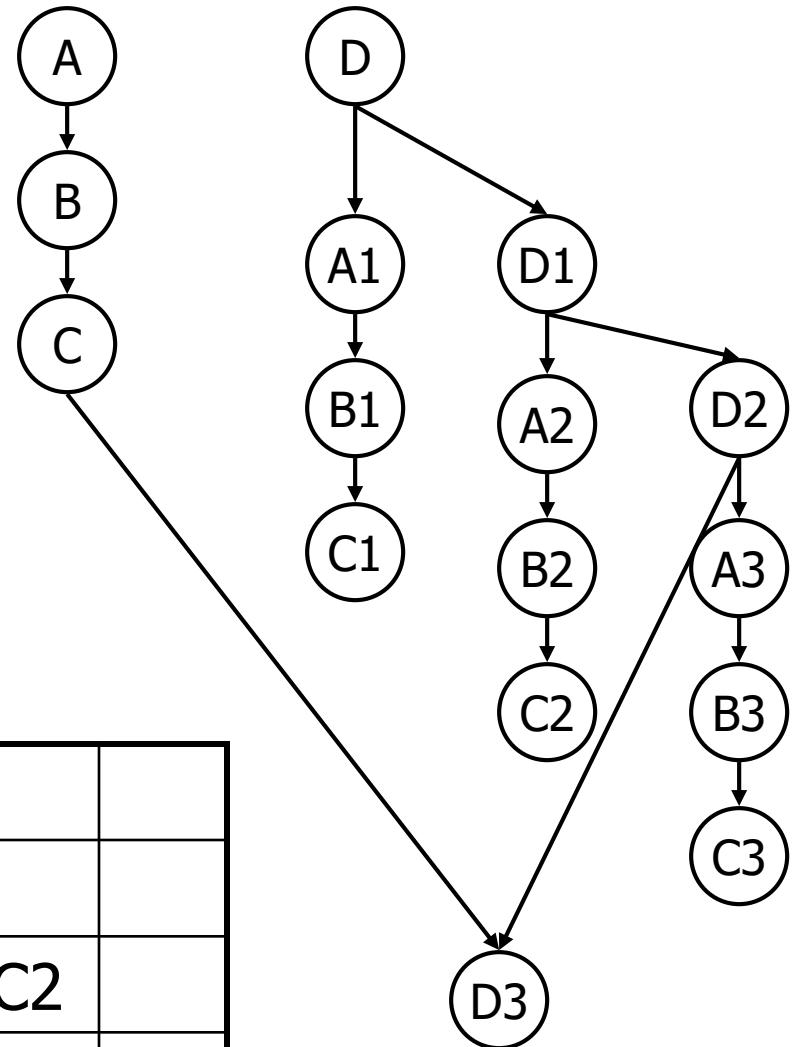


A		B		C		D2	
D		A1		B1		C1	
	D1		A2		B2		C2

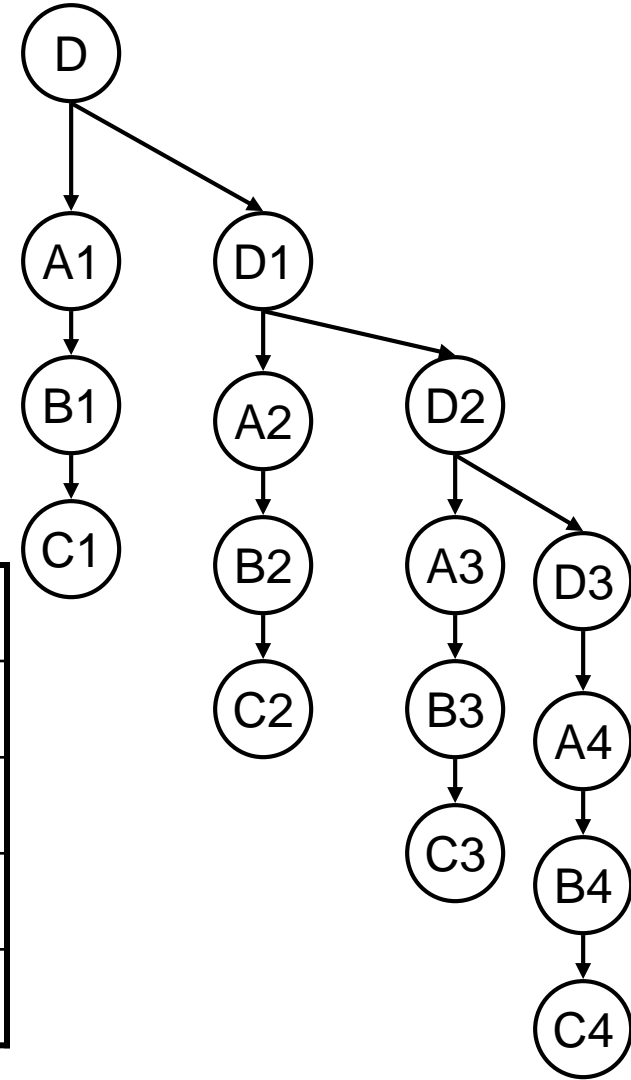
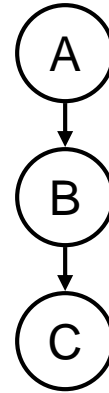
Unroll Some More

A: $a \leftarrow \text{ld}[d]$
 B: $b \leftarrow a * a$
 C: $\text{st}[d], b$
 D: $d1 \leftarrow d + 4$
 A1: $a1 \leftarrow \text{ld}[d1]$
 B1: $b1 \leftarrow a1 * a1$
 C1: $\text{st}[d1], b1$
 D1: $d2 \leftarrow d1 + 4$
 A2: $a2 \leftarrow \text{ld}[d2]$
 B2: $b2 \leftarrow a2 * a2$
 C2: $\text{st}[d2], b2$
 D2: $d \leftarrow d2 + 4$

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

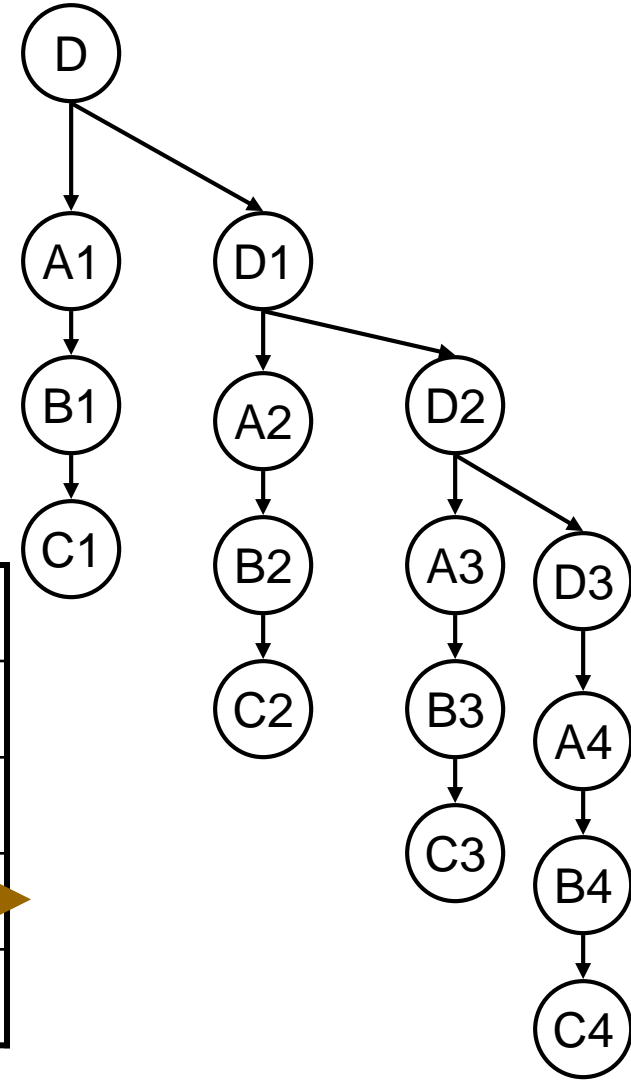
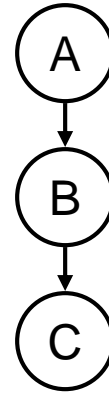


One More Time



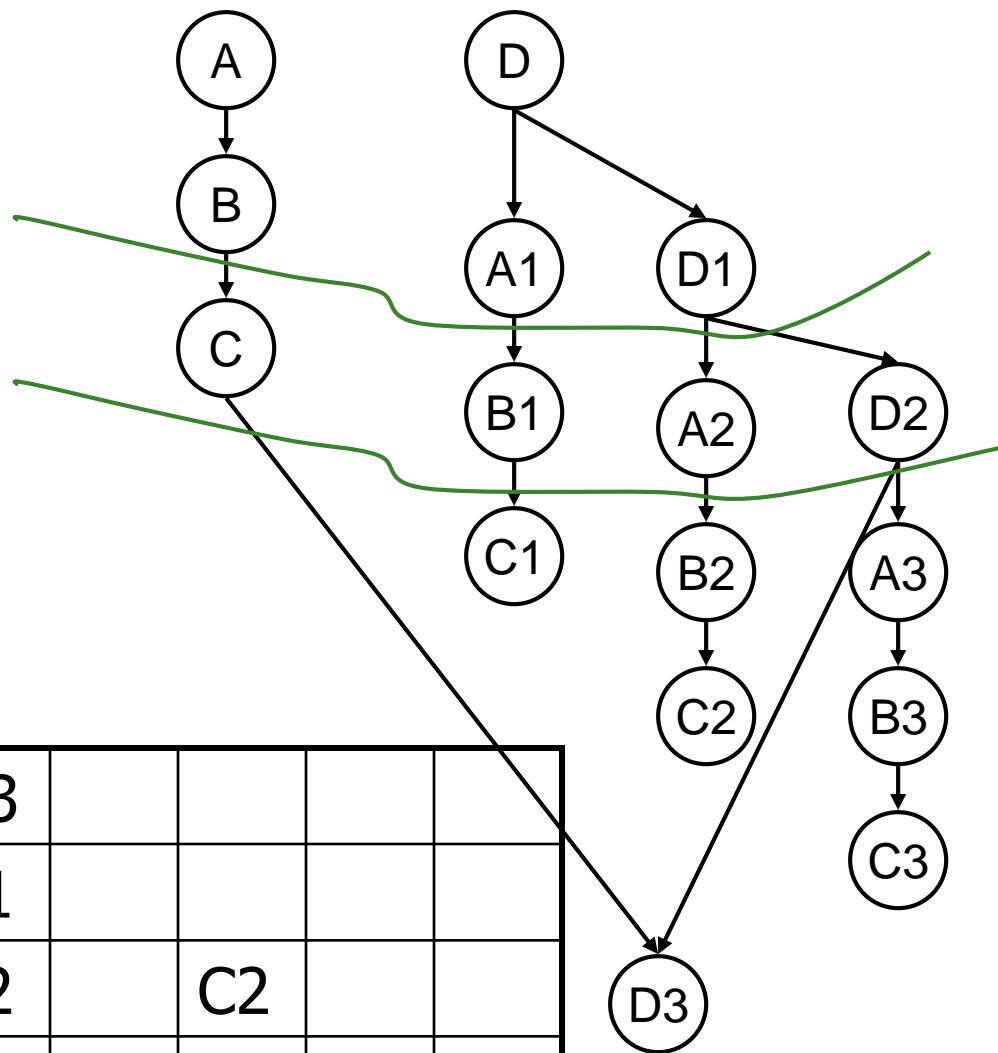
A		B		C					
D		A1		B1		C1			
	D1		A2		B2		C2		
		D2		A3		B3		C3	
			D3		A4		B4		C4

Can Rearrange

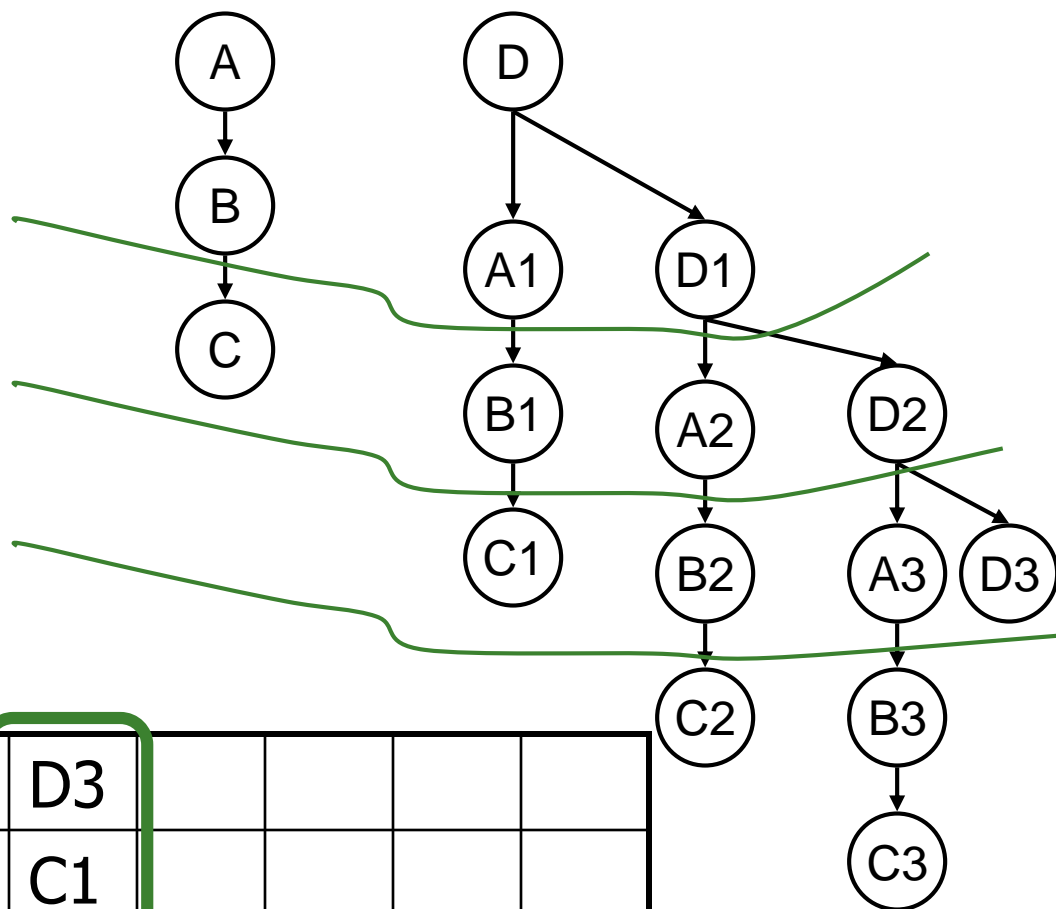


A		B		C					
D		A1		B1		C1			
	D1	→	A2	→	B2	→	C2	→	
		D2	→	A3	→	B3	→	C3	→
			D3		A4		B4		C4

Rearrange



Rearrange



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

SP Loop

A: $a \leftarrow \text{ld}[d]$

B: $b \leftarrow a * a$

D: $d1 \leftarrow d + 4$

A1: $a1 \leftarrow \text{ld}[d1]$

D1: $d2 \leftarrow d1 + 4$

Prolog

C: $\text{st}[d], b$

B1: $b1 \leftarrow a1 * a1$

A2: $a2 \leftarrow \text{ld}[d2]$

D2: $d \leftarrow d2 + 4$

Body

B2: $b2 \leftarrow a2 * a2$

C1: $\text{st}[d1], b1$

D3: $d2 \leftarrow d1 + 4$

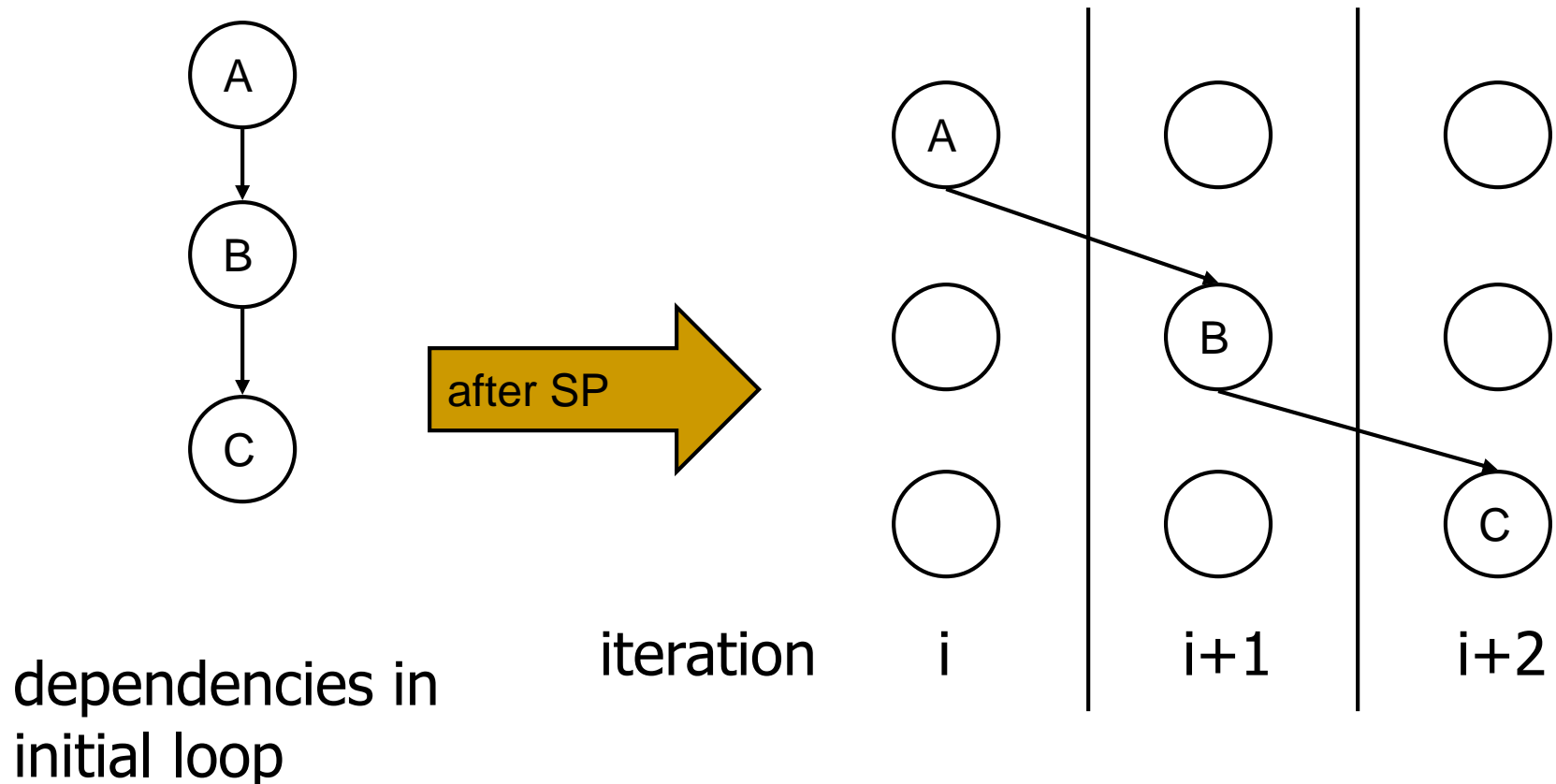
C2: $\text{st}[d2], b2$

Epilog

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

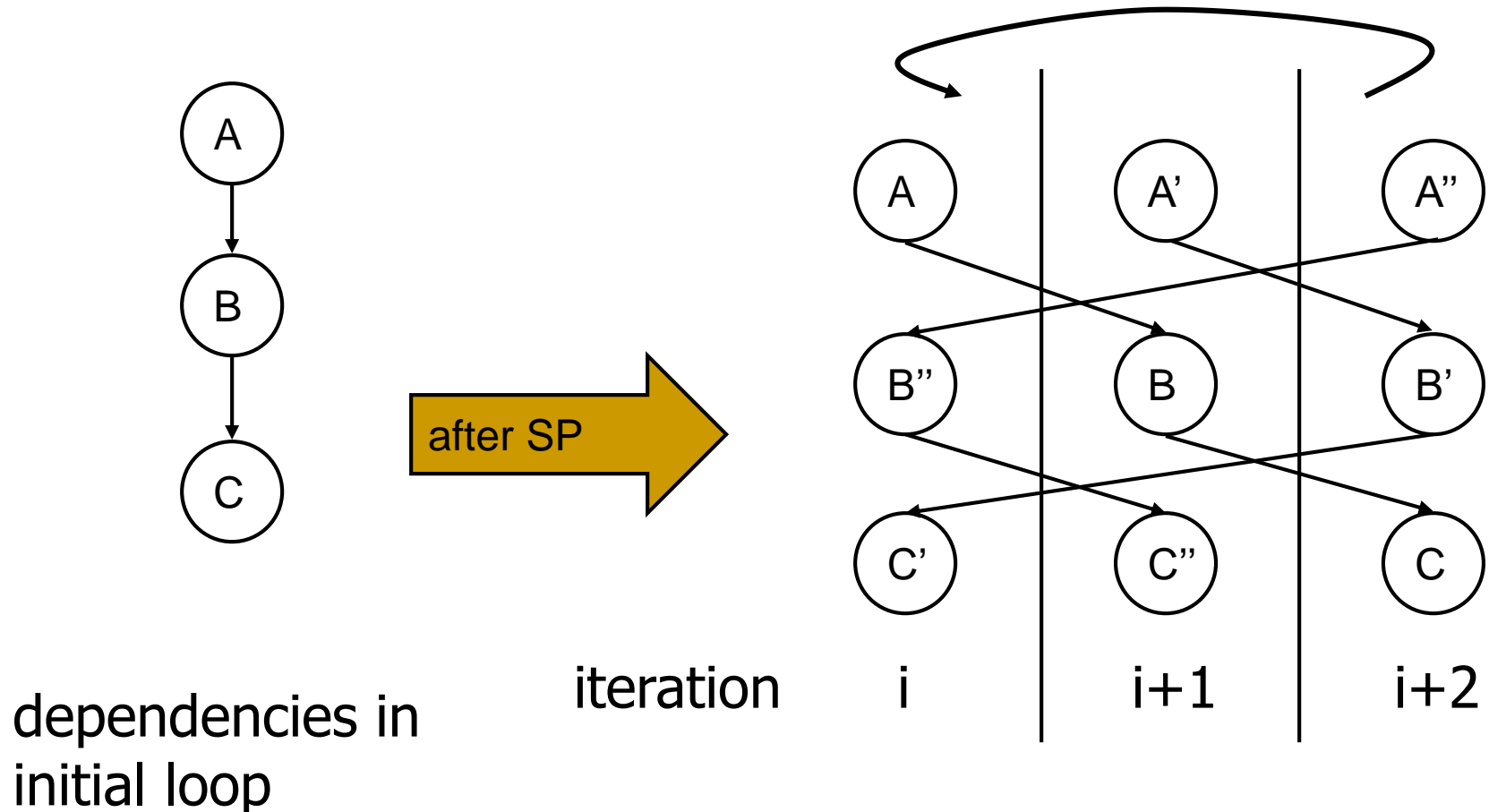
Goal of Software Pipelining

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



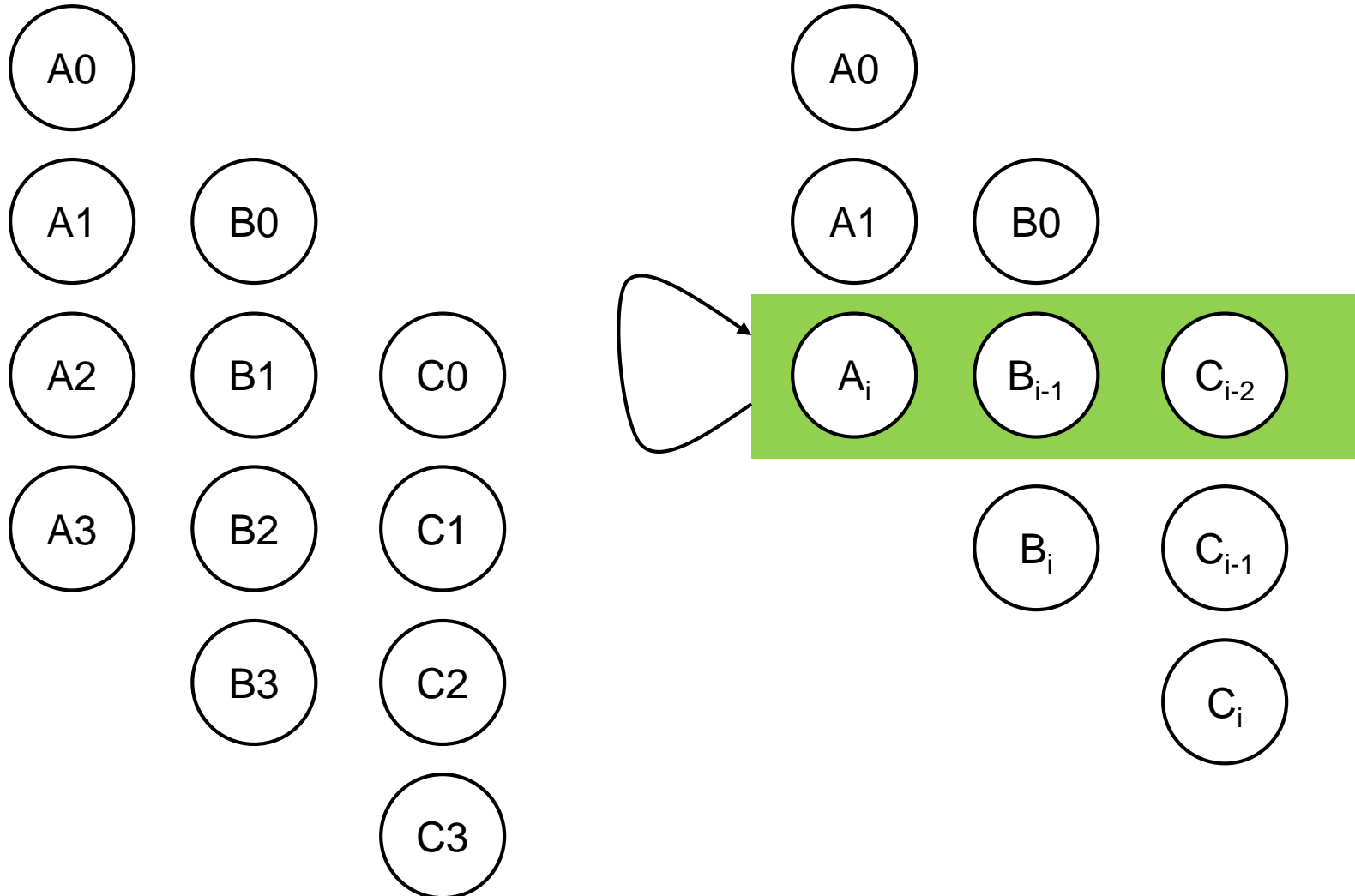
Goal of Software Pipelining

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



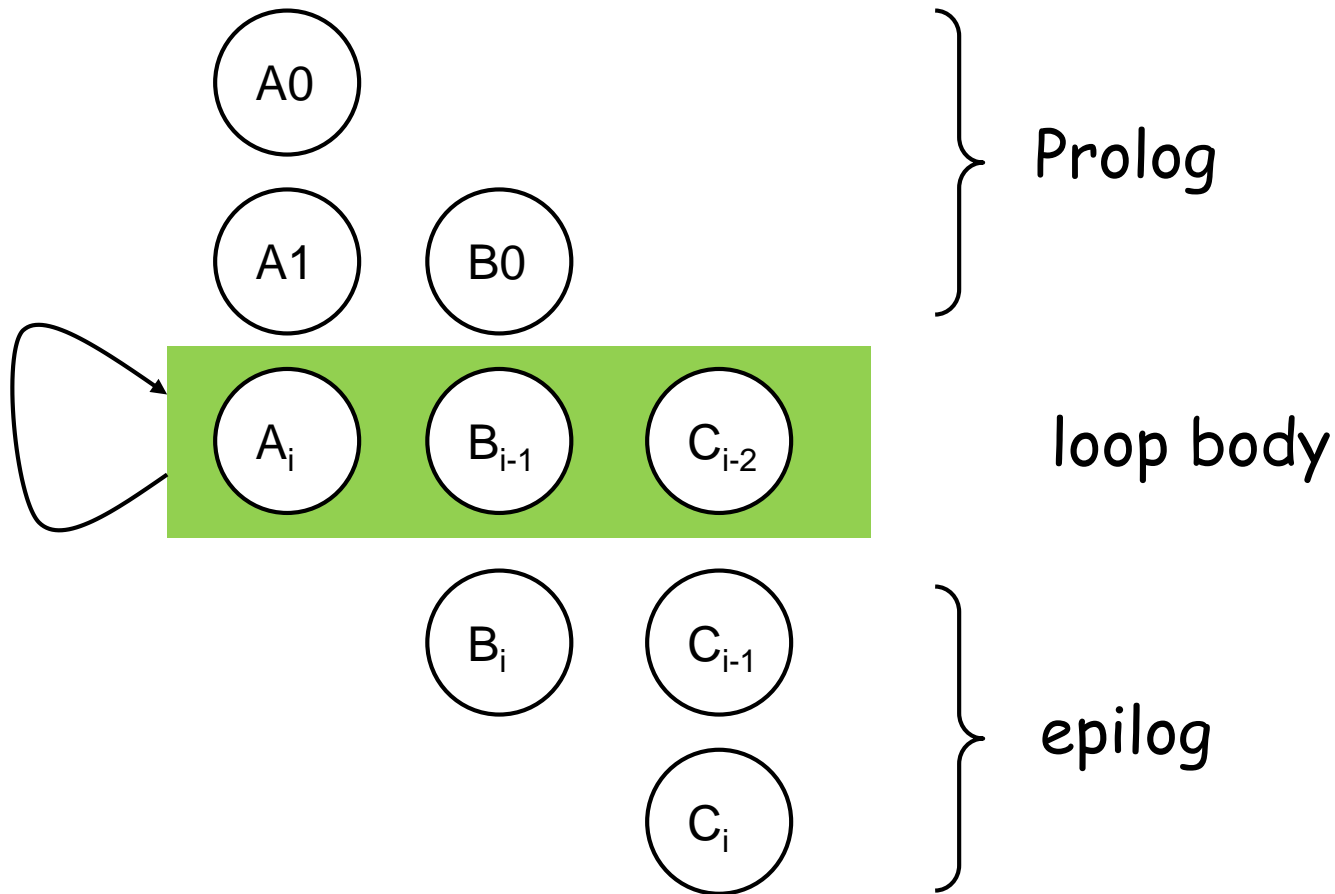
Goal of Software Pipelining

- Discover ILP across iterations!



Example

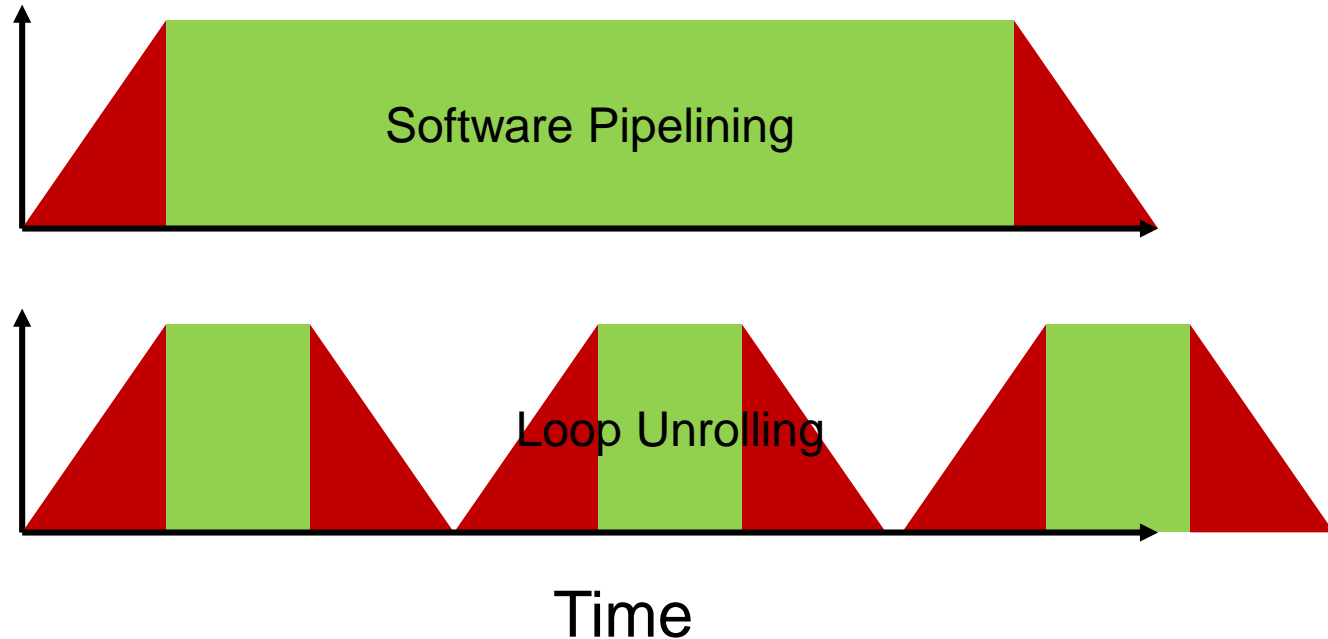
Assume operating on a infinite wide machine



Loop Unrolling vs. Software Pipelining

For SuperScalar or VLIW

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



More complicated SP example

- Functional units

- 1x LD – 2 cycles
- 1x FP – add 1 cycle, multiply 2 cycles
- 2x Integer – ALU 1 cycle, branch 1 cycle

- Code:

```
double sum_array(int N, double *A) {
    int sum = 0;
    for(int i=0; i<N; i++){
        sum += A[i];
    }
}
```

More complicated SP example

- Functional units
 - 1x LD – 2 cycles
 - 1x FP – add 1 cycle, multiply 2 cycles
 - 2x Integer – ALU 1 cycle, branch 1 cycle

- Software pipelined:

	INT1	INT2	FP	MEM
				LD F1 ← 0(R1)
				LD F2 ← 8(R1)
LOOP:	R1 ← R1+16	R0 ← R0 - 2	F0 ← F0 + F1	LD F1 ← 16(R1)
		BNEQ LOOP	F0 ← F0 + F2	LD F2 ← 8(R1)
			F0 ← F0 + F1	
			F0 ← F0 + F2	

Software Pipelining Approaches

- 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).

- "Iterative Modulo Scheduling" Rau MICRO'94
 - Addresses resource constraints
 - Iterate over different loop lengths until one schedules successfully
 - Compute loop lower/upper bounds from required & available resources

TRACE SCHEDULING

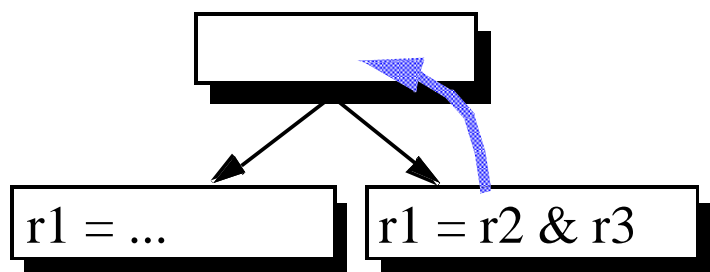
Extending the scheduling domain

- Basic block is too small to get any real parallelism
 - Recall: 88% of OOO benefit from speculation → larger scheduling window
- 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
 - Thus profiling may be very important

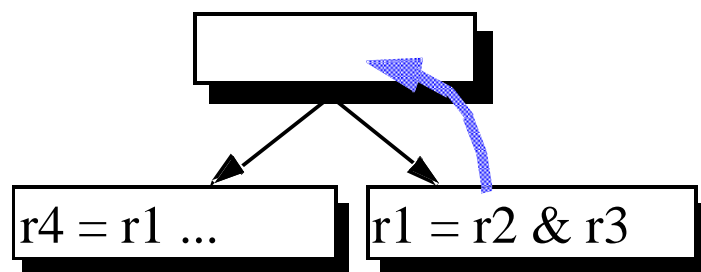
[MTZ'13]

Safety and Legality in Code Motion

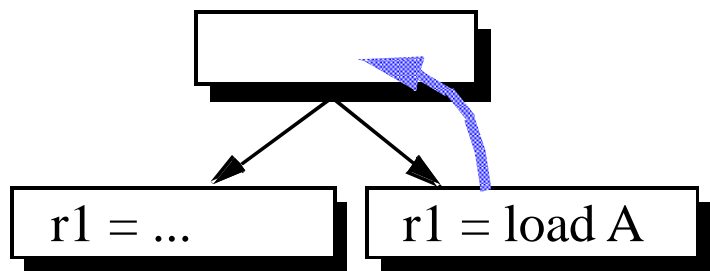
- Two characteristics of speculative code motion:
 - Safety: whether or not spurious exceptions may occur
 - Legality: whether or not result will be always correct
- Four possible types of code motion:



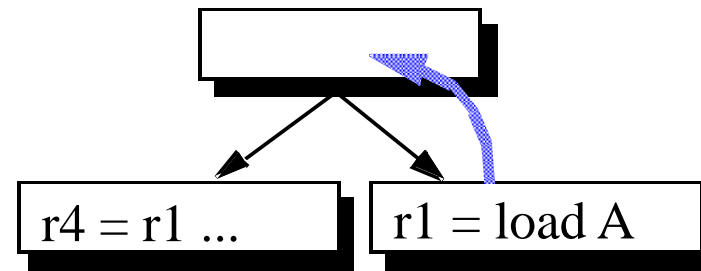
(a) safe and legal



(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

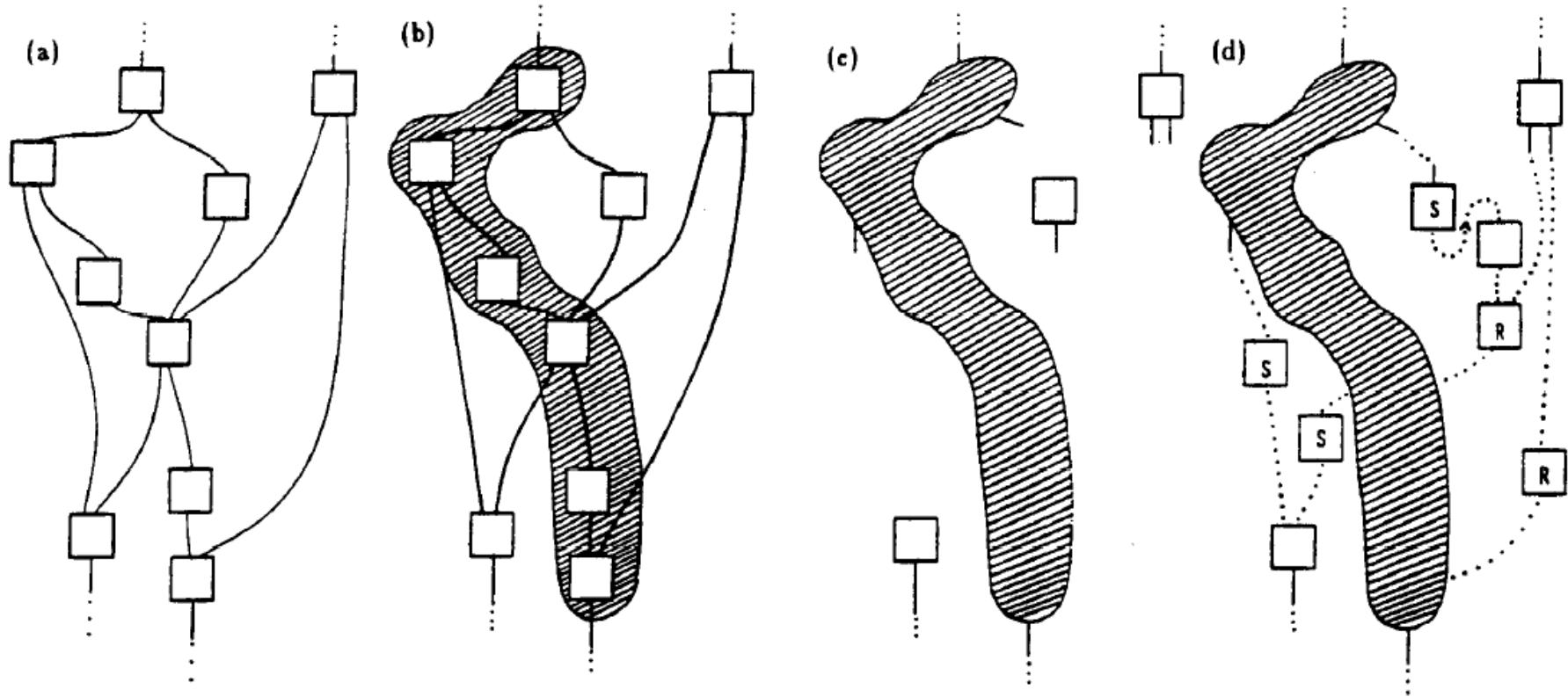
■ Upward

- 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

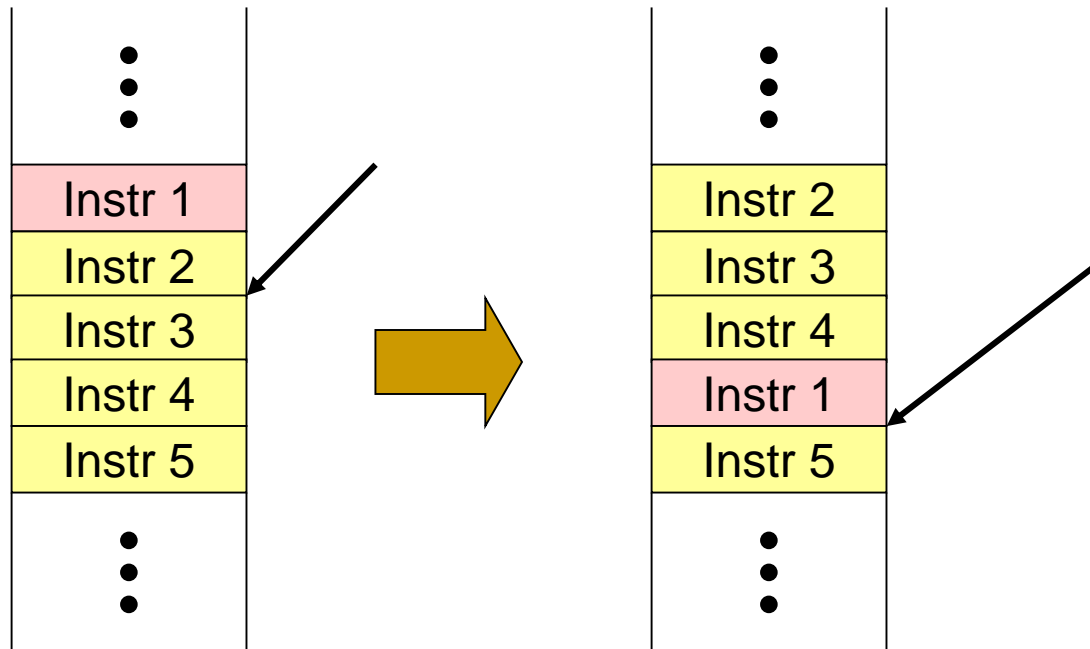


TRACE SCHEDULING LOOP-FREE CODE

Trace Scheduling (II)

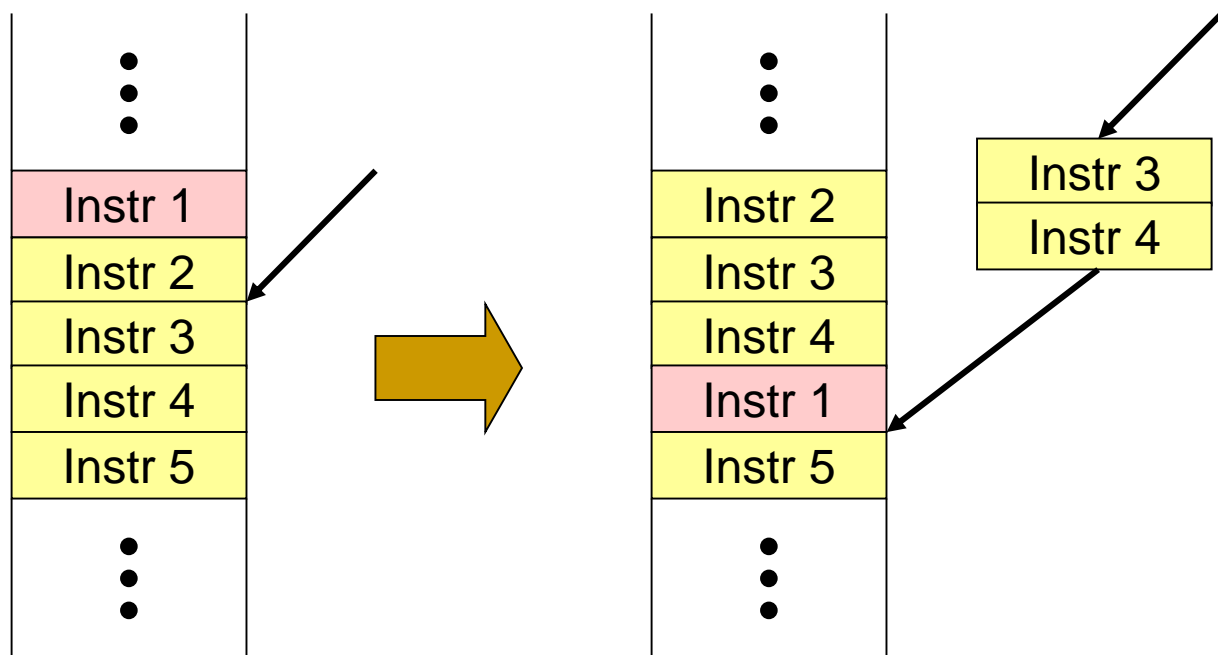
- There may be conditional branches from the middle of the trace (**side exits**) and transitions from other traces into the middle of the trace (**side entrances**).
- These control-flow transitions are ignored during trace scheduling.
- After scheduling, fix-up/bookkeeping code is inserted to ensure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

Trace Scheduling (III)

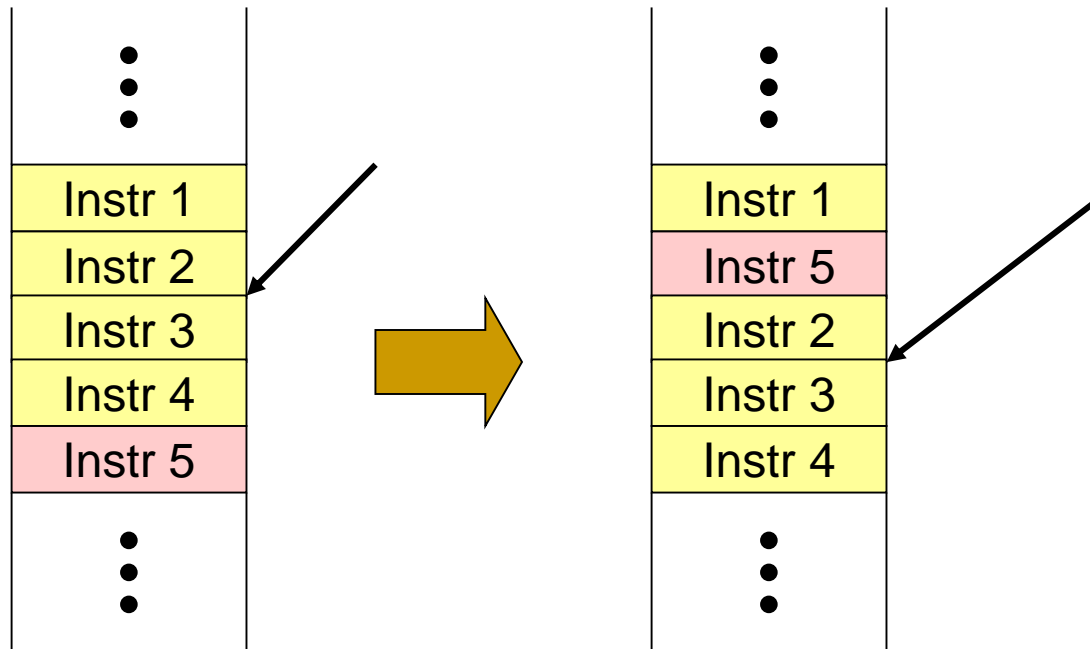


What bookkeeping is required when **Instr 1** is moved below the side entrance in the trace?

Trace Scheduling (IV)

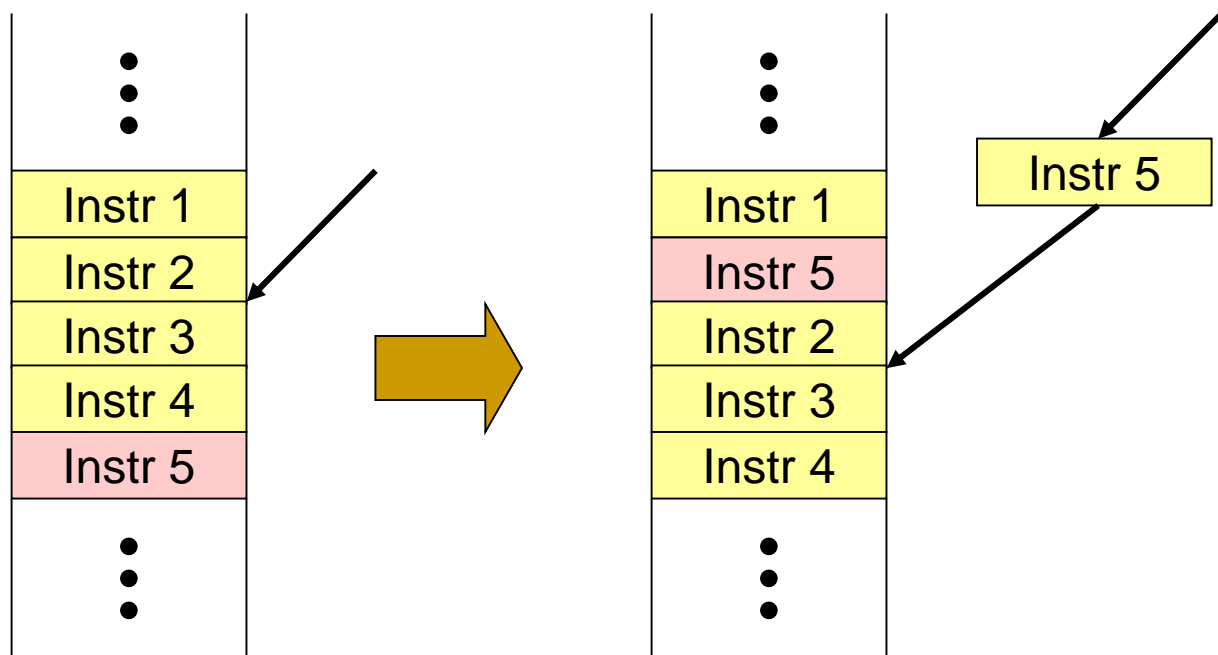


Trace Scheduling (V)



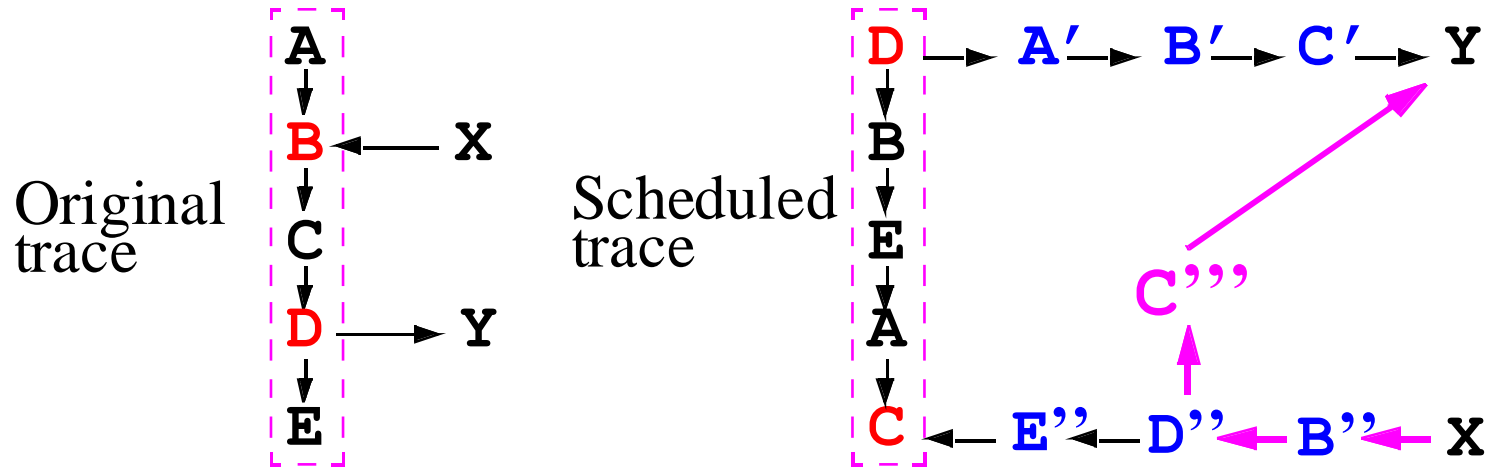
What bookkeeping is required when **Instr 5** moves above the side entrance in the trace?

Trace Scheduling (VI)



Trace Scheduling Fixup Code Issues

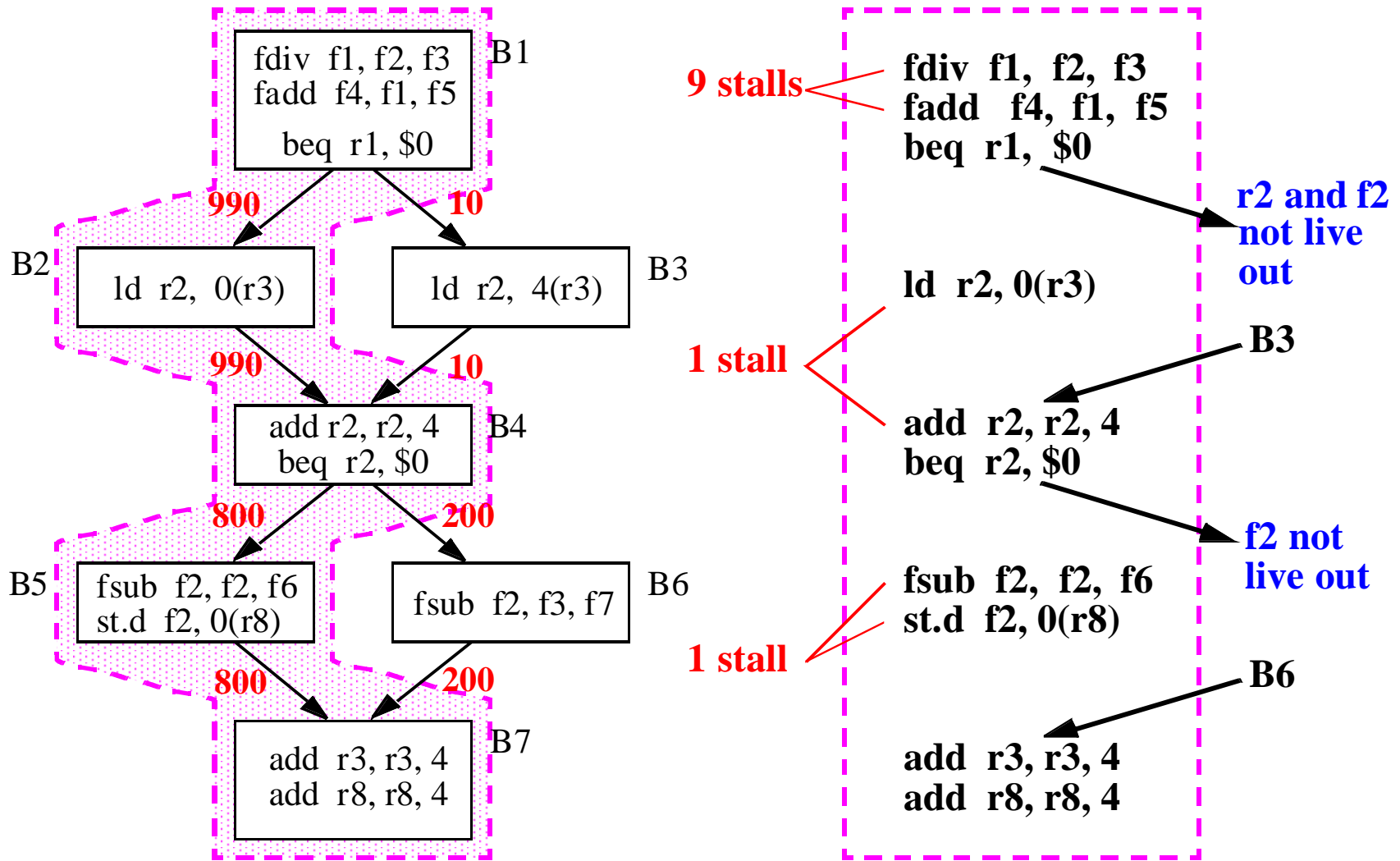
- Sometimes need to copy instructions more than once to ensure correctness on all paths (see C below)



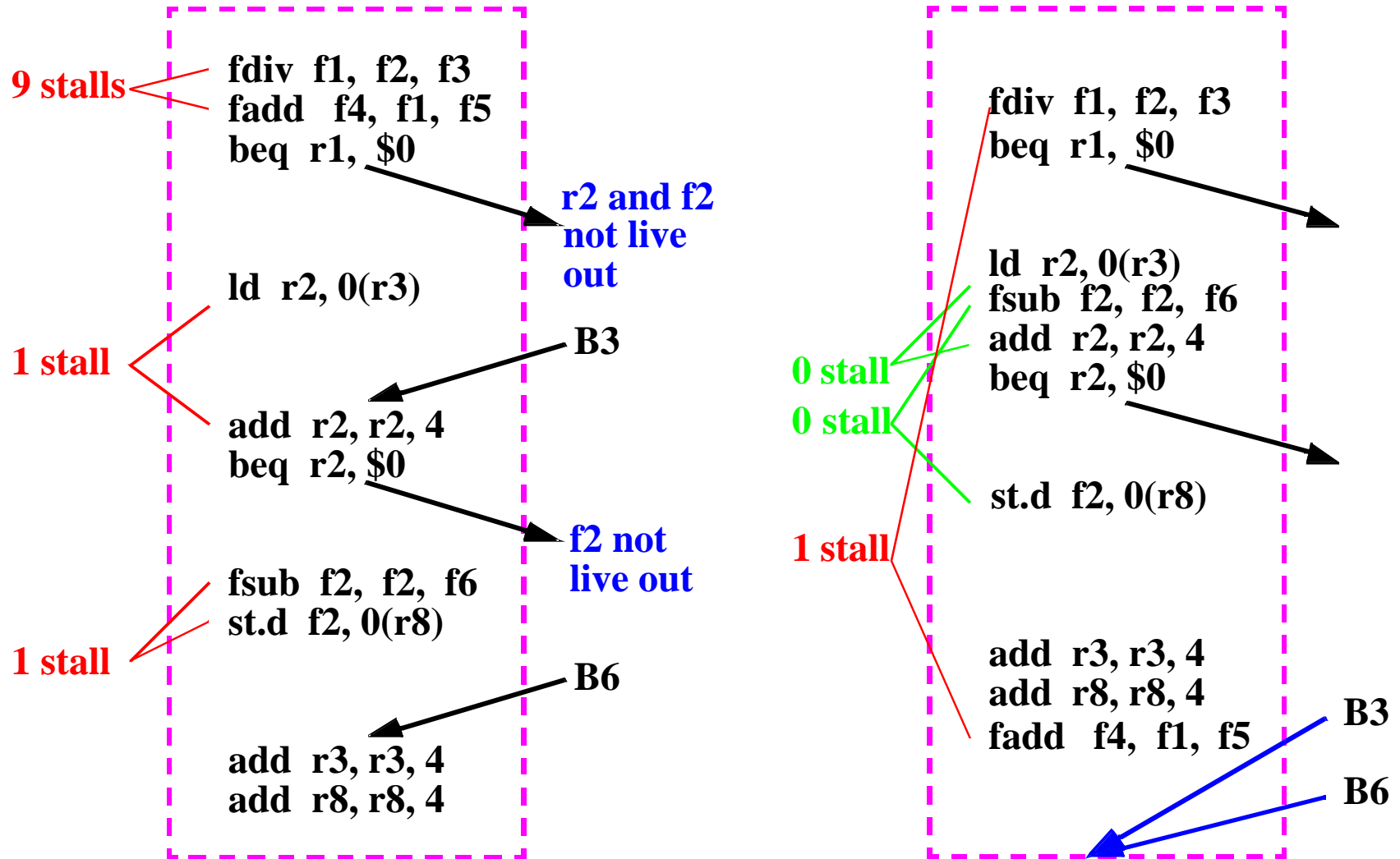
Trace Scheduling Overview

- Trace Selection
 - select seed block (the highest frequency basic block)
 - extend trace (along the highest frequency edges)
 - forward (successor of the last block of the trace)
 - backward (predecessor of the first block of the trace)
 - don't cross loop back edge
 - bound `max_trace_length` heuristically
- Trace Scheduling
 - build **data precedence graph** for a whole trace
 - perform **list scheduling** and **allocate registers**
 - add compensation code to maintain semantic correctness
- Speculative Code Motion (upward)
 - move an instruction above a branch if safe

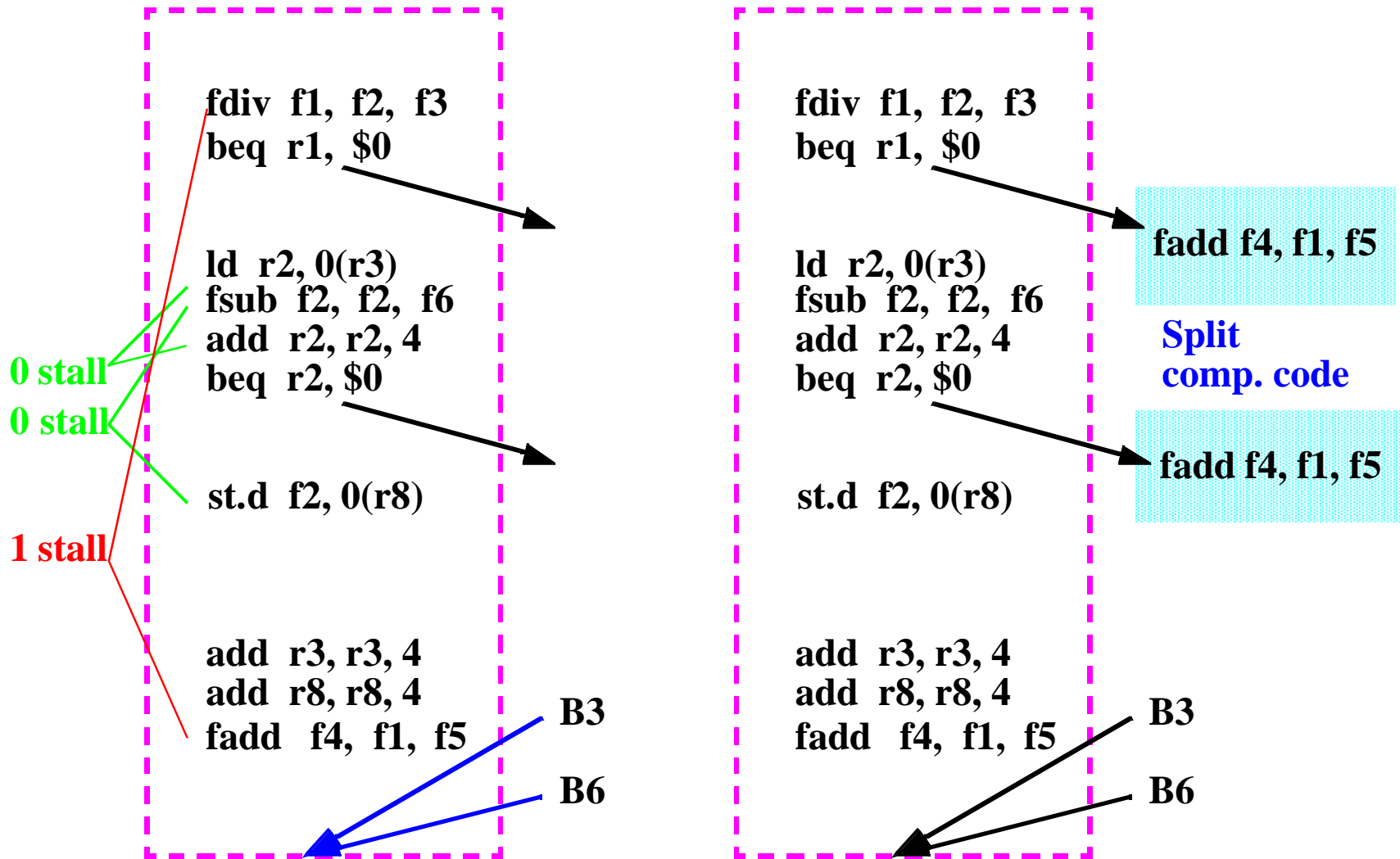
Trace Scheduling Example (I)



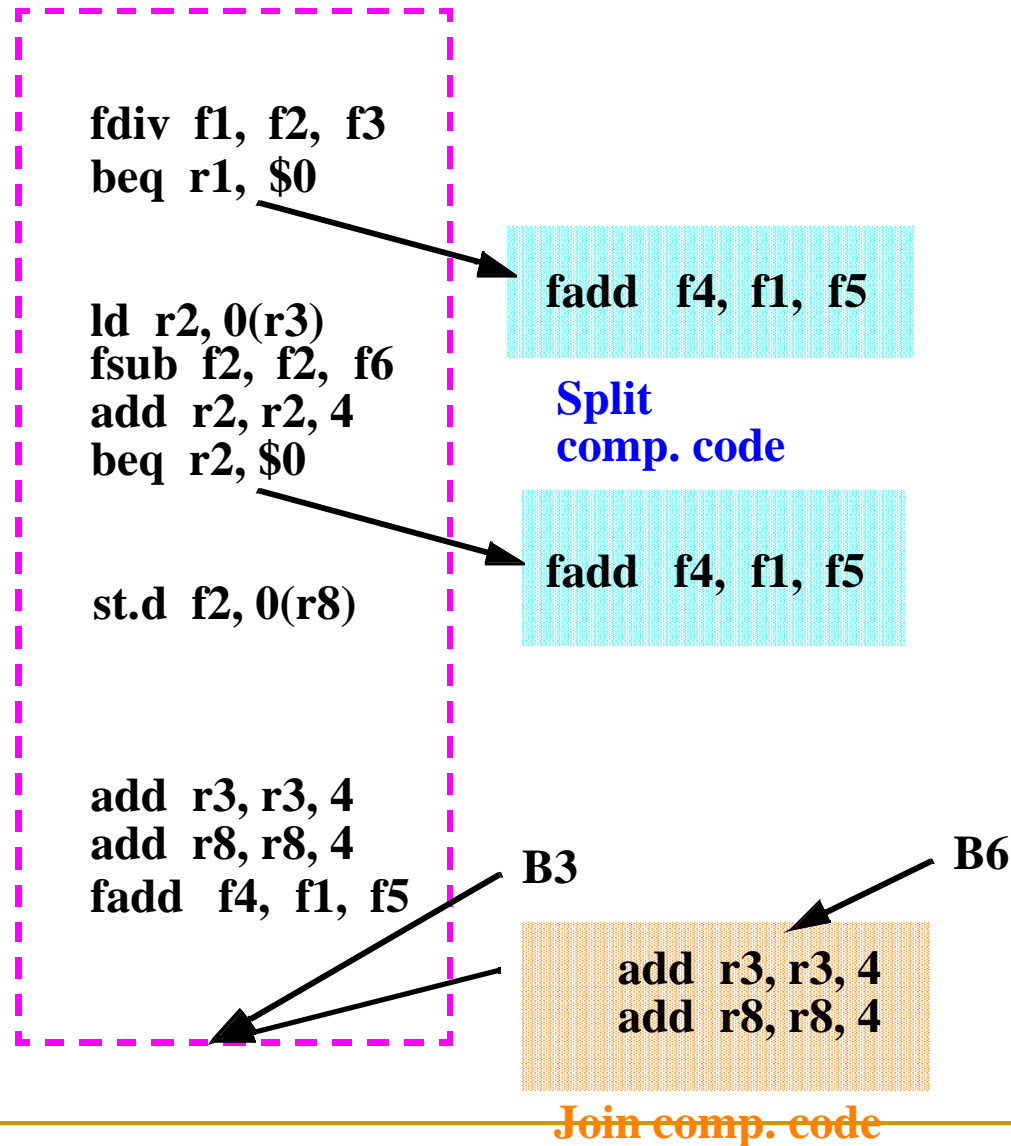
Trace Scheduling Example (I)



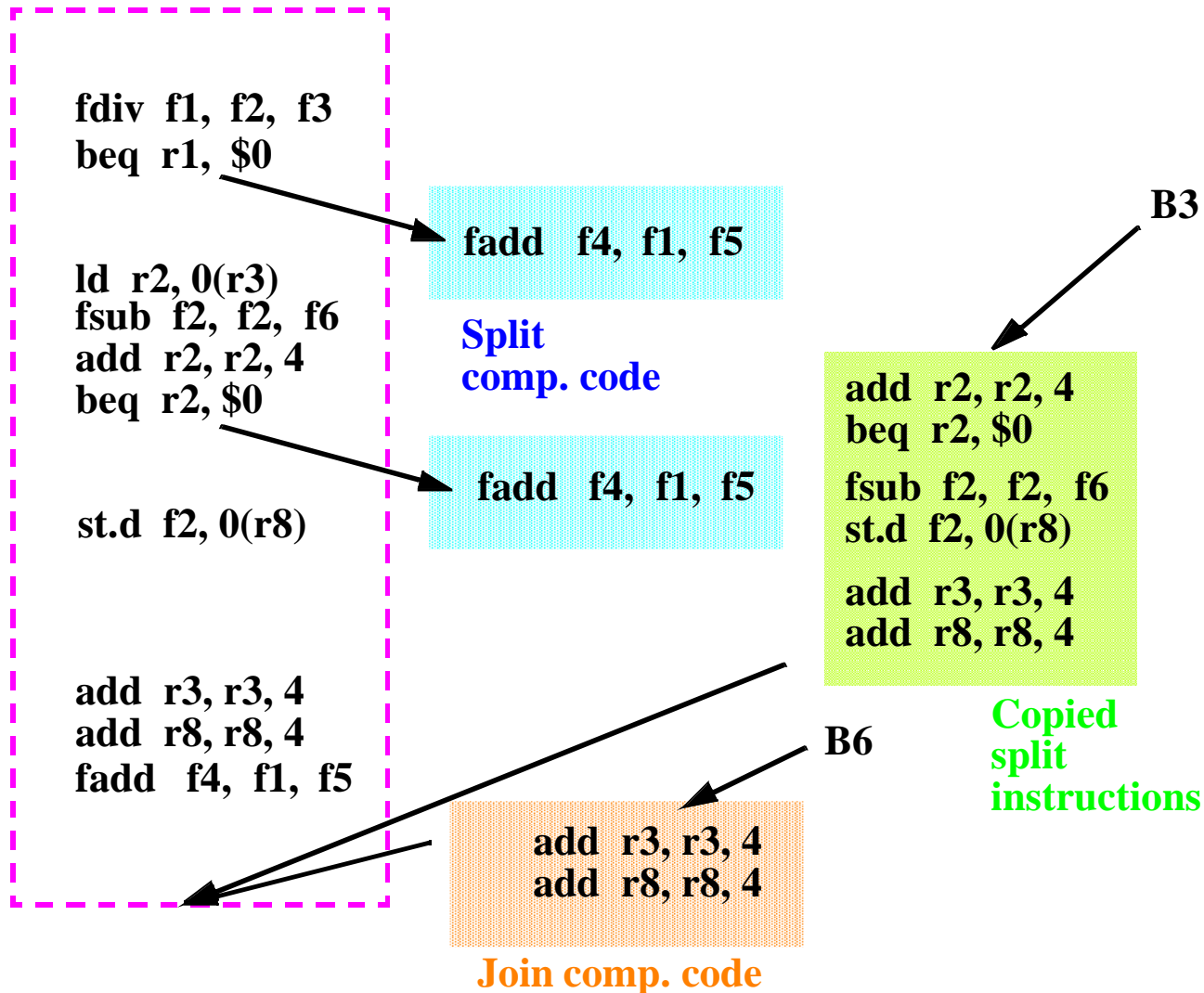
Trace Scheduling Example (II)



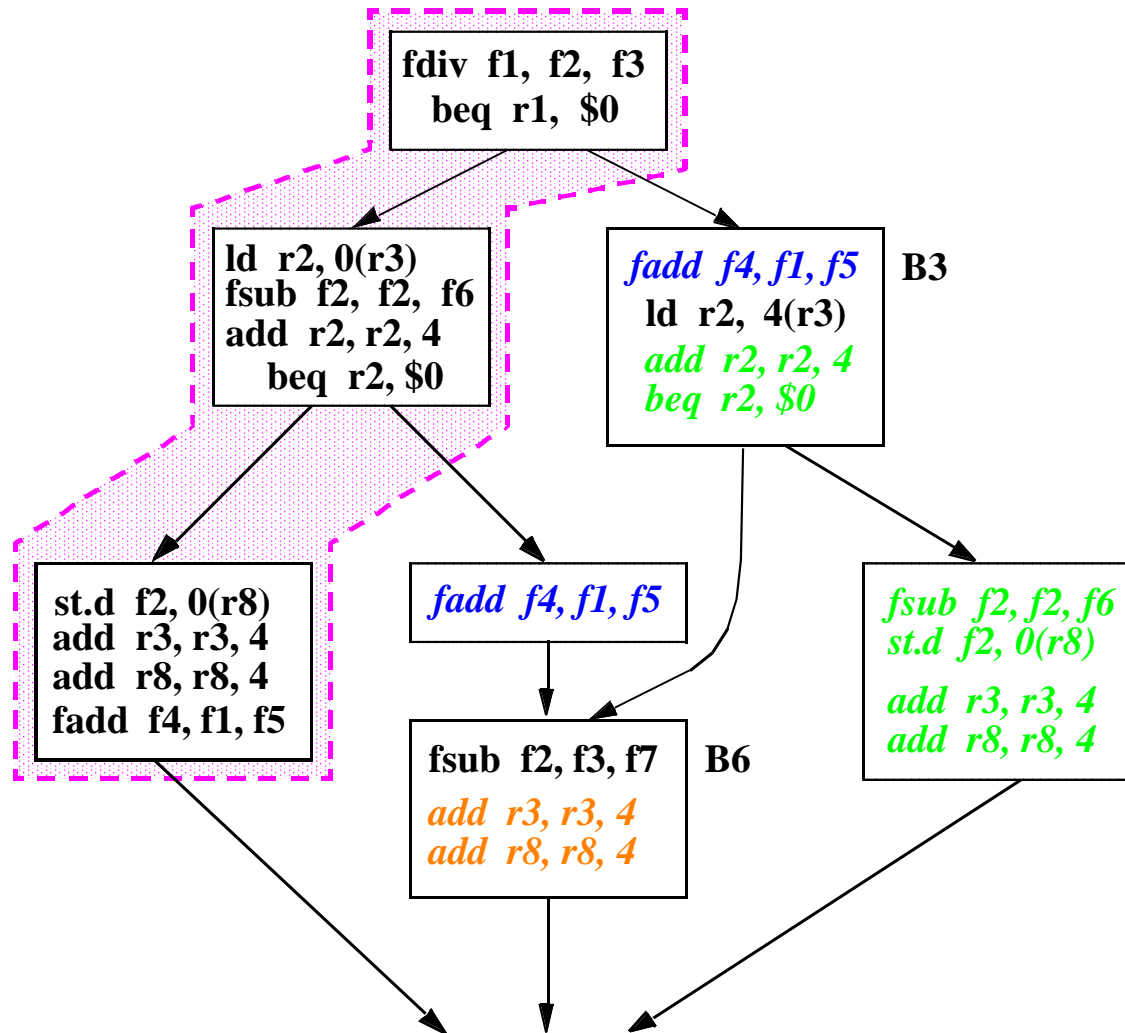
Trace Scheduling Example (III)



Trace Scheduling Example (IV)



Trace Scheduling Example (V)



Trace Scheduling Tradeoffs

■ Advantages

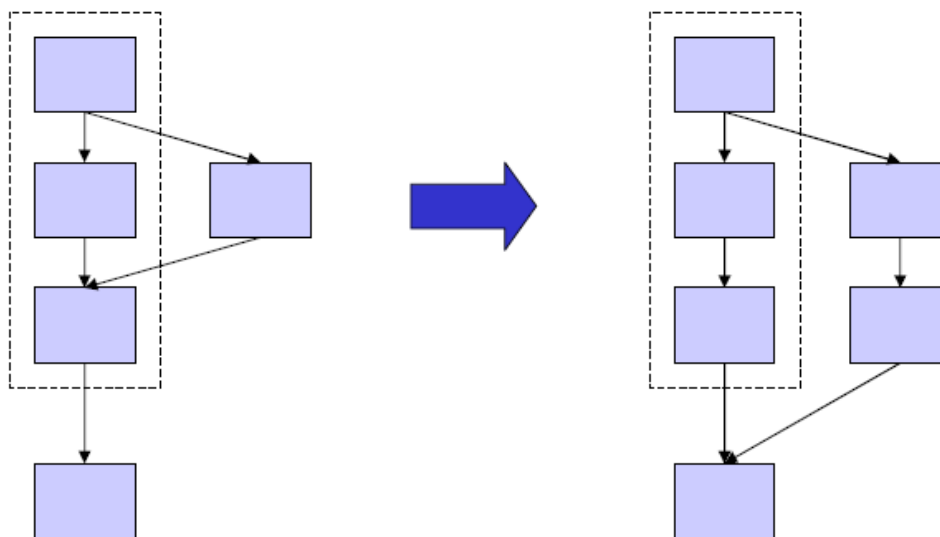
- + Enables the finding of more independent instructions → fewer NOPs in a VLIW instruction

■ Disadvantages

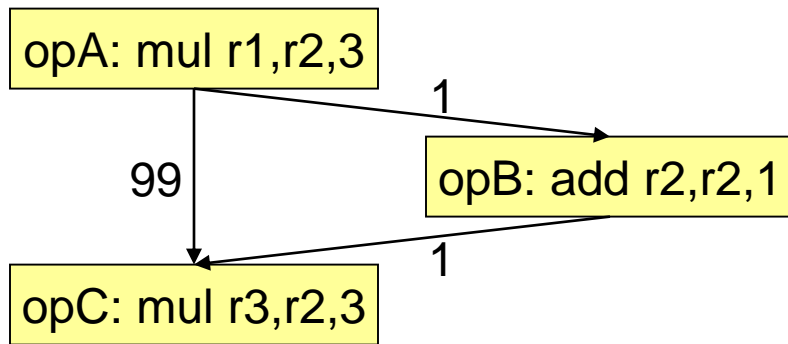
- Profile dependent
 - What if dynamic path deviates from trace → lots of NOPs in the VLIW instructions
- Code bloat and additional fix-up code executed
 - Due to side entrances and side exits
 - **Infrequent paths interfere with the frequent path**
- Effectiveness depends on the bias of branches
 - Unbiased branches → smaller traces → less opportunity for finding independent instructions

Superblock Scheduling

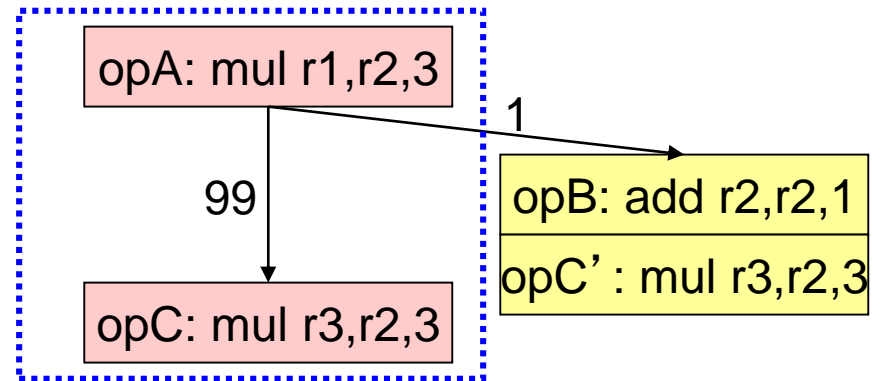
- Trace: multiple entry, multiple exit block
 - Superblock: single-entry, multiple exit block
 - A trace with side entrances are eliminated
 - Infrequent paths do not interfere with the frequent path
- + More optimization/scheduling opportunity than traces
- + Eliminates “difficult” bookkeeping due to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



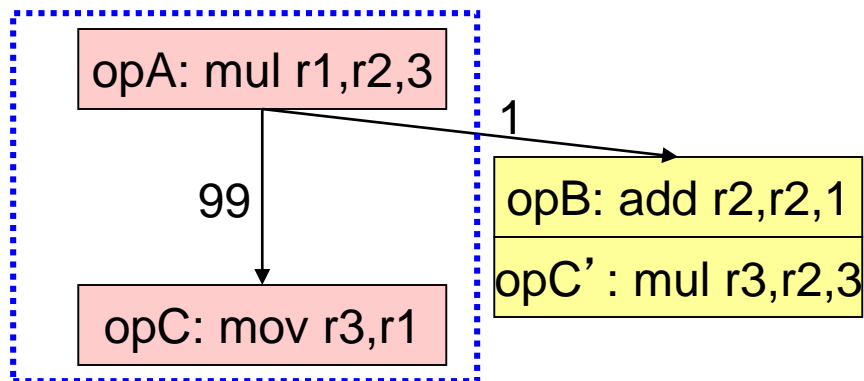
Superblock example



Original Code



Code After Superblock Formation



Code After Common Subexpression Elimination

Superblock Scheduling Shortcomings

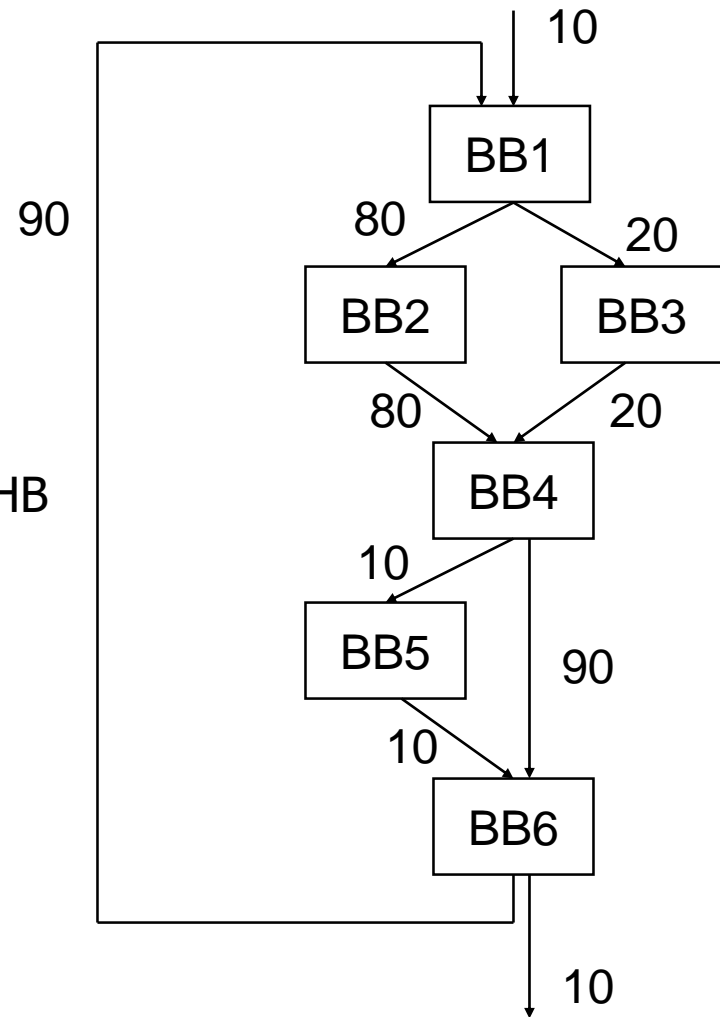
- Still profile-dependent
- No single frequently executed path if there is an unbiased branch
 - Reduces the size of superblocks
- Code bloat and additional fix-up code executed
 - Due to side exits

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)

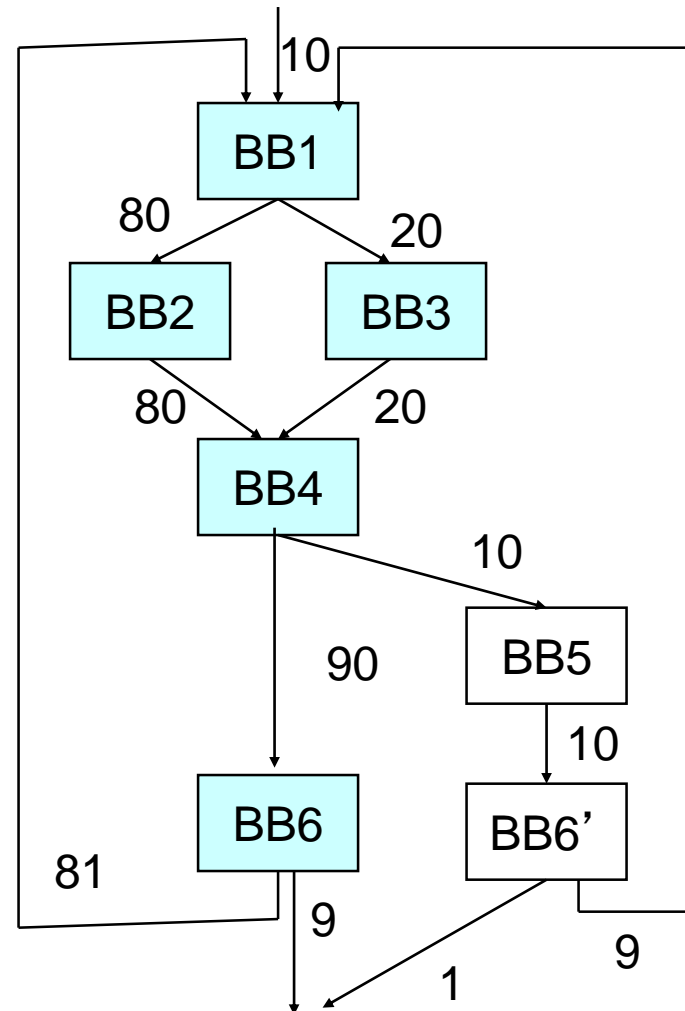
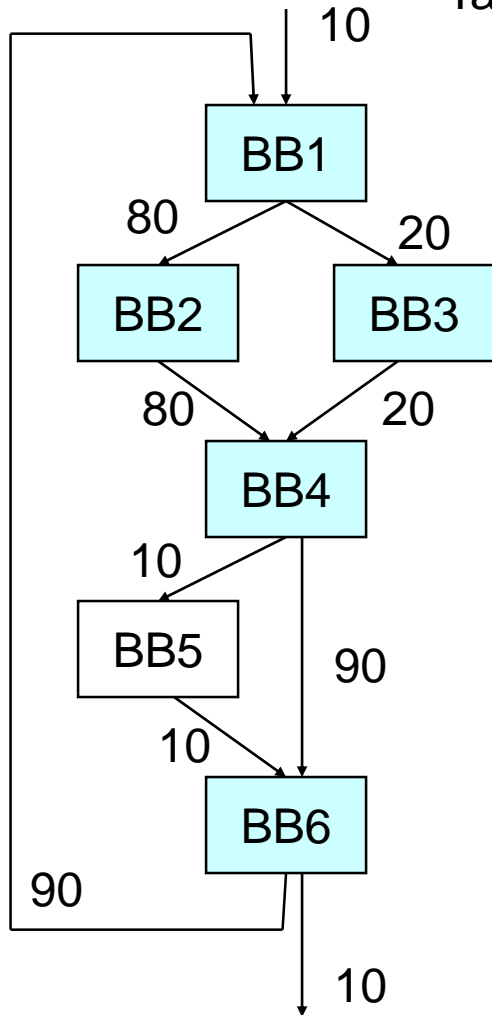
- Hyperblock formation
 1. Block selection
 2. Tail duplication
 3. If-conversion
- Block selection
 - Select subset of BBs for inclusion in HB
 - Difficult problem
 - Weighted cost/benefit function
 - Height overhead
 - Resource overhead
 - Dependency overhead
 - Branch elimination benefit
 - Weighted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



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

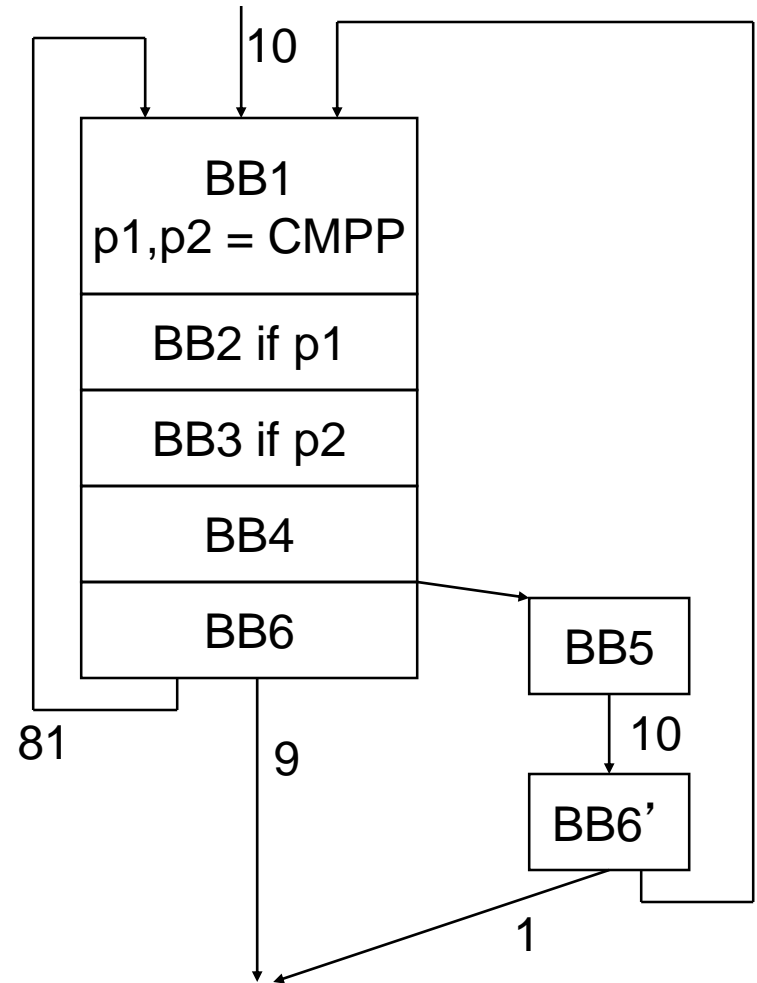
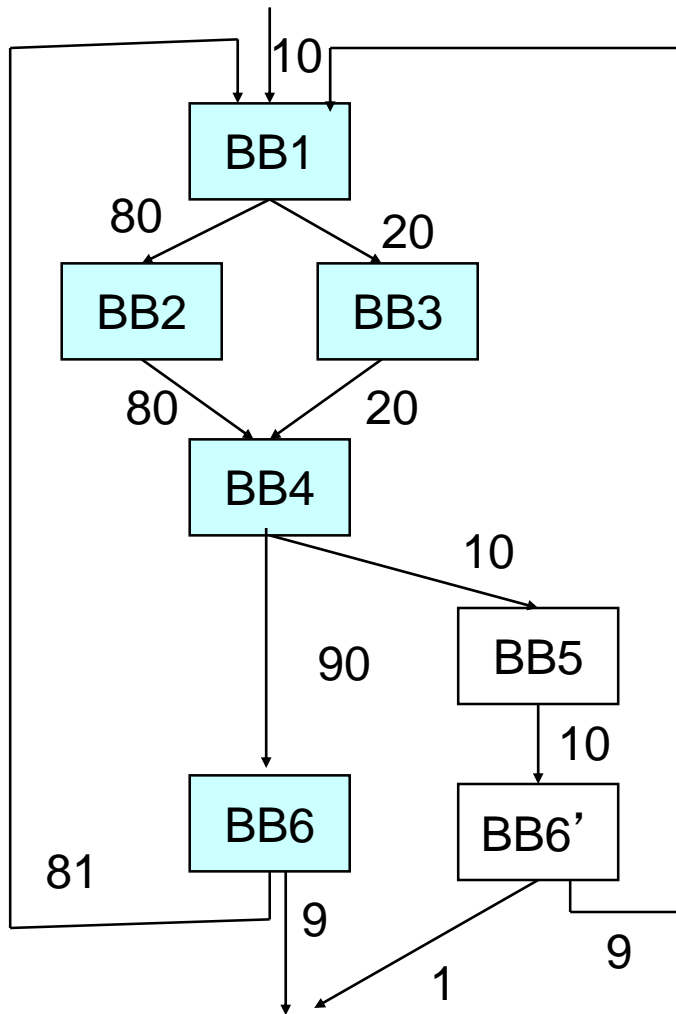
Hyperblock Formation (II)

Tail duplication same as with Superblock formation



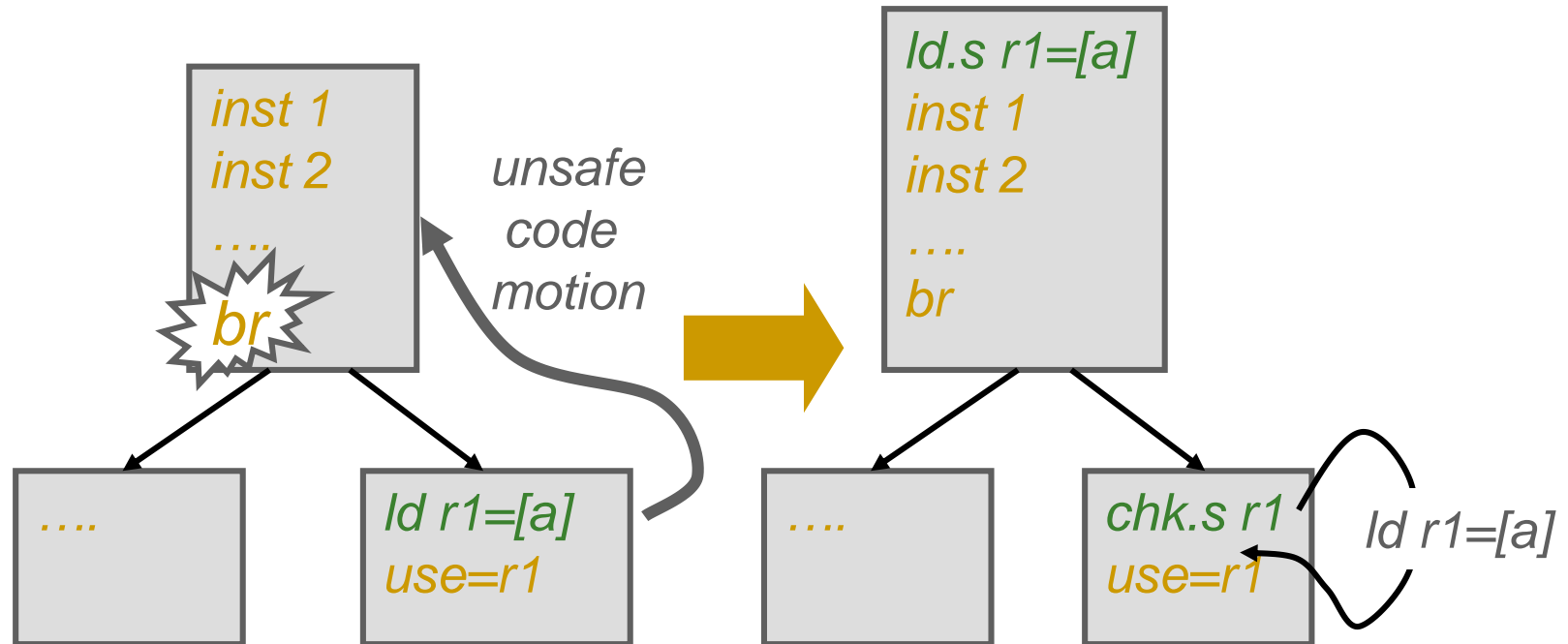
Hyperblock Formation (III)

If-convert (predicate) intra-hyperblock branches



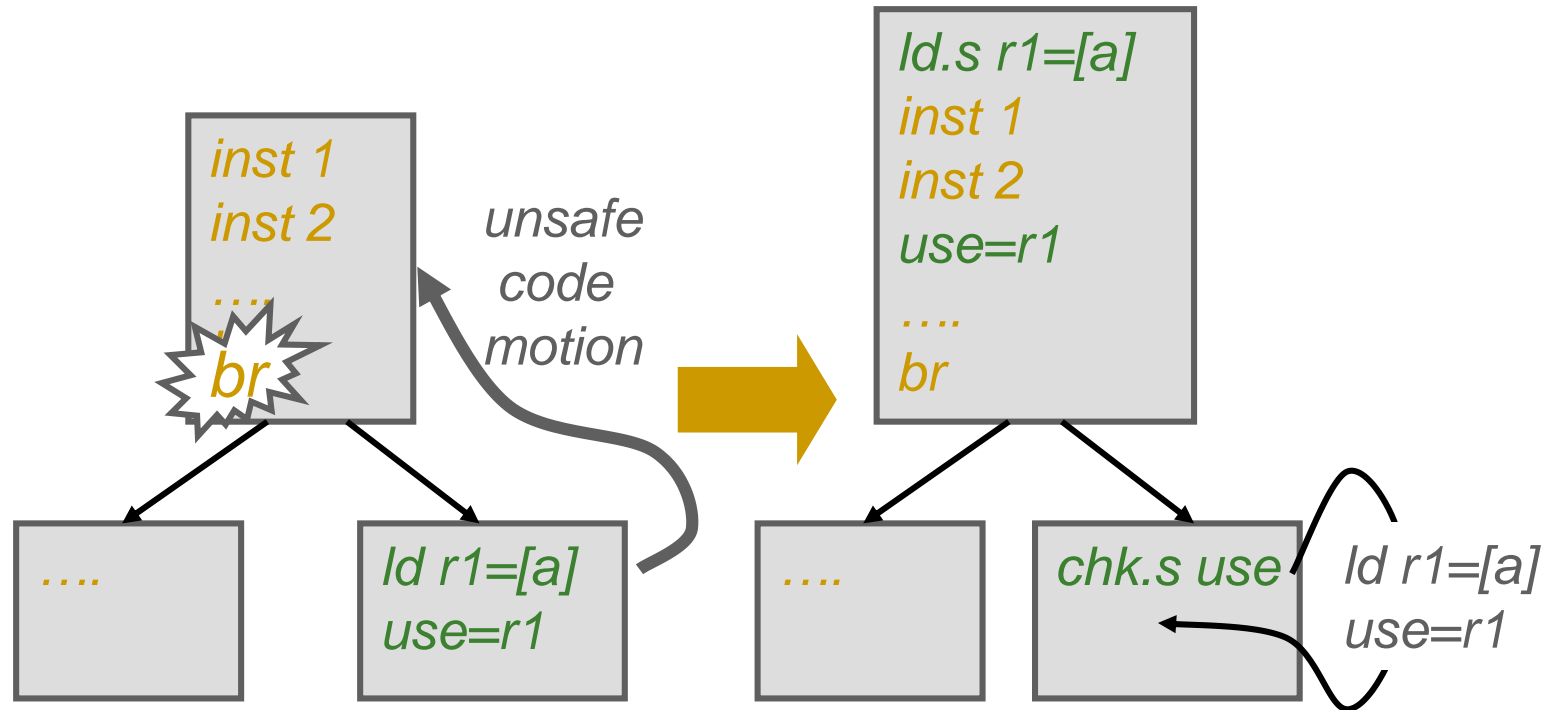
WHAT ABOUT MEMORY?

Non-Faulting Loads and Exception Propagation



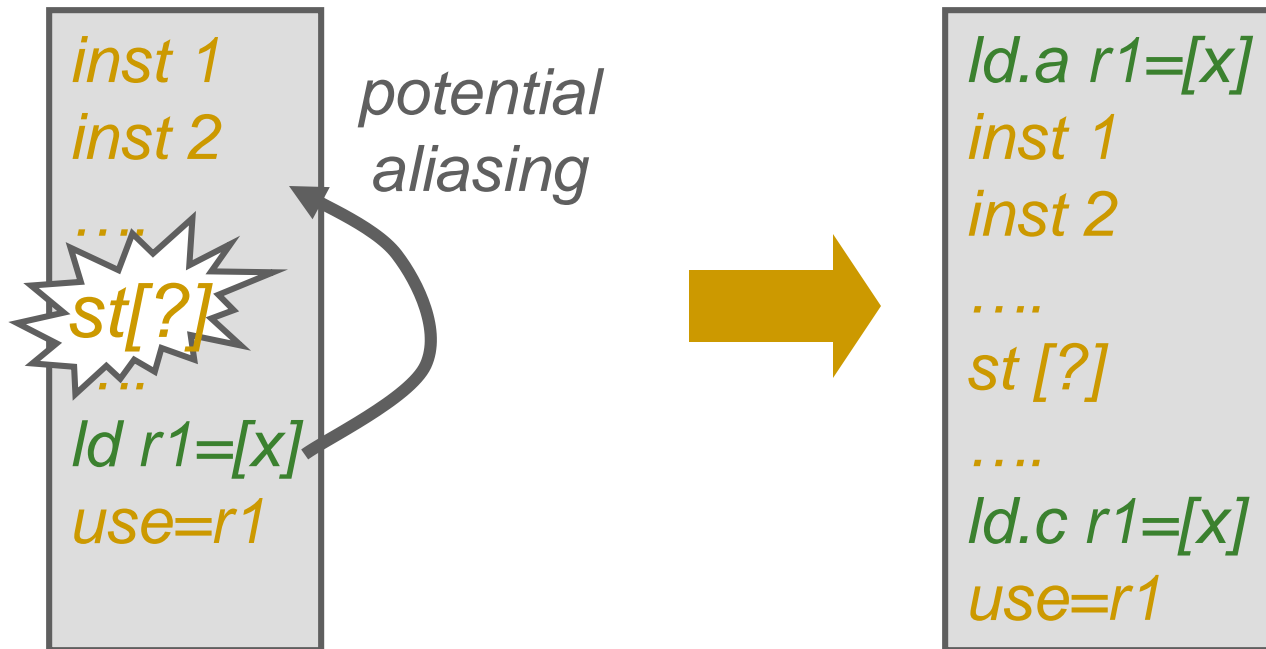
- *ld.s* fetches *speculatively* from memory
i.e. any exception due to *ld.s* is suppressed
- If *ld.s r1* did not cause an exception then *chk.s r1* is a NOP, else a branch is taken (to execute some compensation code)

Non-Faulting Loads and Exception Propagation in IA-64



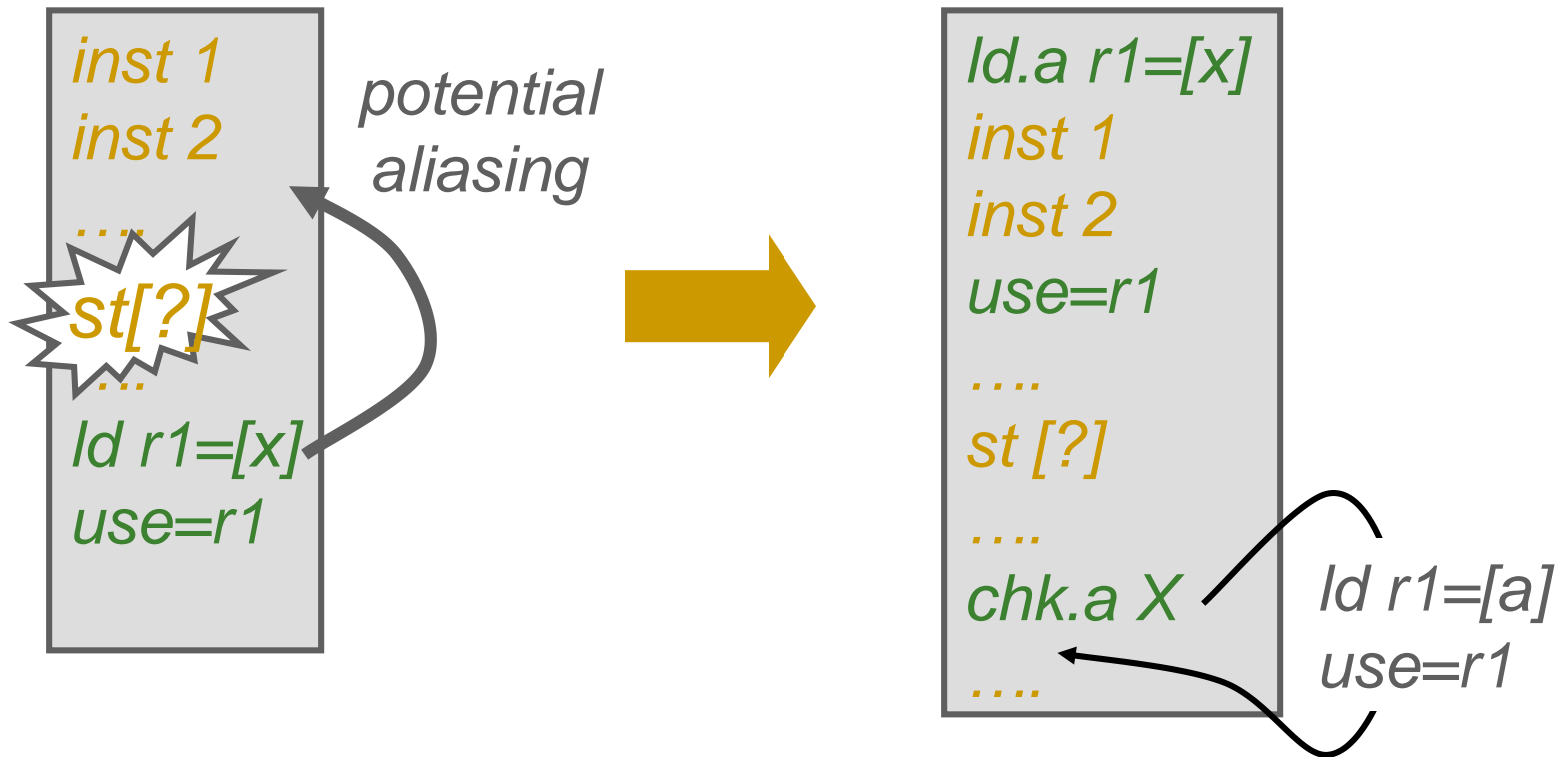
- Load data can be speculatively consumed prior to check
- “speculation” status is propagated with speculated data
- Any instruction that uses a speculative result also becomes speculative itself (i.e. suppressed exceptions)
- *chk.s* checks the entire dataflow sequence for exceptions

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

VLIW Summary

- Heavy reliance on compiler (push RISC to the extreme)
 - Compiler algorithms (e.g., software pipelining) have lasting impact outside of VLIW
- Is there enough statically knowable parallelism?
 - E.g., memory aliasing and branch bias
- What about wasted FUs? Code bloat?
 - Code size is already a big problem with x86 apps!
- Architecture joke: “VLIW is the architecture of the future, and always will be.”
- Yet many DSPs are VLIW. Why?

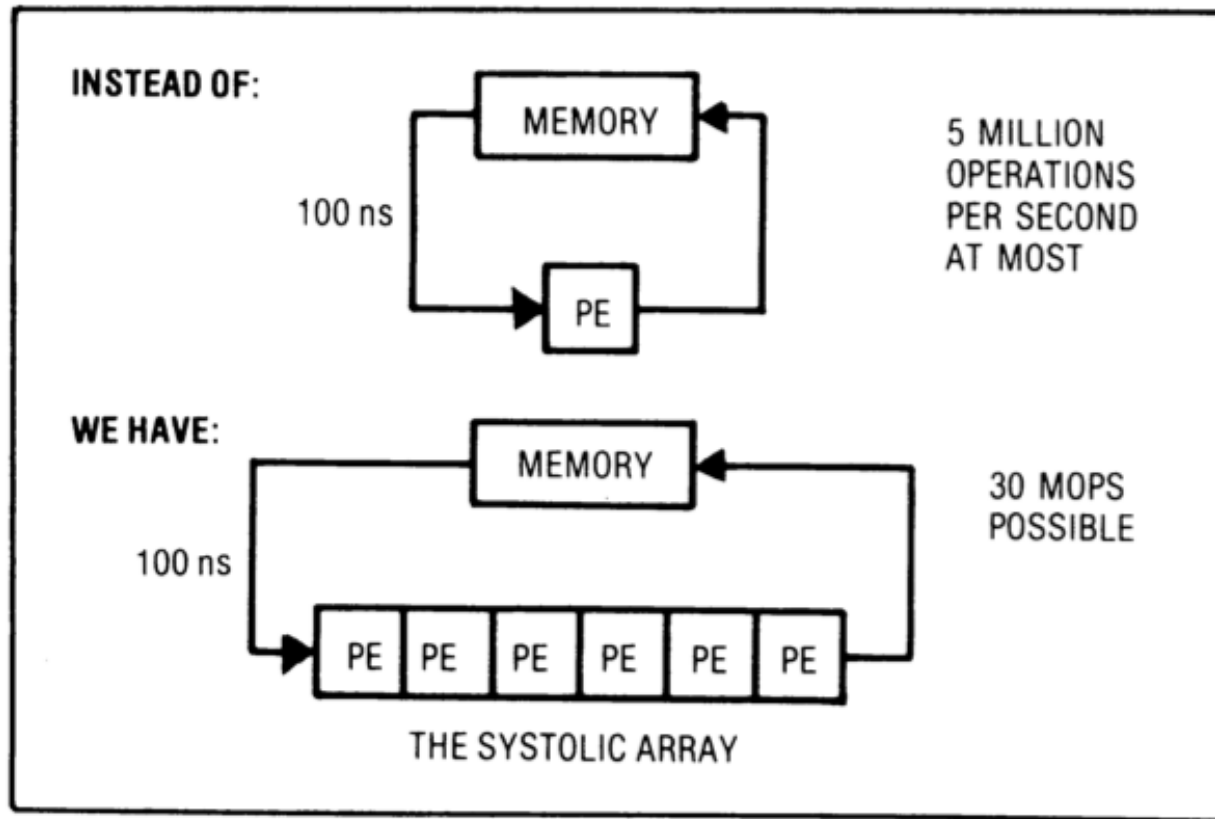
SYSTOLIC ARRAYS

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.



Memory: heart
PEs: cells

Memory pulses
data through
cells

Figure 1. Basic principle of a systolic system.

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

- 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)

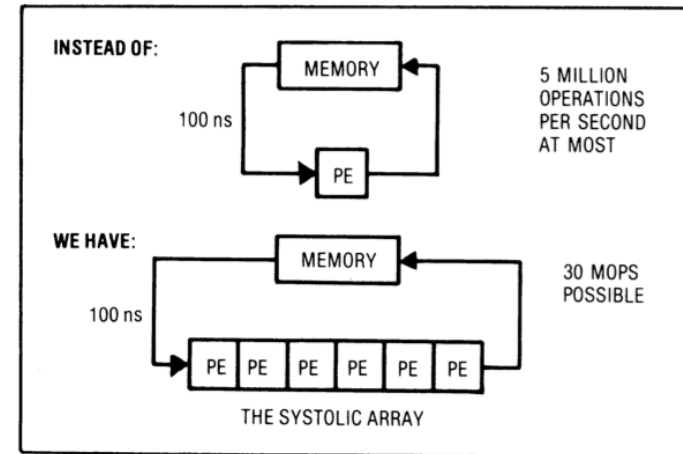


Figure 1. Basic principle of a systolic system.

Systemic Computation Example

- Convolution

- Given weights $w_1 \dots w_k$ and numbers $x_1 \dots x_n$

- Compute sequence $y_1 \dots y_{n+1-k}$ where

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

- Used often in image processing (eg, filtering), pattern matching, correlation, polynomial evaluation, etc ...

Systolic Computation Example: Convolution

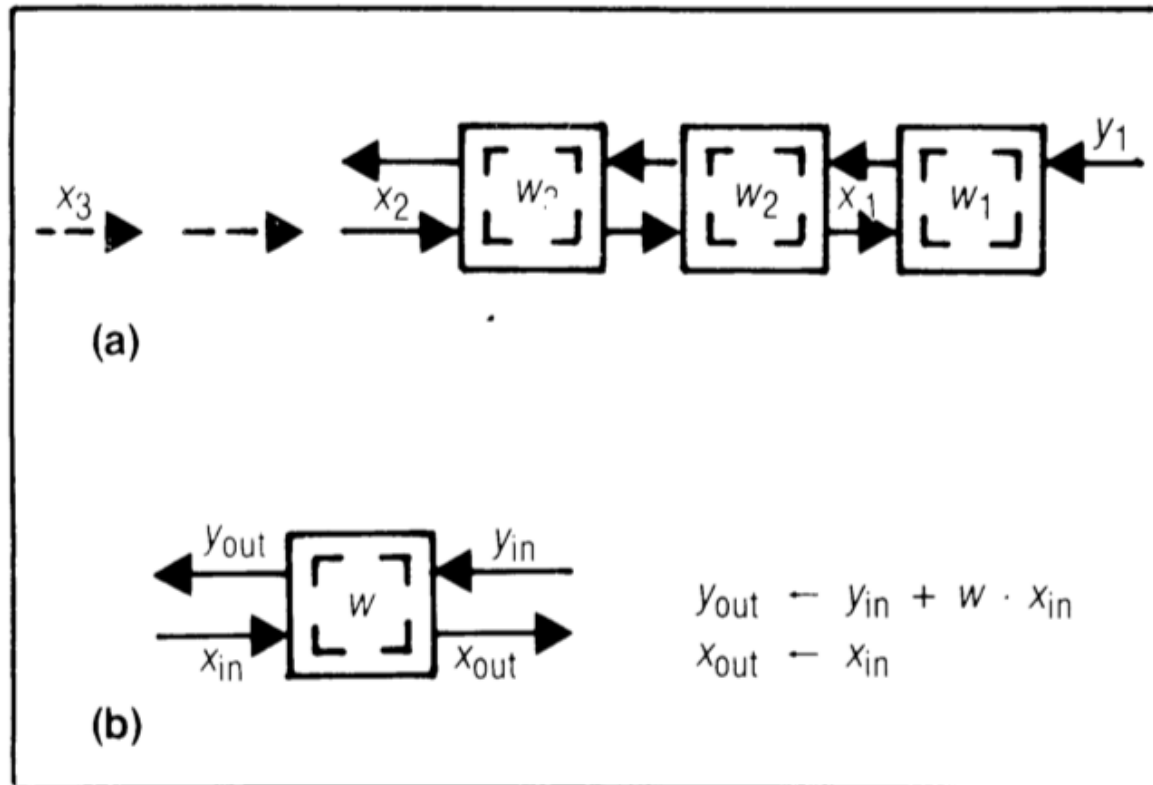
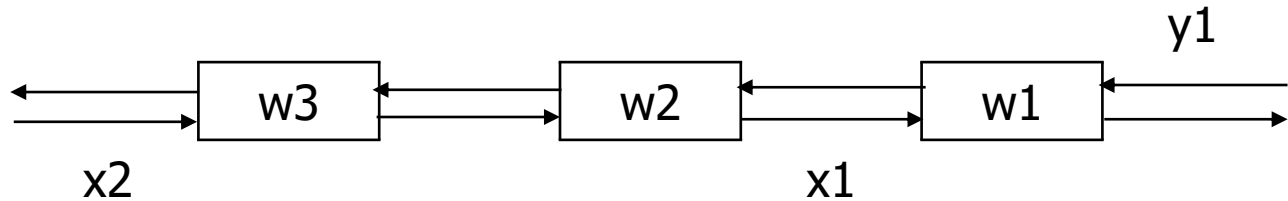
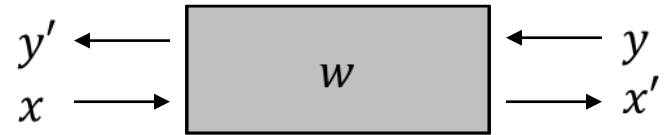


Figure 8. Design W1: systolic convolution array (a) and cell (b) where w_i 's stay and x_i 's and y_i 's move systolically in opposite directions.

Systemic Computation: Convolution

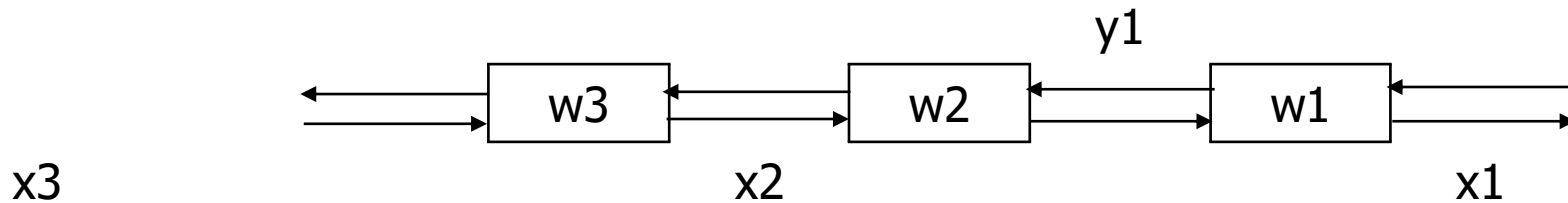
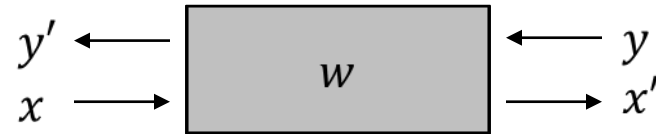
$$y' = y + w \cdot x$$
$$x' = x$$



$$y_1 = 0$$
$$y_2 = 0$$

Systolic Computation: Convolution

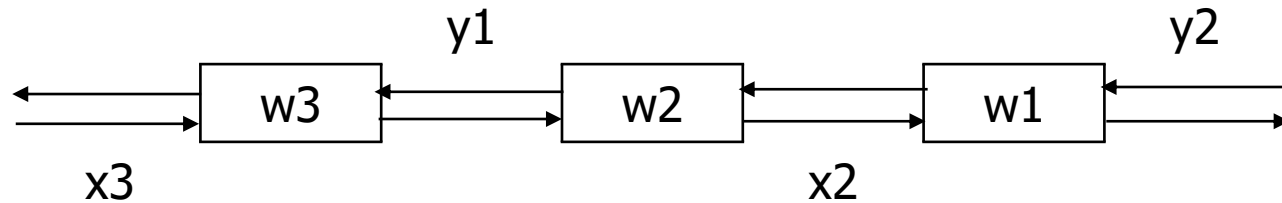
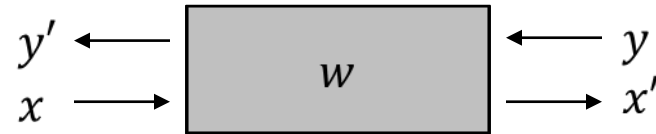
$$\begin{aligned} y' &= y + w \cdot x \\ x' &= x \end{aligned}$$



$$\begin{aligned} y_1 &= w_1 x_1 \\ y_2 &= 0 \end{aligned}$$

Systolic Computation: Convolution

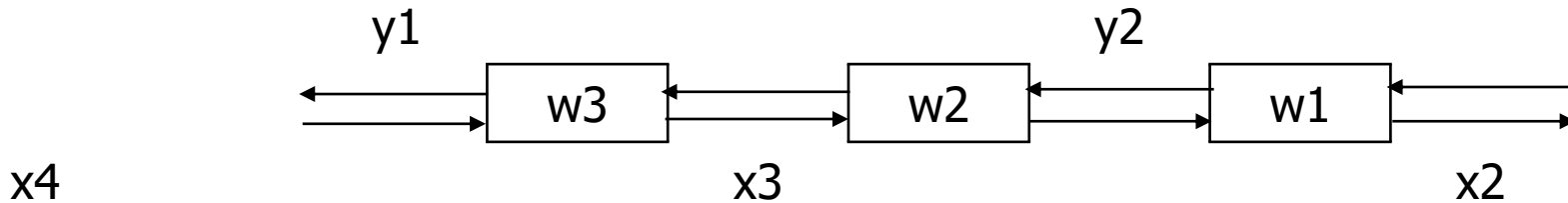
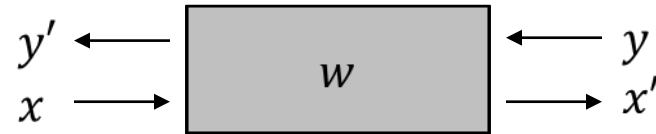
$$\begin{aligned}y' &= y + w \cdot x \\x' &= x\end{aligned}$$



$$\begin{aligned}y_1 &= w_1 x_1 + w_2 x_2 \\y_2 &= 0\end{aligned}$$

Systolic Computation: Convolution

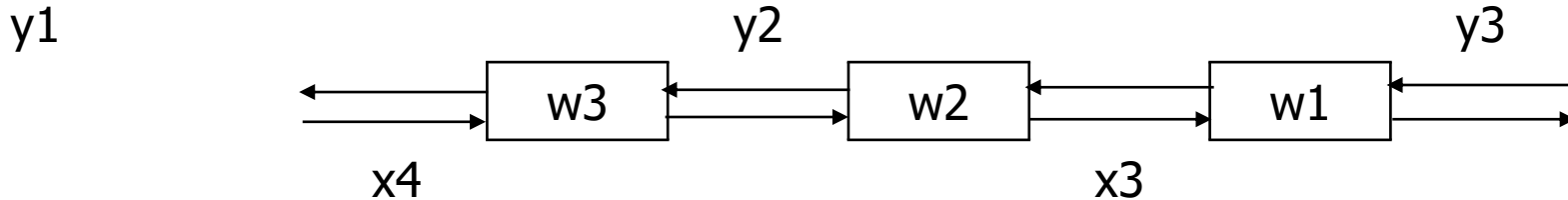
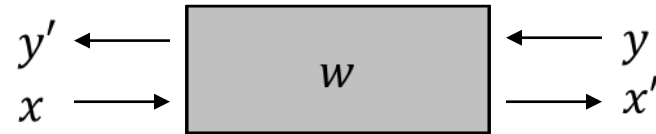
$$\begin{aligned} y' &= y + w \cdot x \\ x' &= x \end{aligned}$$



$$\begin{aligned} y_1 &= w_1 x_1 + w_2 x_2 + w_3 x_3 \\ y_2 &= w_1 x_2 \end{aligned}$$

Systolic Computation: Convolution

$$y' = y + w \cdot x$$
$$x' = x$$



$$y_1 = w_1 x_1 + w_2 x_2 + w_3 x_3$$
$$y_2 = w_1 x_2 + w_2 x_3$$

Systolic Computation Example: Convolution

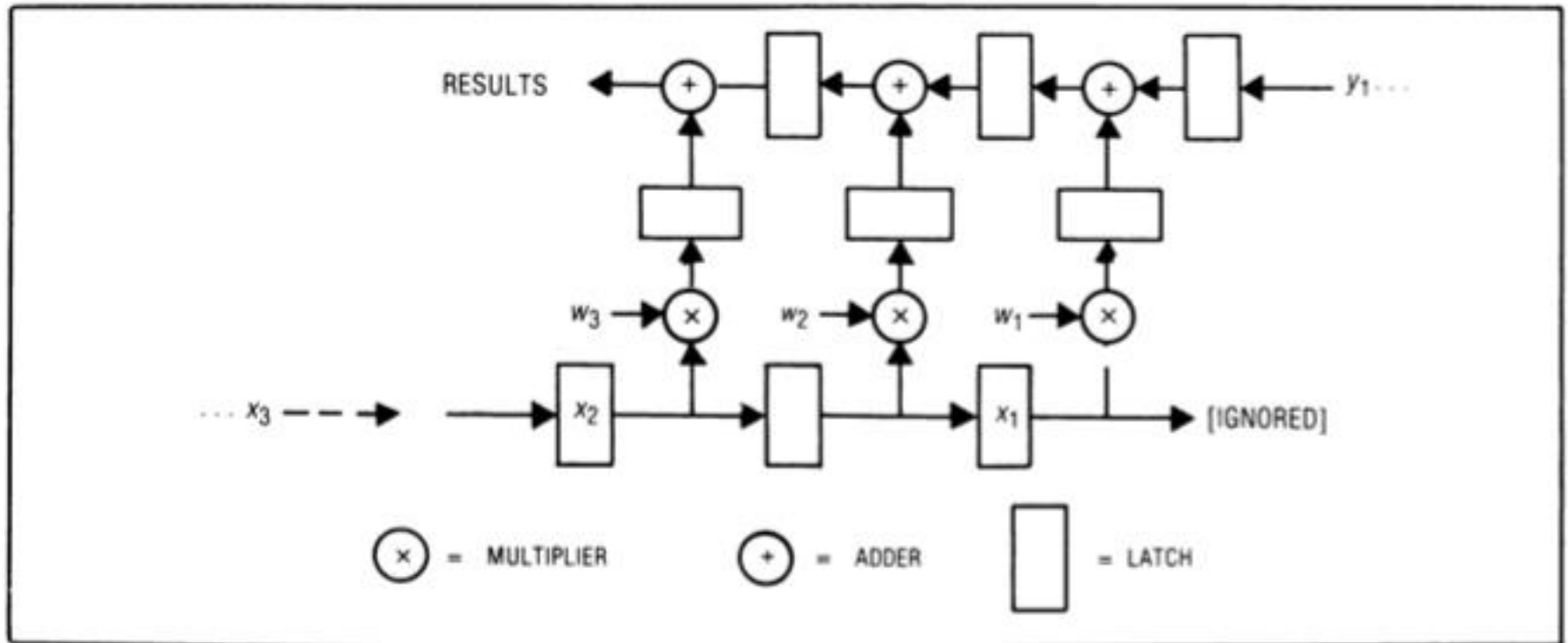


Figure 10. Overlapping the executions of multiply and add in design W1.

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

TODO: Example relating SP to systolic
architecture for some computation (maybe
the convolution)

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

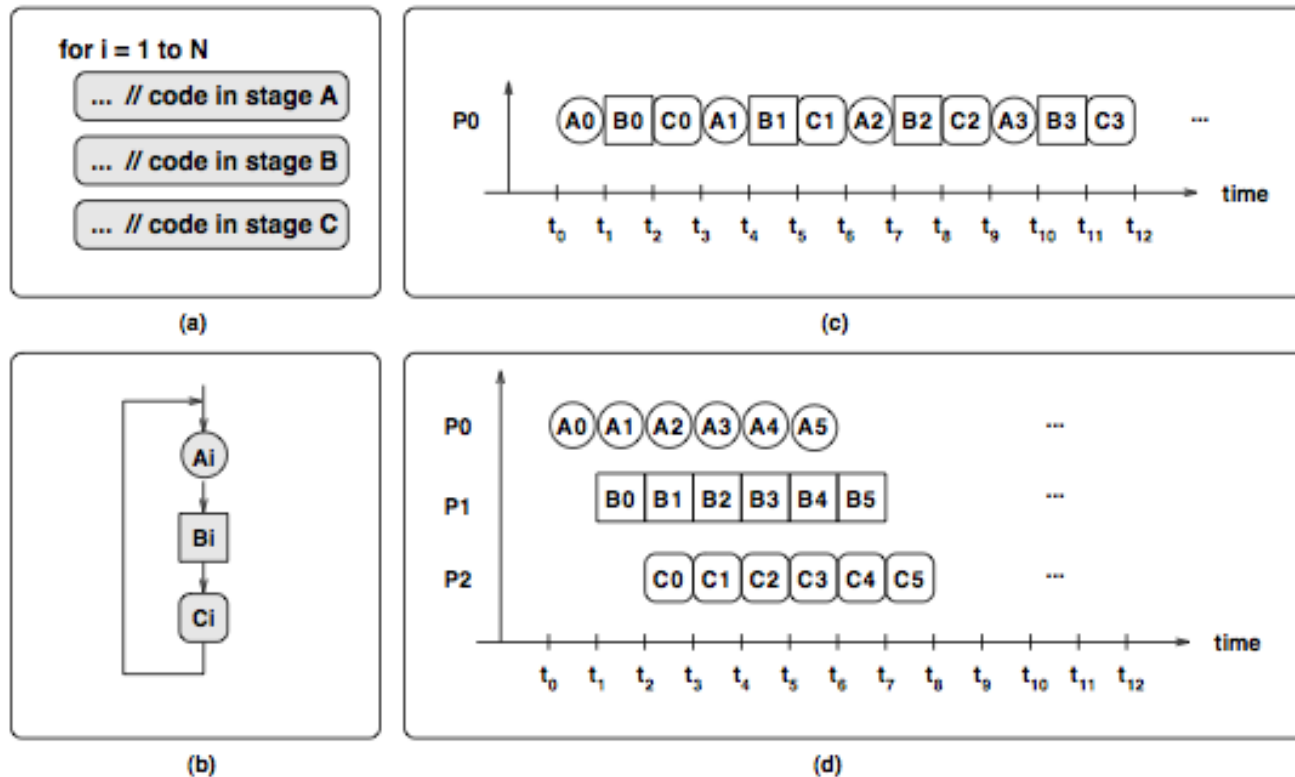


Figure 1. (a) The code of a loop, (b) Each iteration is split into 3 pipeline stages: A, B, and C. Iteration i comprises A_i , B_i , C_i . (c) Sequential execution of 4 iterations. (d) Parallel execution of 6 iterations using pipeline parallelism on a three-core machine. Each stage executes on one core.

File Compression Example

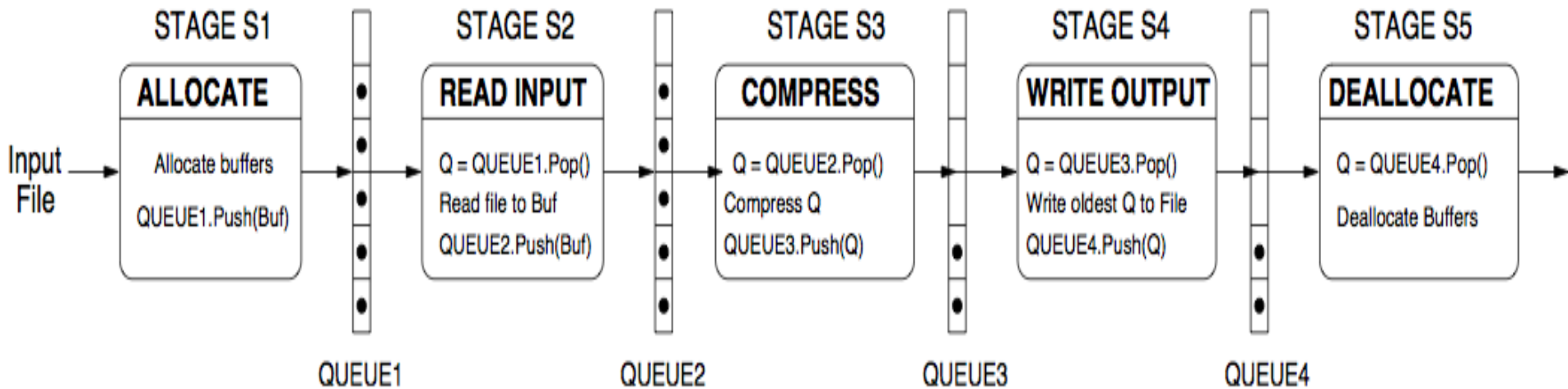


Figure 3. File compression algorithm executed using pipeline parallelism

Why pipeline parallelism in software?

- Pipeline parallelism vs data parallelism
 - *Why split pipeline stages across PEs?*
 - No cycle-time benefit like we got in hardware
- Data movement patterns differ
 - Pipeline parallelism: move input data between PEs
 - Data parallelism: move task code/data between PEs
- Tight feedback loops within single stage
 - E.g., compression or encryption
- Appropriate design depends on application

Systolic Array Summary

■ Advantages

- ❑ Makes multiple uses of each data item → reduce data fetches
- ❑ 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
- High-level language 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

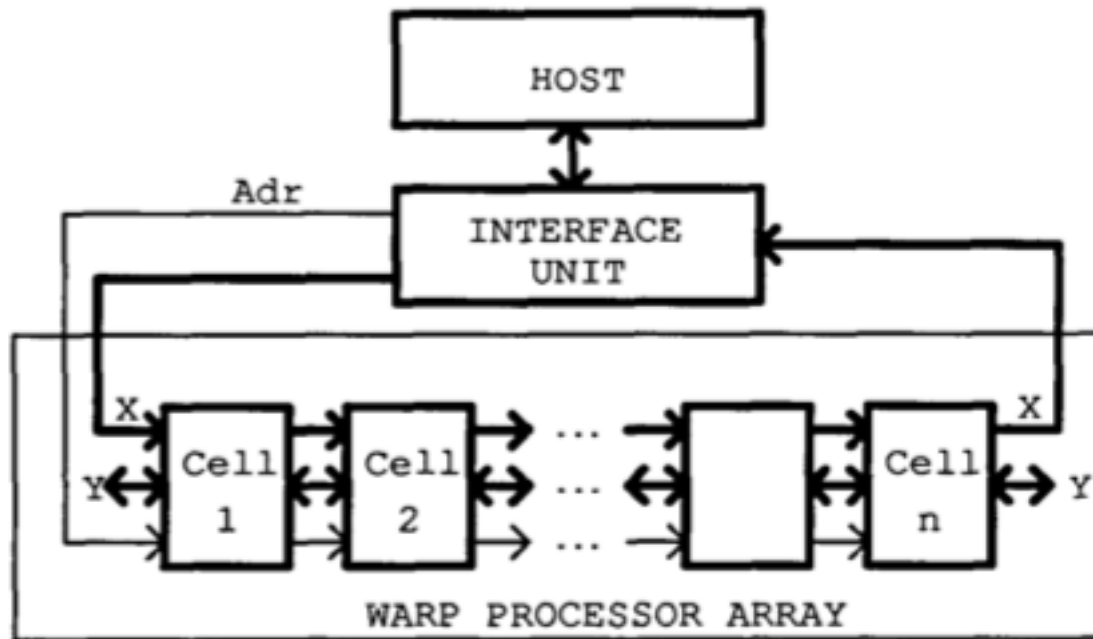
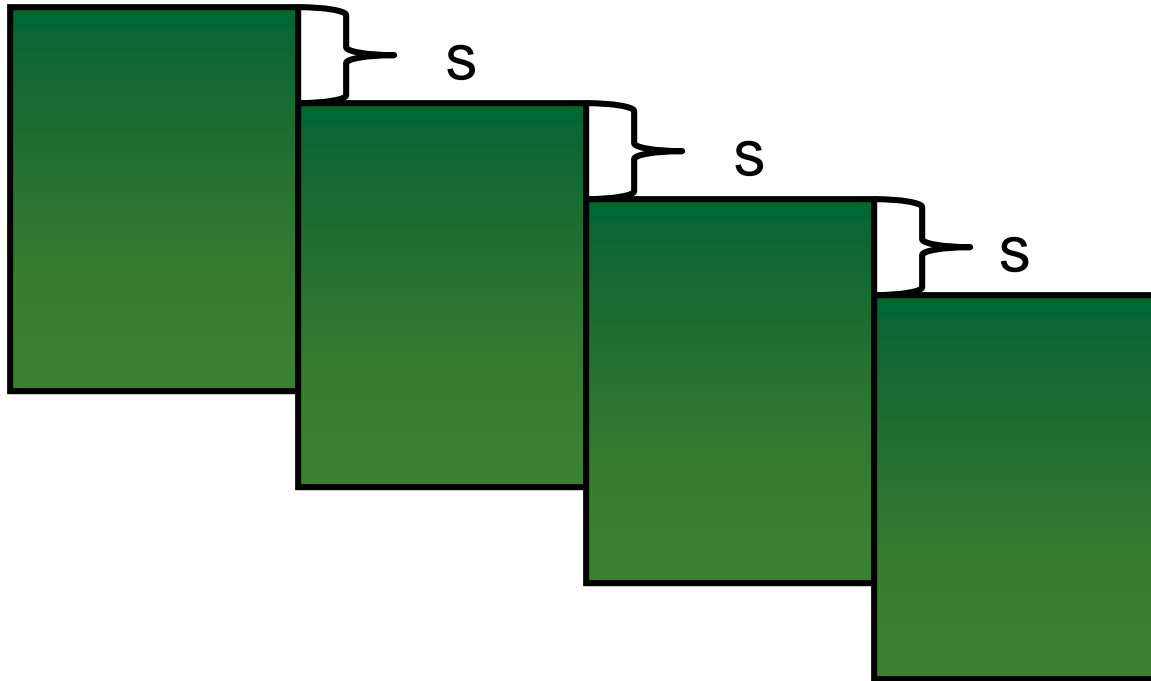


Figure 1: Warp system overview

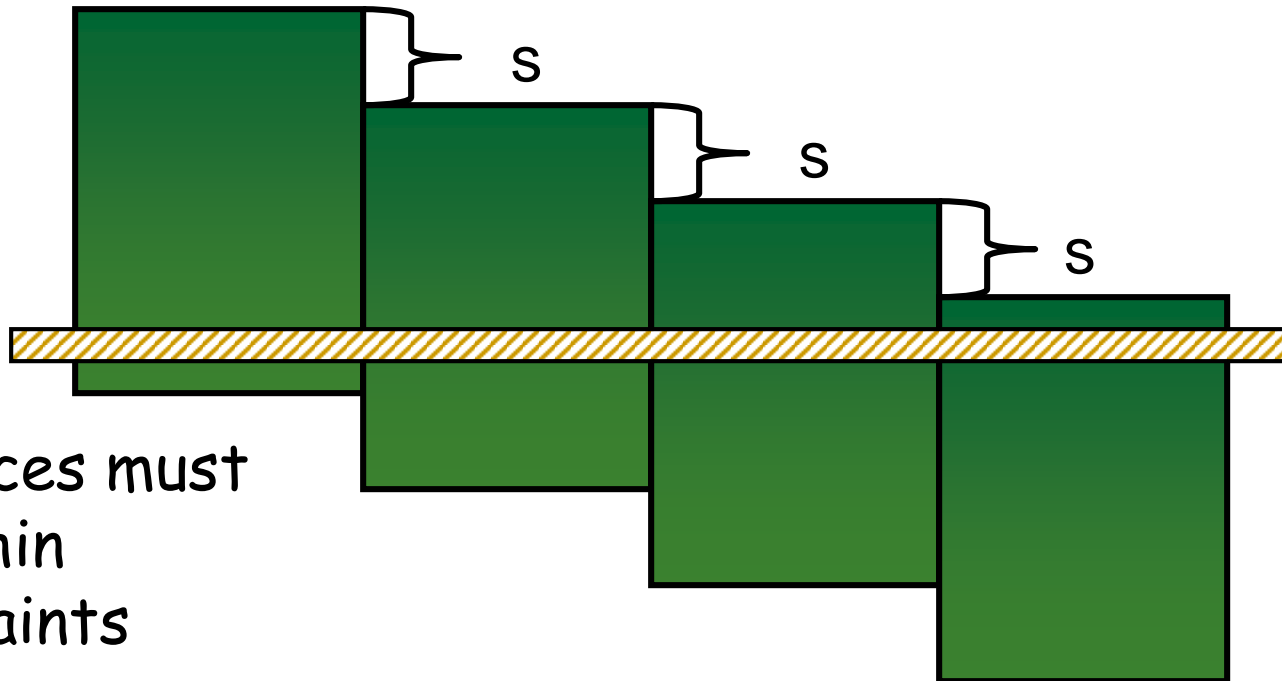
Software Pipelining Goal

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



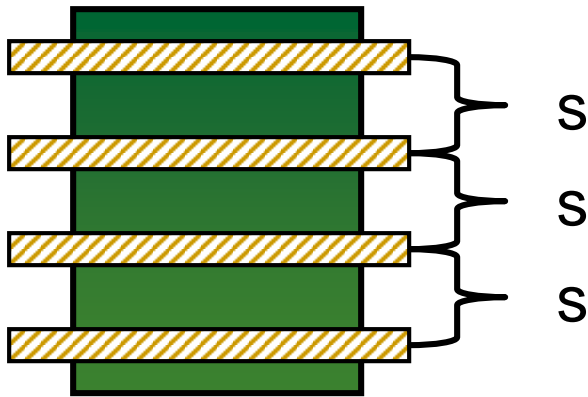
Software Pipelining Goal

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



Software Pipelining Goal

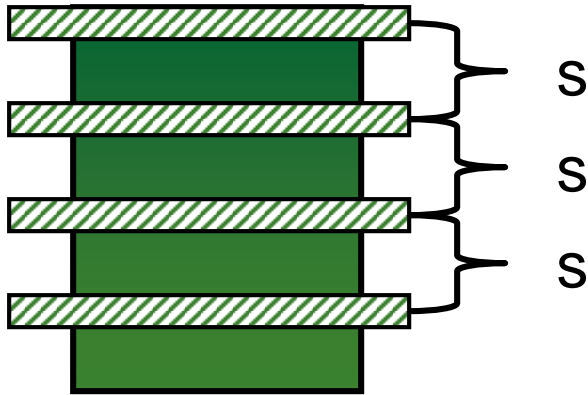
- Find the same schedule for each iteration.
- Stagger by iteration initiation interval, s
- Goal: minimize s .



resources must
be within
constraints

Software Pipelining Goal

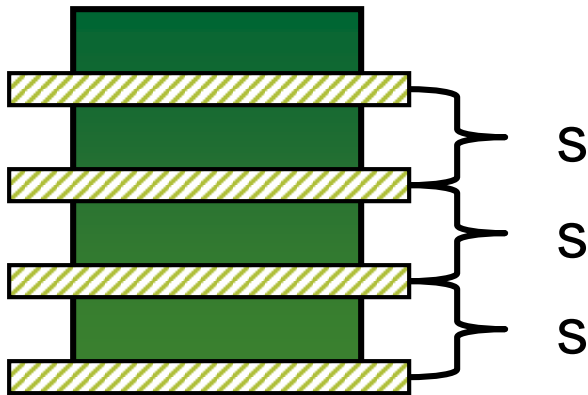
- Find the same schedule for each iteration.
- Stagger by iteration initiation interval, s
- Goal: minimize s .



resources must
be within
constraints

Software Pipelining Goal

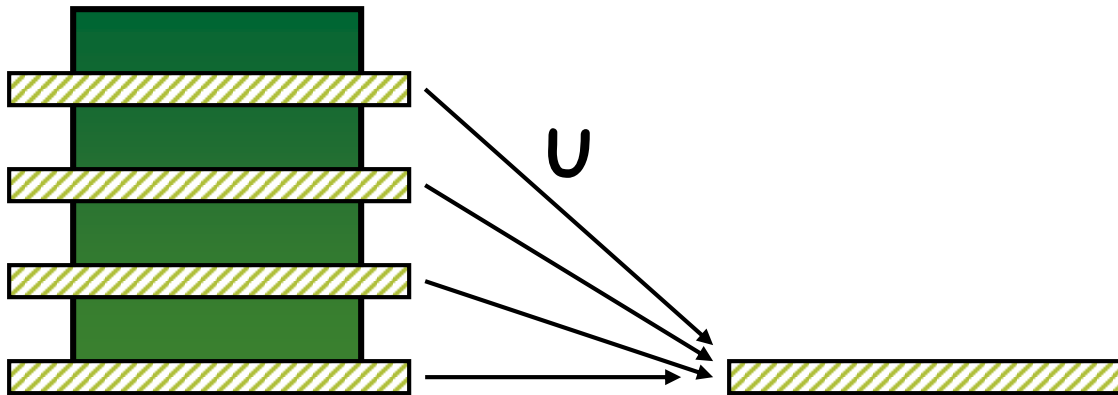
- Find the same schedule for each iteration.
- Stagger by iteration initiation interval, s
- Goal: minimize s .



resources must
be within
constraints

Software Pipelining Goal

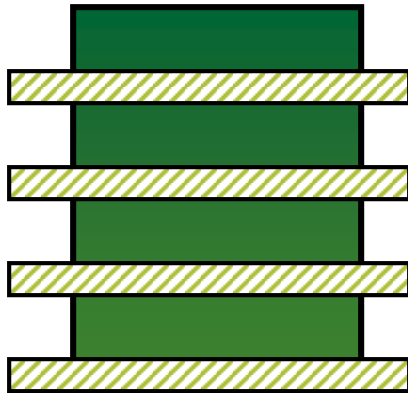
- Find the same schedule for each iteration.
- Stagger by iteration initiation interval, s
- Goal: minimize s .



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 :
 $\langle d(u,v) \rangle$
- For an edge, $u \rightarrow v$, we must have
 $\sigma(v) - \sigma(u) \geq d(u,v)$

Precedence Constraints

- Cyclic: constraint becomes a tuple: $\langle p, d \rangle$
 - 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 MICRO’94

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
maximum over all resources of # of uses divided by # available...
- Precedence Constraints: S_E
max delay over all cycles in dataflow graph

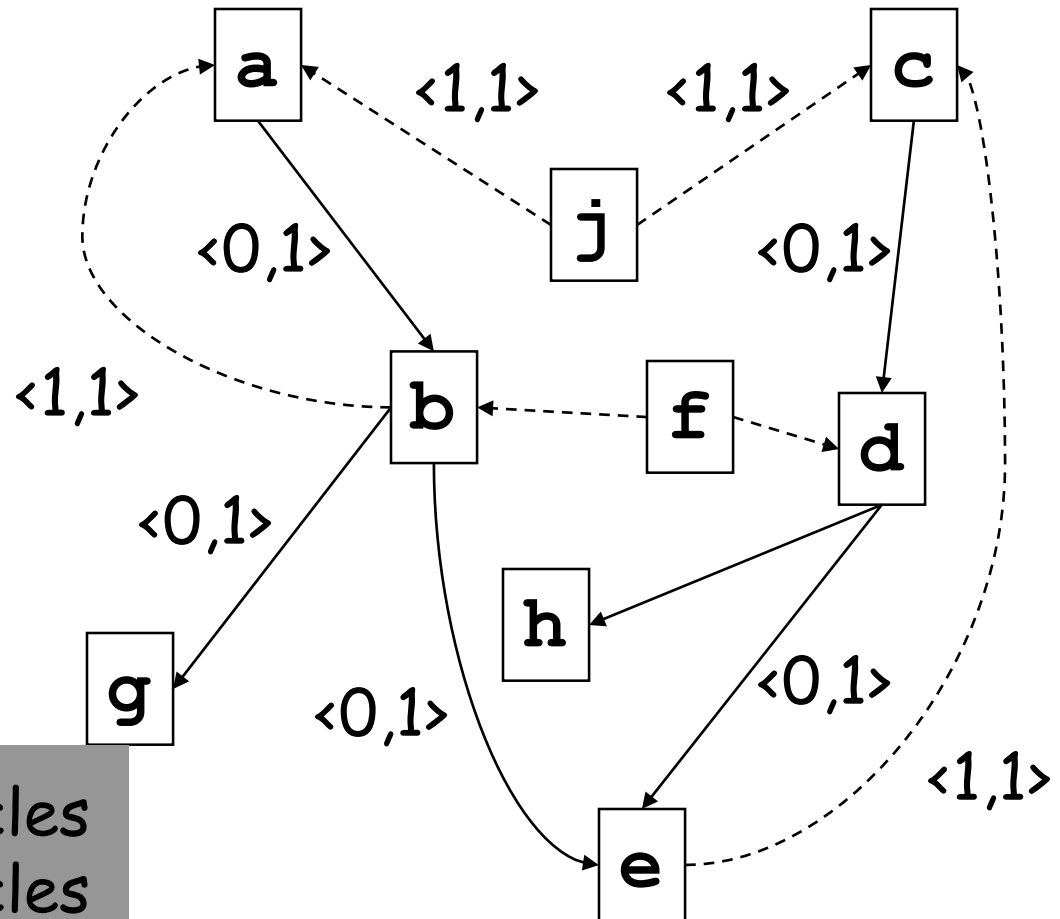
In practice, one is easy, other is hard.

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 \Rightarrow 5 cycles
Dependencies \Rightarrow 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

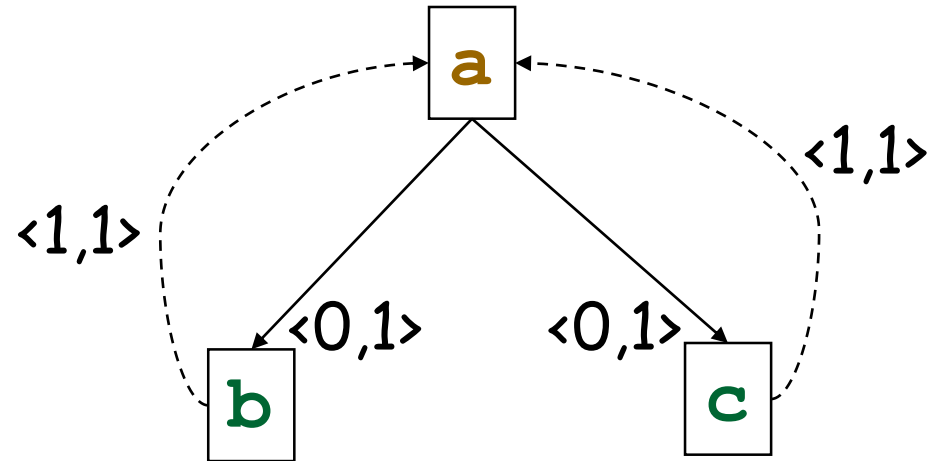
- Pick an instruction, n
- Calculate earliest time due to dependence constraints
For all $x = \text{pred}(n)$,
$$\text{earliest} = \max(\text{earliest}, \sigma(x) + d(x, n) - s \cdot p(x, n))$$
- try and schedule n from earliest to $(\text{earliest} + s - 1)$
s.t. resource constraints are obeyed.
 - 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., $\sigma(n) = \text{earliest}$
- Fix up schedule
 - Successors, x , of n must be scheduled s.t.
 $\sigma(x) \geq \sigma(n) + d(n,x) - s \cdot p(n,x)$, 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  
}
```



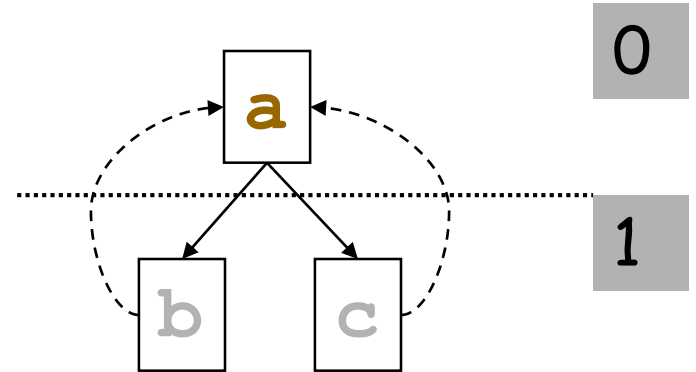
Resources: $\boxed{1}$ $\boxed{1}$

What is IIres?
What is IIrec?

Simplest Example

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

Try II = 2



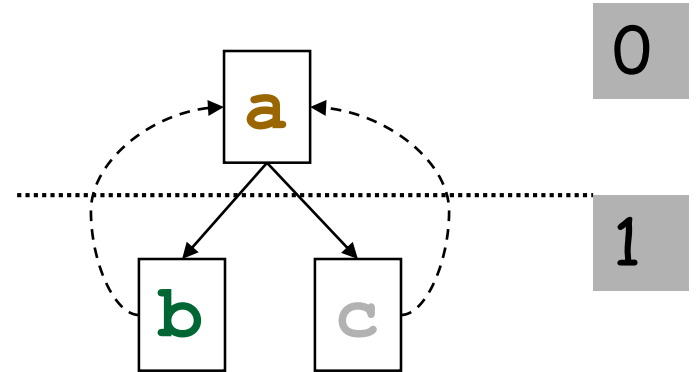
Modulo Resource Table:

0	1	
1		

Simplest Example

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

Try II = 2



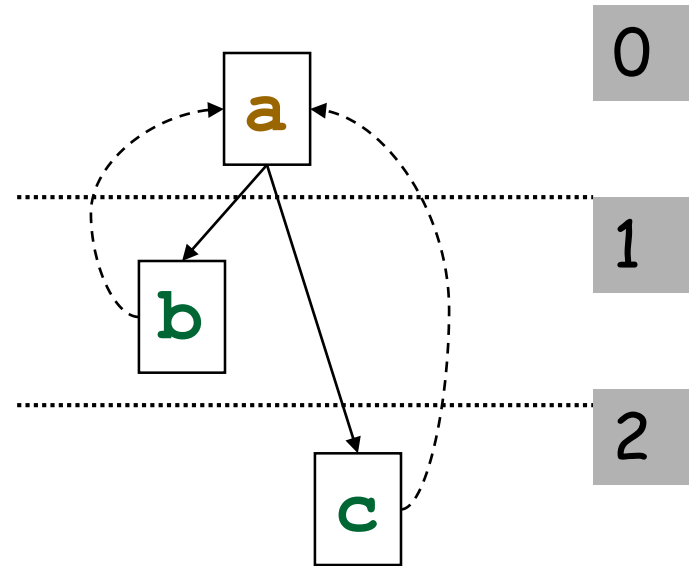
Modulo Resource Table:

0	1	
1		1

Simplest Example

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

Try II = 2



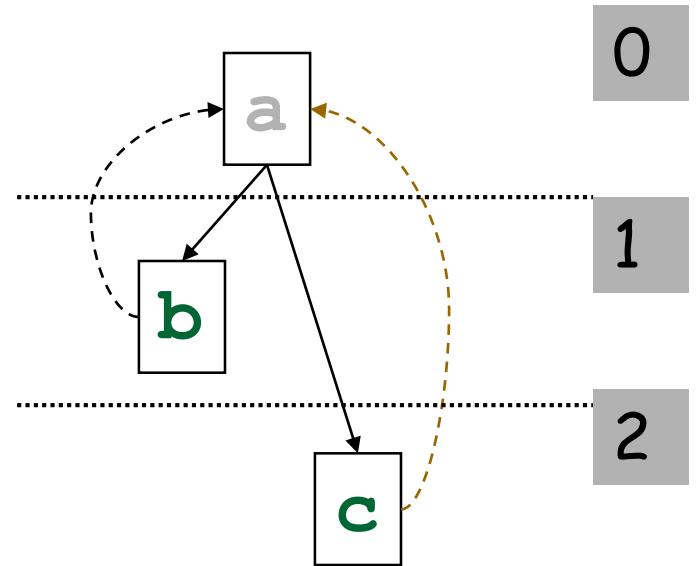
Modulo Resource Table:

0	1	1
1		1

Simplest Example

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

Try II = 2



Modulo Resource Table:

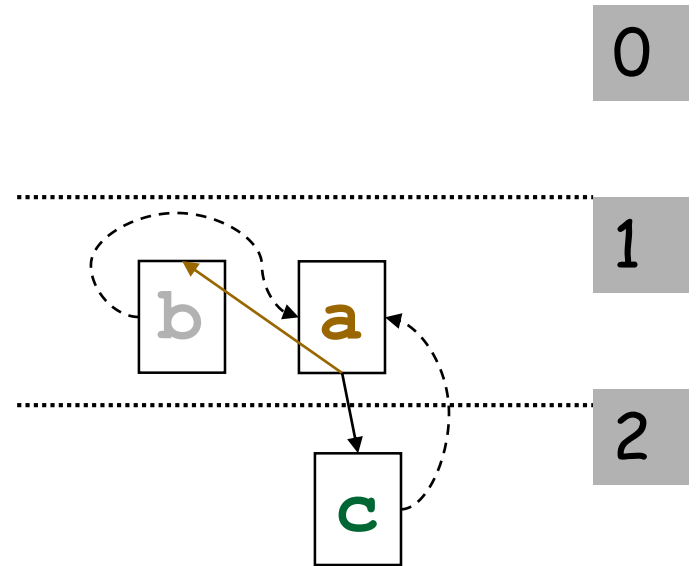
0		1
1		1

$$\begin{aligned} \text{earliest a: } & \sigma(c) + \text{delay}(c) - 2 \\ & = 2 + 1 - 2 = 1 \end{aligned}$$

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?

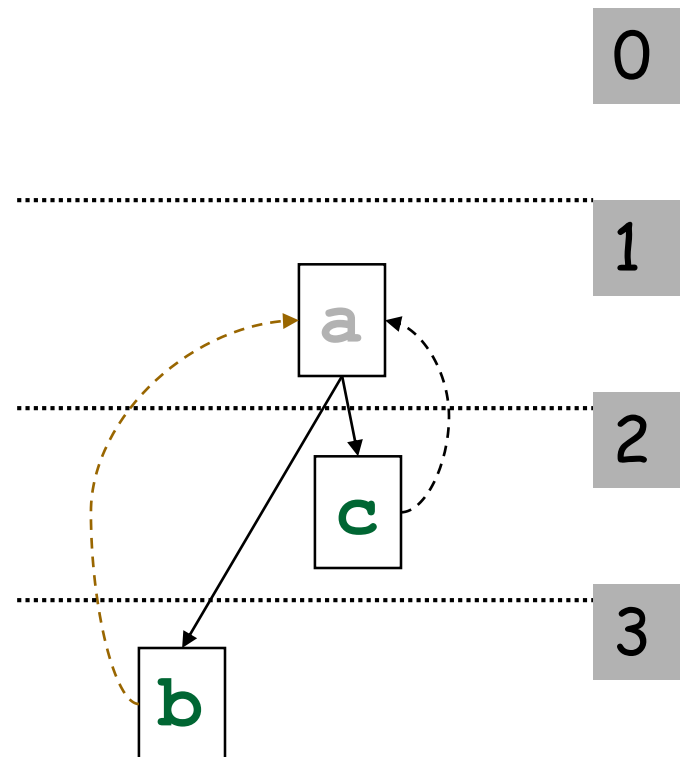
Simplest Example

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

Try II = 2

Modulo Resource Table:

0		1
1		1

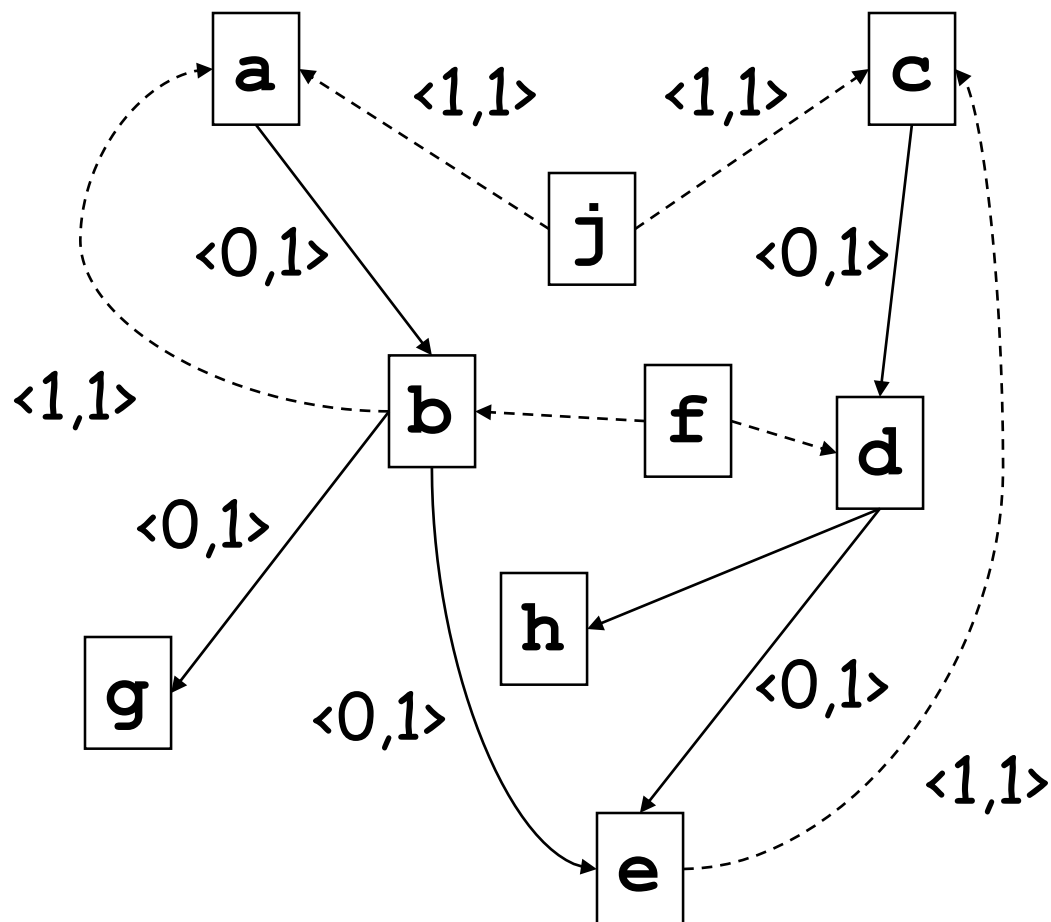


Lesson: lower bound
may not be achievable

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]
```

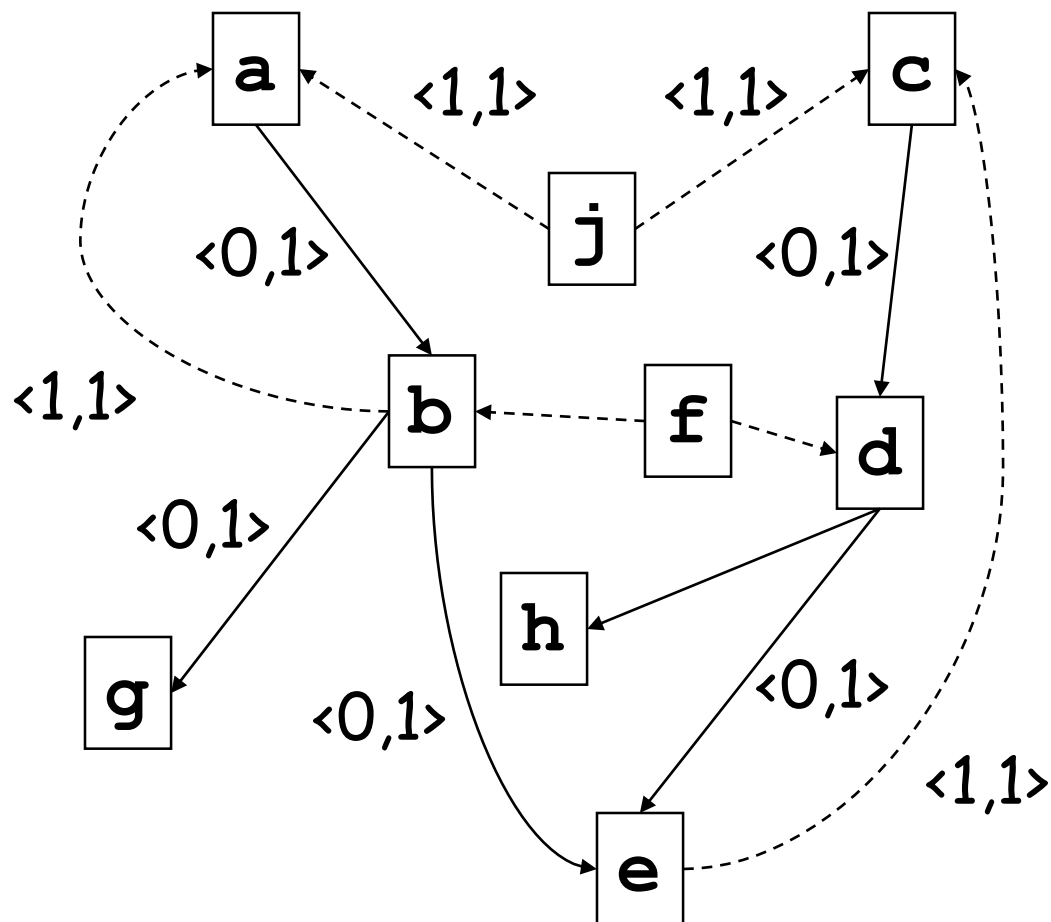
Priorities: ?



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  $\oplus$  b
```

```
  b := a  $\oplus$  f
```

```
  c := e  $\oplus$  j
```

```
  d := f  $\oplus$  c
```

```
  e := b  $\oplus$  d
```

```
  f := U[i]
```

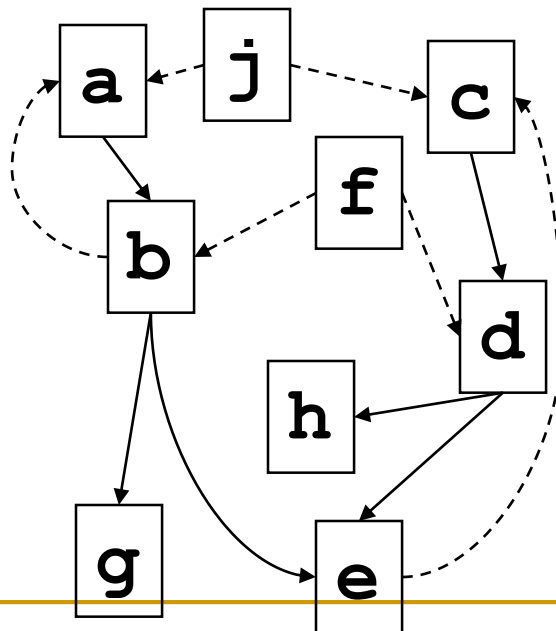
```
g: V[i] := b
```

```
h: W[i] := d
```

```
  j := X[i]
```

s=5

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



ALU	MU

instr	σ
a	
b	
c	
d	
e	
f	
g	
h	
j	

```
for i:=1 to N do
```

```
  a := j ⊕ b
```

```
  b := a ⊕ f
```

```
  c := e ⊕ j
```

```
  d := f ⊕ c
```

```
  e := b ⊕ d
```

```
  f := U[i]
```

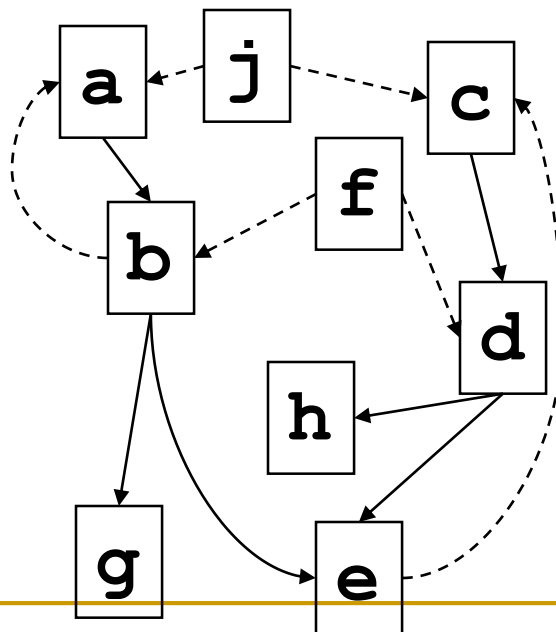
```
g: V[i] := b
```

```
h: W[i] := d
```

```
  j := X[i]
```

s=5

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



ALU	MU
c	
d	
e	

instr	σ
a	
b	
c	0
d	1
e	2
f	
g	
h	
j	

```
for i:=1 to N do
```

```
  a := j ⊕ b
```

```
  b := a ⊕ f
```

```
  c := e ⊕ j
```

```
  d := f ⊕ c
```

```
  e := b ⊕ d
```

```
  f := U[i]
```

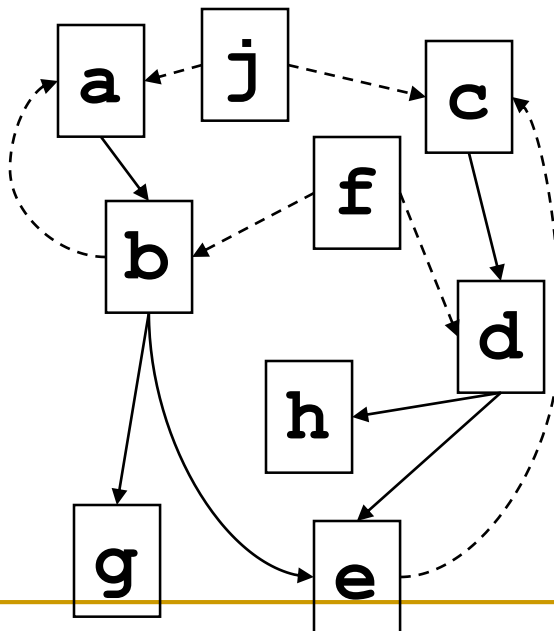
```
g: V[i] := b
```

```
h: W[i] := d
```

```
  j := x[i]
```

s=5

Priorities: b,f,j,g,h



ALU	MU
c	
d	
e	
a	

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

```
for i:=1 to N do
```

```
  a := j ⊕ b
```

```
  b := a ⊕ f
```

```
  c := e ⊕ j
```

```
  d := f ⊕ c
```

```
  e := b ⊕ d
```

```
  f := U[i]
```

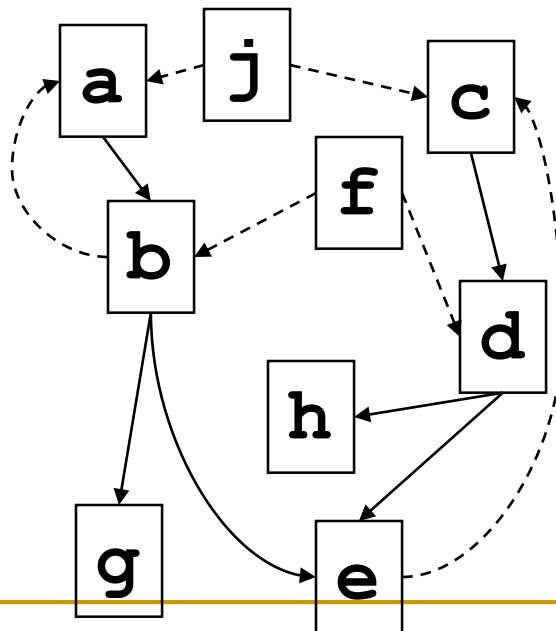
```
g: V[i] := b
```

```
h: W[i] := d
```

```
  j := x[i]
```

s=5

Priorities: b,f,j,g,h



ALU	MU
c	
d	
e	
a	
b	

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


```
for i:=1 to N do
```

```
  a := j ⊕ b
```

```
  b := a ⊕ f
```

```
  c := e ⊕ j
```

```
  d := f ⊕ c
```

```
  e := b ⊕ d
```

```
  f := U[i]
```

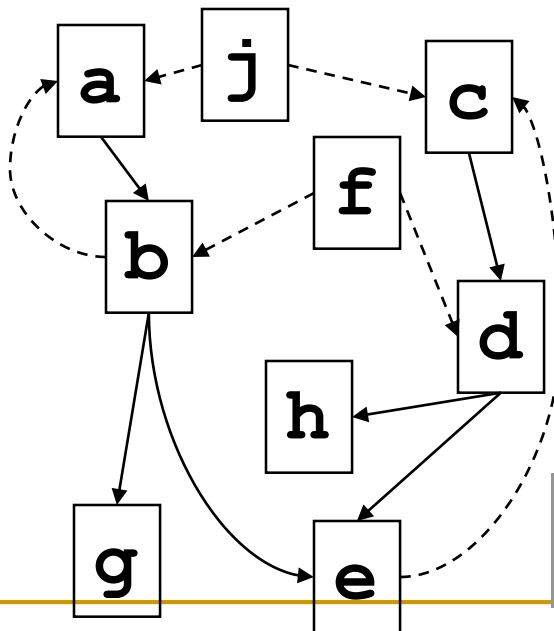
```
g: V[i] := b
```

```
h: W[i] := d
```

```
  j := X[i]
```

s=5

Priorities: e,f,j,g,h



ALU	MU
c	
d	
a	
b	

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

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]
```

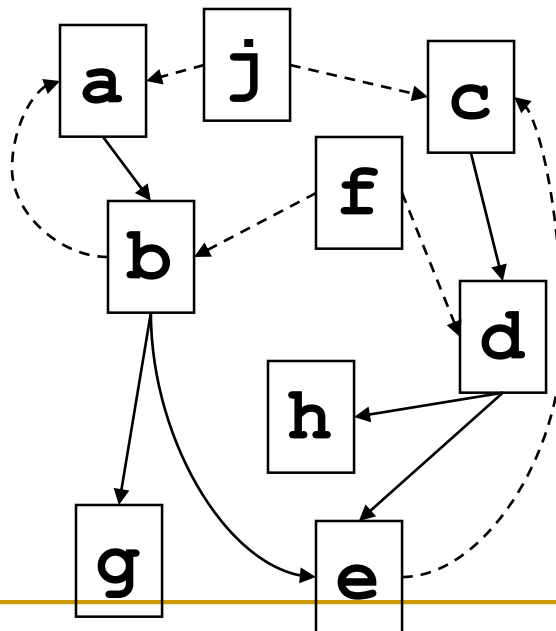
```
g: V[i] := b
```

```
h: W[i] := d
```

```
  j := X[i]
```

s=5

Priorities: e,f,j,g,h



ALU	MU
c	
d	
e	
a	
b	

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

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]
```

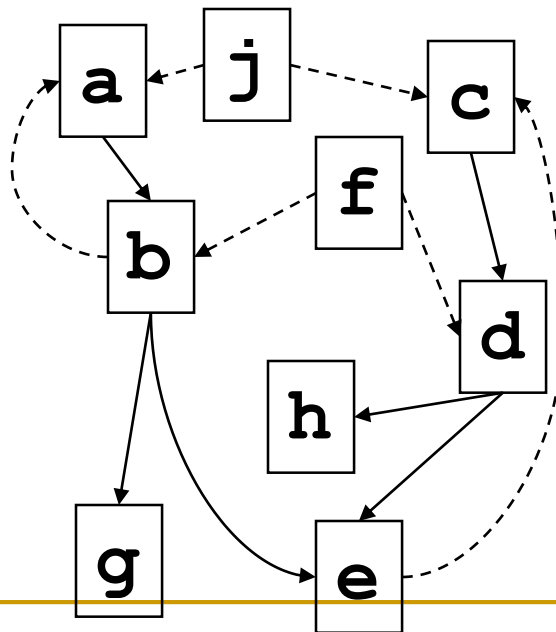
```
g: V[i] := b
```

```
h: W[i] := d
```

```
  j := X[i]
```

s=5

Priorities: f,j,g,h



ALU	MU
c	f
d	
e	
a	
b	

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

```
for i:=1 to N do
```

```
  a := j ⊕ b
```

```
  b := a ⊕ f
```

```
  c := e ⊕ j
```

```
  d := f ⊕ c
```

```
  e := b ⊕ d
```

```
  f := U[i]
```

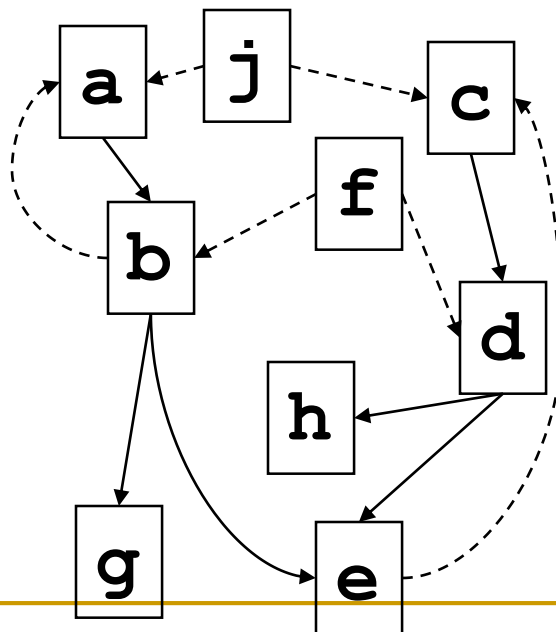
```
g: V[i] := b
```

```
h: W[i] := d
```

```
  j := X[i]
```

s=5

Priorities: j,g,h



ALU	MU
c	f
d	j
e	
a	
b	

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

```
for i:=1 to N do
```

```
  a := j ⊕ b
```

```
  b := a ⊕ f
```

```
  c := e ⊕ j
```

```
  d := f ⊕ c
```

```
  e := b ⊕ d
```

```
  f := U[i]
```

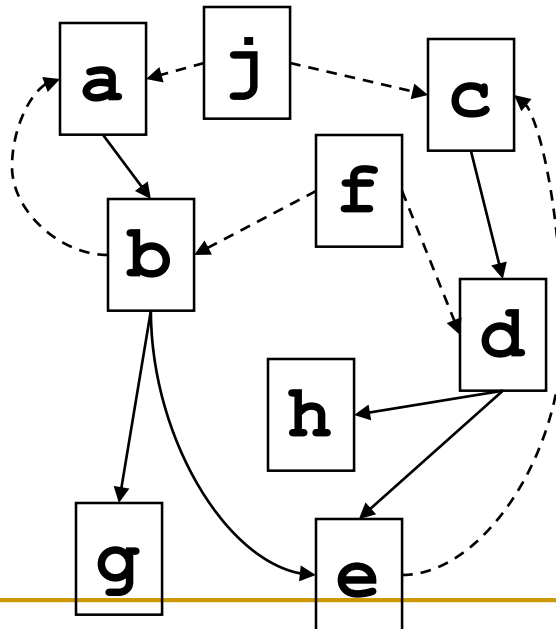
```
g: V[i] := b
```

```
h: W[i] := d
```

```
  j := X[i]
```

s=5

Priorities:g,h



ALU	MU
c	f
d	j
e	g
a	h
b	

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

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

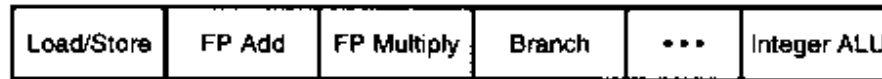
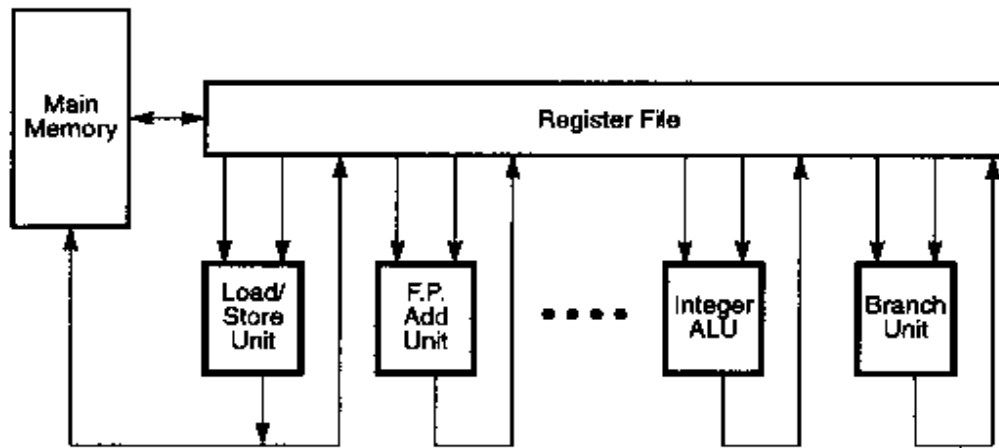
 j0 = j1

Conditionals

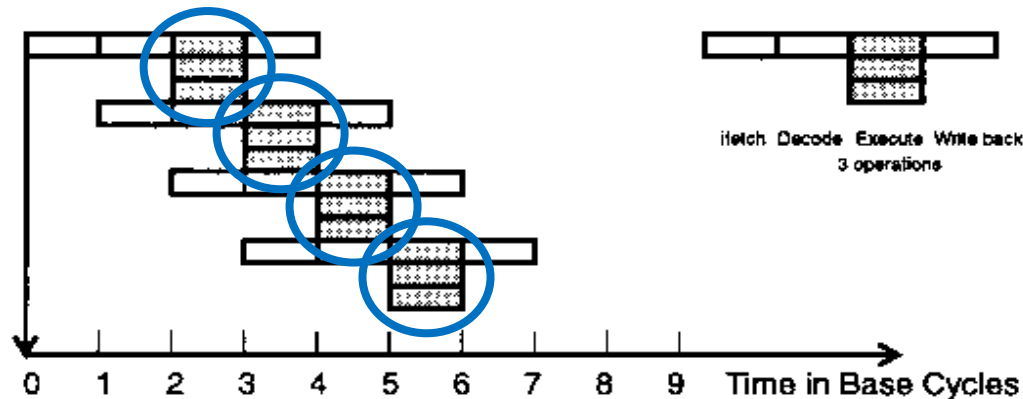
- What about internal control structure, I.e., conditionals
- Three approaches
 - Schedule both sides and use conditional moves
 - Schedule each side, then make the body of the conditional a macro op with appropriate resource vector
 - Trace schedule the loop

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



(a) A typical VLIW processor and instruction format



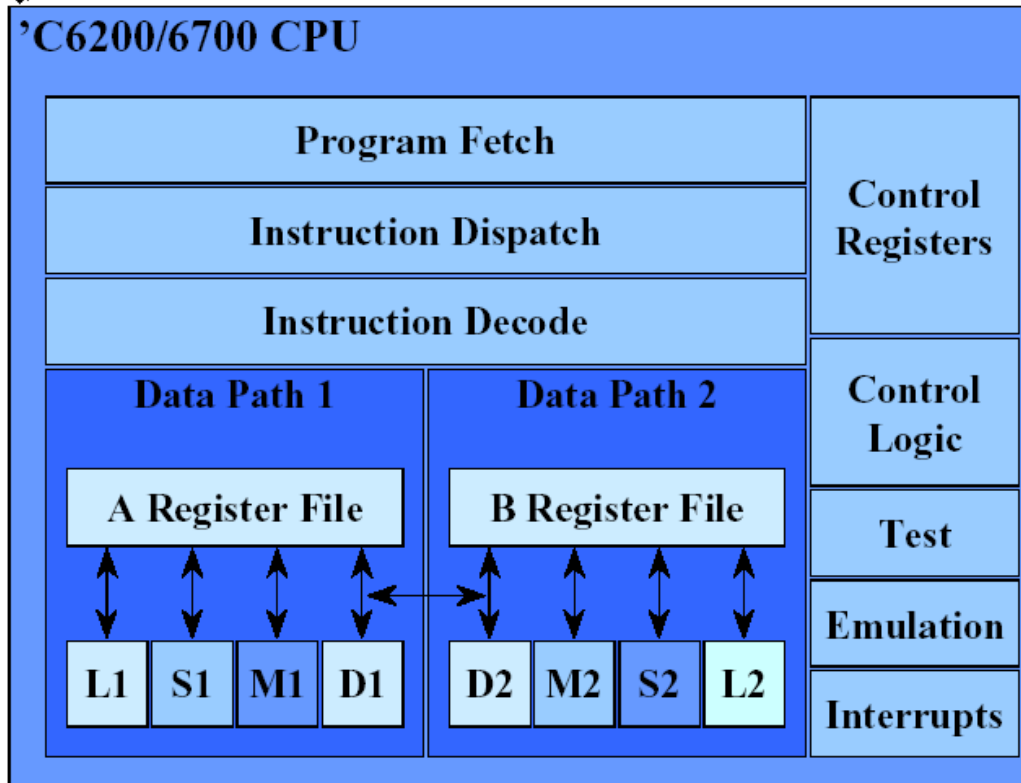
(b) VLIW execution with degree $m = 3$

Multiflow:
An early VLIW
architecture
(1987)

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

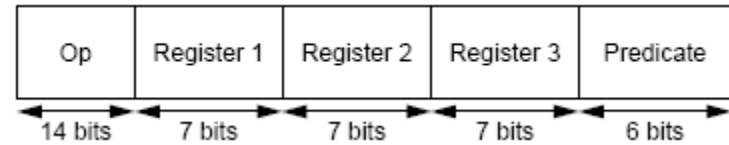
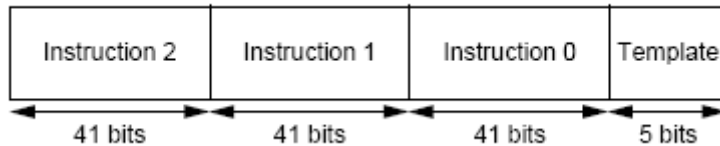
- 4 ns cycle time
- 2000 MIPS @ 250 MHz
- 500 MMACS (Mega MACs per Second)

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

```
{ .mi1
  add r1 = r2, r3
  sub r4 = r4, r5 ;;
  shr r7 = r4, r12 ;;
}
{ .mm1
  ld8 r2 = [r1] ;;
  st8 [r1] = r23
  tbit p1,p2=r4,5
}
{ .mbb
  ld8 r45 = [r55]
  (p3)br.call b1=func1
  (p4)br.cond Label1
}
{ .mfi
  st4 [r45]=r6
  fmac f1=f2,f3
  add r3=r3,8 ;;
}
```



- Groups of instructions can be executed safely in parallel
 - Marked by “stop bits”
- Bundles are for packaging
 - Groups can span multiple bundles
 - Alleviates recompilation need somewhat