

Instruction Scheduling

Last time

- Register allocation

Today

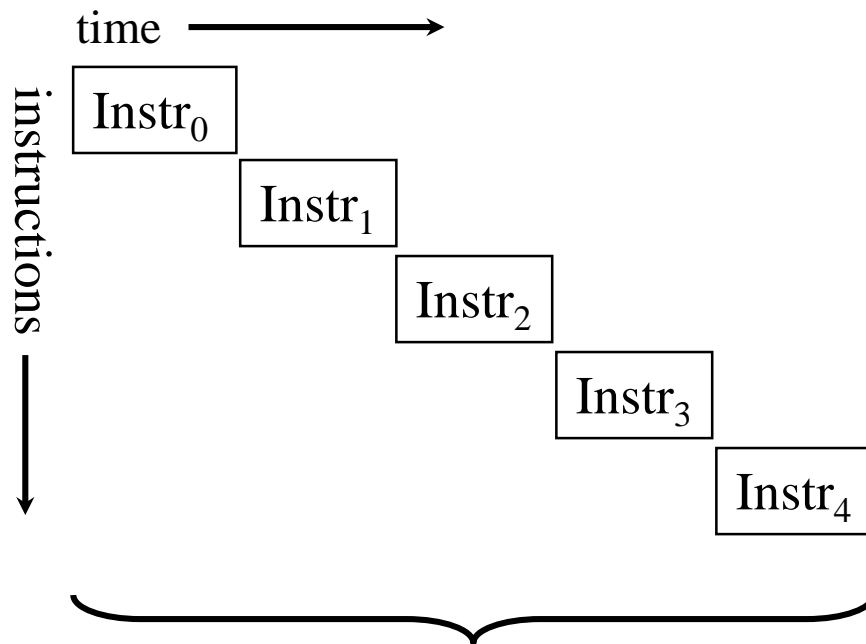
- Instruction scheduling
 - The problem: Pipelined computer architecture
 - A solution: List scheduling
 - Improvements on this solution

Background: Pipelining Basics

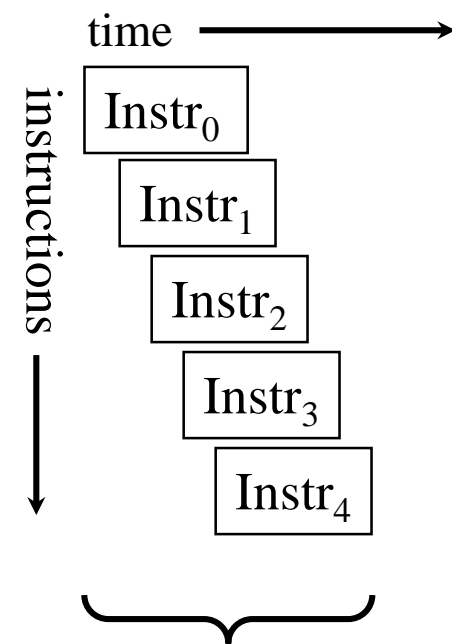
Idea

- Begin executing an instruction **before** completing the previous one

Without Pipelining



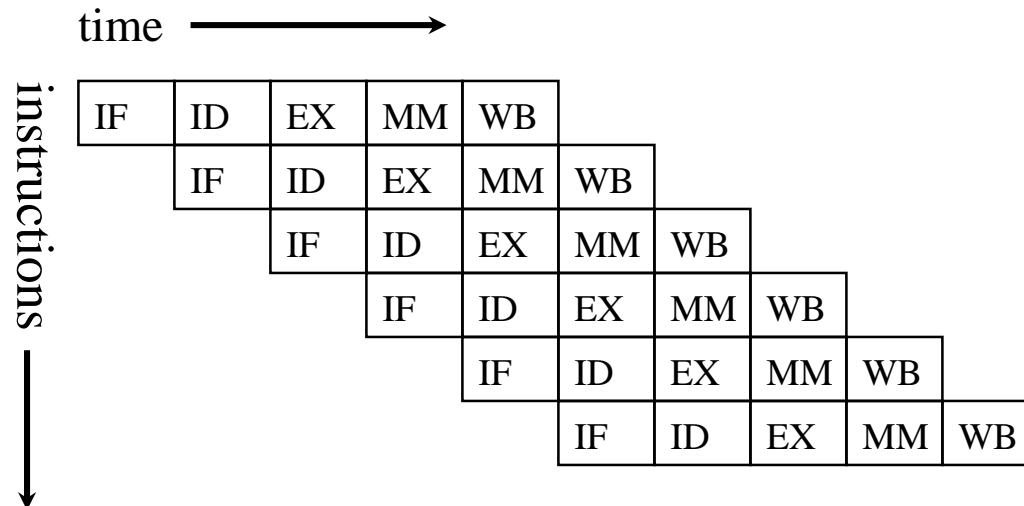
With Pipelining



Idealized Instruction Data-Path

Instructions go through several stages of execution

Stage 1		Stage 2		Stage 3		Stage 4		Stage 5
Instruction Fetch	⇒	Instruction Decode & Register Fetch	⇒	Execute	⇒	Memory Access	⇒	Register Write-back
IF	⇒	ID/RF	⇒	EX	⇒	MEM	⇒	WB



Pipelining Details

Observations

- Individual instructions are no faster (but throughput is higher)
- Potential speedup determined by number of stages (more or less)
- Filling and draining pipe limits speedup
- Rate through pipe is limited by slowest stage
- Less work per stage implies faster clock

Modern Processors

- Long pipelines: 5 (Pentium), 14 (Pentium Pro), 22 (Pentium 4), 31 (Prescott), 14 (Core i7), 8 ARM 11
- Issue width: 2 (Pentium), 4 (UltraSPARC) or more (dead Compaq EV8)
- Dynamically schedule instructions (from limited instruction window) or statically schedule (*e.g.*, IA-64)
- Speculate
 - Outcome of branches
 - Value of loads (research)

What Limits Performance?

Data hazards

- Instruction depends on result of prior instruction that is still in the pipe

Structural hazards

- Hardware cannot support certain instruction sequences because of limited hardware resources

Control hazards

- Control flow depends on the result of branch instruction that is still in the pipe

An obvious solution

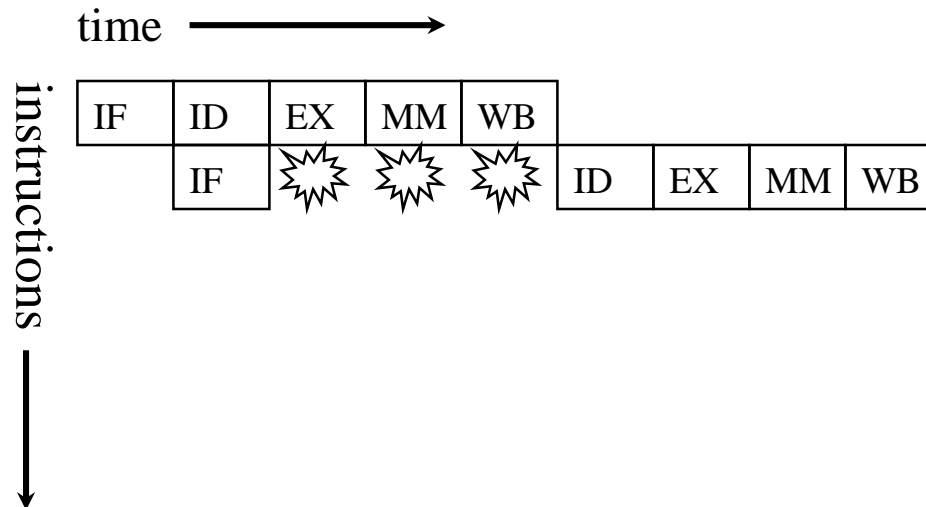
- Stall (insert bubbles into pipeline)

Stalls (Data Hazards)

Code

```
add $r1, $r2, $r3    // $r1 is the destination
mul $r4, $r1, $r1    // $r4 is the destination
```

Pipeline picture

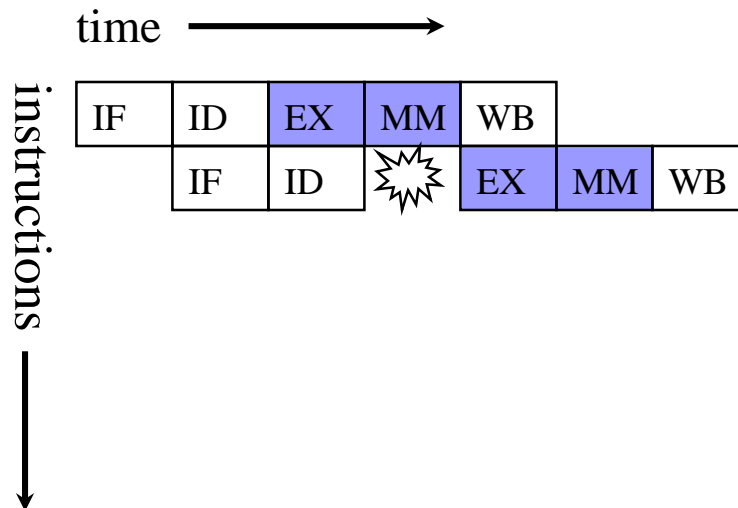


Stalls (Structural Hazards)

Code

```
mul $r1, $r2, $r3    // Suppose multiplies take two cycles  
mul $r4, $r5, $r6
```

Pipeline Picture

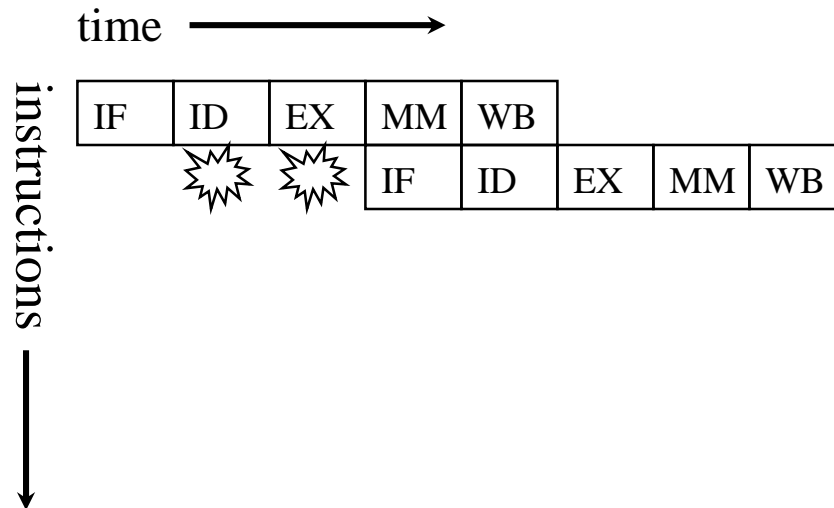


Stalls (Control Hazards)

Code

```
bz $r1, label // if $r1==0, branch to label  
add $r2,$r3,$r4
```

Pipeline Picture



Hardware Solutions

Data hazards

- Data forwarding (doesn't completely solve problem)
- Runtime speculation (doesn't always work)

Structural hazards

- Hardware replication (expensive)
- More pipelining (doesn't always work)

Control hazards

- Runtime speculation (branch prediction)

Dynamic scheduling

- Can address all of these issues
- Very successful

Context: The MIPS R2000

MIPS Computer Systems

- “First” commercial RISC processor (R2000 in 1984)
- Began trend of requiring nontrivial instruction scheduling by the compiler

What does MIPS mean?

- **M**icroprocessor without **I**nterlocked **P**ipeline **S**tages

Instruction Scheduling for Pipelined Architectures

Goal

- An efficient algorithm for reordering instructions to minimize pipeline stalls

Constraints

- Data dependences (for correctness)
- Hazards (can *only* have performance implications)

Simplifications

- Do scheduling after instruction selection and register allocation
- Only consider data hazards

Recall Data Dependences

Data dependence

- A data dependence is an ordering constraint on 2 statements
- When reordering statements, all data dependences must be observed to preserve program correctness

True (or flow) dependences

- Write to variable x followed by a read of x (read after write or RAW)

```
x = 5;  
print (x);
```

Anti-dependences

- Read of variable x followed by a write (WAR)

```
print (x);  
x = 5;
```

Output dependences

- Write to variable x followed by another write to x (WAW)

```
x = 6;  
x = 5;
```

} false
dependences

List Scheduling [Gibbons & Muchnick '86]

Scope

- Basic blocks

Assumptions

- Pipeline interlocks are provided (*i.e.*, algorithm need not introduce no-ops)
- Pointers can refer to any memory address (*i.e.*, no alias analysis)
- Hazards take a single cycle (stall); here let's assume there are two...
 - **Load** immediately followed by **ALU op** produces interlock
 - **Store** immediately followed by **load** produces interlock

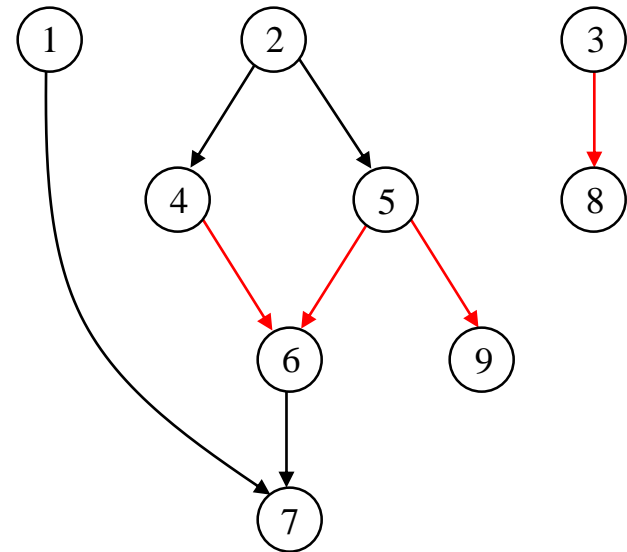
Main data structure: dependence DAG

- Nodes represent instructions
- Edges (s_1, s_2) represent dependences between instructions
 - Instruction s_1 must execute before s_2
- Sometimes called **data dependence graph** or **data-flow graph**

Dependence Graph Example

Sample code	dst	src	src
1 addi	\$r2	1	\$r1
2 addi	\$sp	12	\$sp
3 st	a	\$r0	
4 ld	\$r3	-4 (\$sp)	
5 ld	\$r4	-8 (\$sp)	
6 addi	\$sp	8	\$sp
7 st	0 (\$sp)	\$r2	
8 ld	\$r5	a	
9 addi	\$r4	1	\$r4

Dependence graph



Hazards in current schedule

– (3,4), (5,6), (7,8), (8,9)

Any topological sort is okay, but we want best one

Scheduling Heuristics

Goal

- Avoid stalls

What are some good heuristics?

- Does an instruction interlock with any immediate successors in the dependence graph?
- How many immediate successors does an instruction have?
- Is an instruction on the critical path?

Scheduling Heuristics (cont)

Idea: schedule an instruction earlier when...

- It does not interlock with the previously scheduled instruction (avoid stalls)
- It interlocks with its successors in the dependence graph (may enable successors to be scheduled without stall)
- It has many successors in the graph (may enable successors to be scheduled with greater flexibility)
- It is on the critical path (the goal is to minimize time, after all)

Scheduling Algorithm

Build dependence graph G

Candidates \leftarrow set of all roots (nodes with no in-edges) in G

while Candidates $\neq \emptyset$

 Select instruction s from Candidates {Using heuristics—in order}

 Schedule s

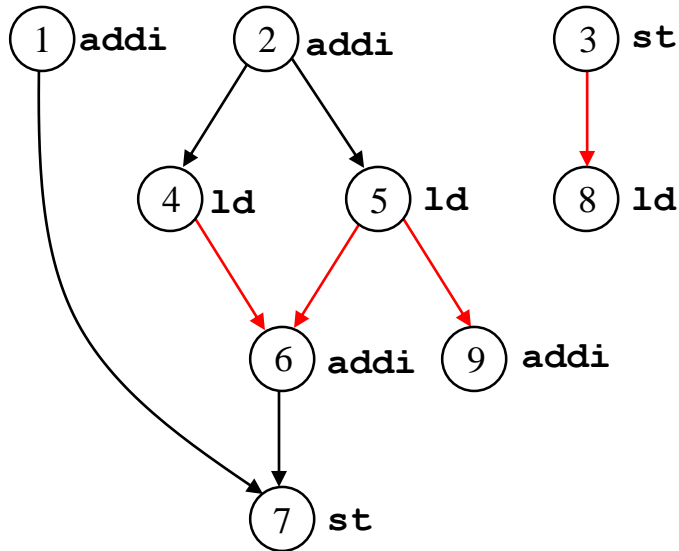
 Candidates \leftarrow Candidates $- s$

 Candidates \leftarrow Candidates \cup “exposed” nodes

{Add to Candidates those nodes whose
predecessors have all been scheduled}

Scheduling Example

Dependence Graph



Scheduled Code

```
3  st    a, $r0
2  addi  $sp, 12, $sp
5  ld    $r4, -8($sp)
4  ld    $r3, -4($sp)
8  ld    $r5, a
1  addi  $r2, 1, $r1
6  addi  $sp, 8, $sp
7  st    0($sp), $r2
9  addi  $r4, 1, $r4
```

Candidates

```
1  addi  $r2, 1, $r1
2  addi  $sp, 12, $sp
```

Hazards in new schedule

-(8,1)

Scheduling Example (cont)

Original code

1	addi	\$r2, 1, \$r1	3	st	a, \$r0
2	addi	\$sp, 12, \$sp	2	addi	\$sp, 12, \$sp
3	st	a, \$r0	5	ld	\$r4, -8(\$sp)
4	ld	\$r3, -4(\$sp)	4	ld	\$r3, -4(\$sp)
5	ld	\$r4, -8(\$sp)	8	ld	\$r5, a
6	addi	\$sp, 8, \$sp	1	addi	\$r2, 1, \$r1
7	st	0(\$sp), \$r2	6	addi	\$sp, 8, \$sp
8	ld	\$r5, a	7	st	0(\$sp), \$r2
9	addi	\$r4, 1, \$r4	9	addi	\$r4, 1, \$r4

Hazards in original schedule

–(3,4), (5,6), (7,8), (8,9)

Hazards in new schedule

–(8,1)

Complexity

Quadratic in the number of instructions

- Building dependence graph is $O(n^2)$
- May need to inspect each instruction at each scheduling step: $O(n^2)$
- In practice: closer to linear

Improving Instruction Scheduling

Techniques

- Scheduling loads
 - Register renaming
- } Deal with data hazards
- Loop unrolling
 - Software pipelining
 - Predication and speculation
- } Deal with control hazards

Scheduling Loads

Reality

- Loads can take many cycles (slow caches, cache misses)
- Many cycles may be wasted

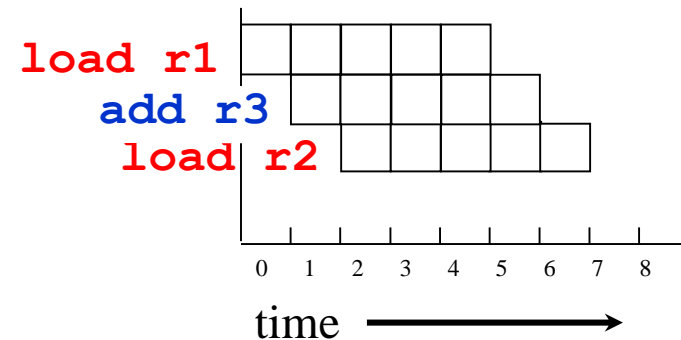
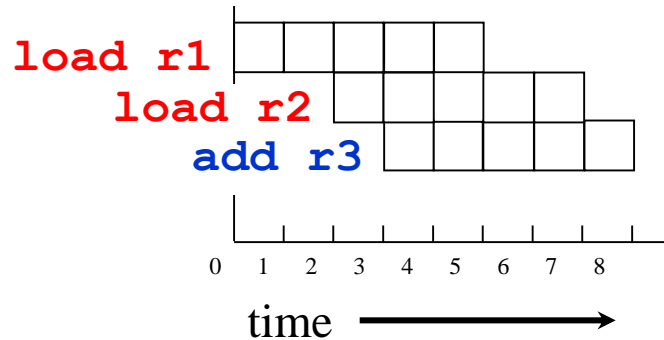
Most modern architectures provide non-blocking (delayed) loads

- Loads never stall
- Instead, the use of a register stalls if the value is not yet available
- Scheduler should try to place loads well before the use of target register

Scheduling Loads (cont)

Hiding latency

- Place independent instructions behind loads



- How many instructions should we insert?
 - Depends on latency
 - Difference between cache miss and cache hits are growing
 - If we underestimate latency: Stall waiting for the load
 - If we overestimate latency: Hold register longer than necessary
Wasted parallelism

Balanced Scheduling [Kerns and Eggers'92]

Idea

- Impossible to know the latencies statically
- Instead of estimating latency, balance the ILP (instruction-level parallelism) across all loads
- Schedule for characteristics of the code instead of for characteristics of the machine

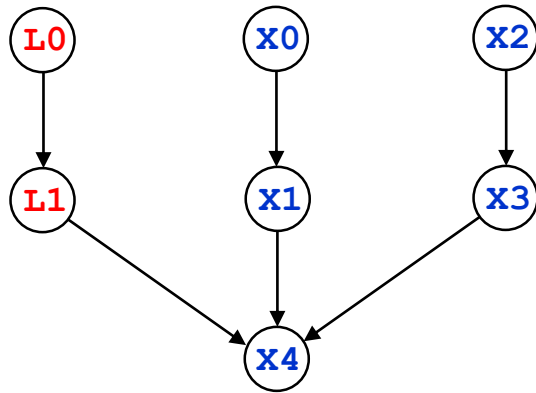
Balancing load

- Compute **load level parallelism**

$$\text{LLP} = 1 + \frac{\text{\# independent instructions}}{\text{\# of loads that can use this parallelism}}$$

Balanced Scheduling Example

Example



LLP for L0 = $1 + 4/2 = 3$

LLP for L1 = $1 + 2/1 = 3$

list scheduling		balanced scheduling
w=5	w=1	
L0	L0	L0
X0	L1	X0
X1	X0	X1
X2	X1	
X3	X2	
L1	X3	
X4	X4	

↖ Pessimistic
 ↖ Optimistic


Register Renaming

Idea


- Reduce false data dependences by reducing register reuse
- Give the instruction scheduler greater freedom

Example

```
add    $r1, $r2, 1
st     $r1, [$fp+52]
mul    $r1, $r3, 2
st     $r1, [$fp+40]
```



```
add    $r1, $r2, 1
st     $r1, [$fp+52]
mul    $r11, $r3, 2
st     $r11, [$fp+40]
```



```
add    $r1, $r2, 1
mul    $r11, $r3, 2
st     $r1, [$fp+52]
st     $r11, [$fp+40]
```

Loop Unrolling

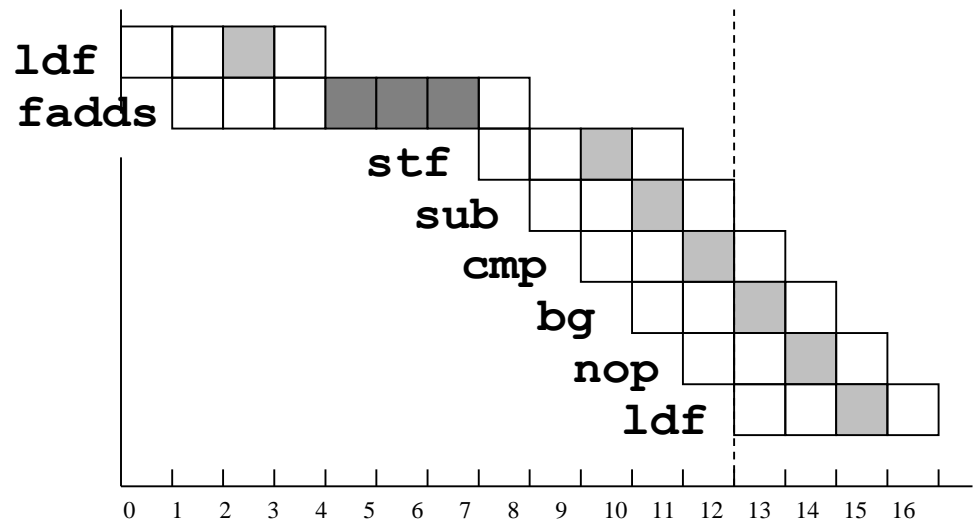
Idea

- Replicate body of loop and iterate fewer times
- Reduces loop overhead (test and branch)
- Creates larger loop body \Rightarrow more scheduling freedom

Example

```
L: ldf    [r1], f0
    fadds f0, f1, f2
    stf    f2, [r1]
    sub    r1, 4, r1
    cmp    r1, 0
    bg     L
    nop
```

Loop
overhead



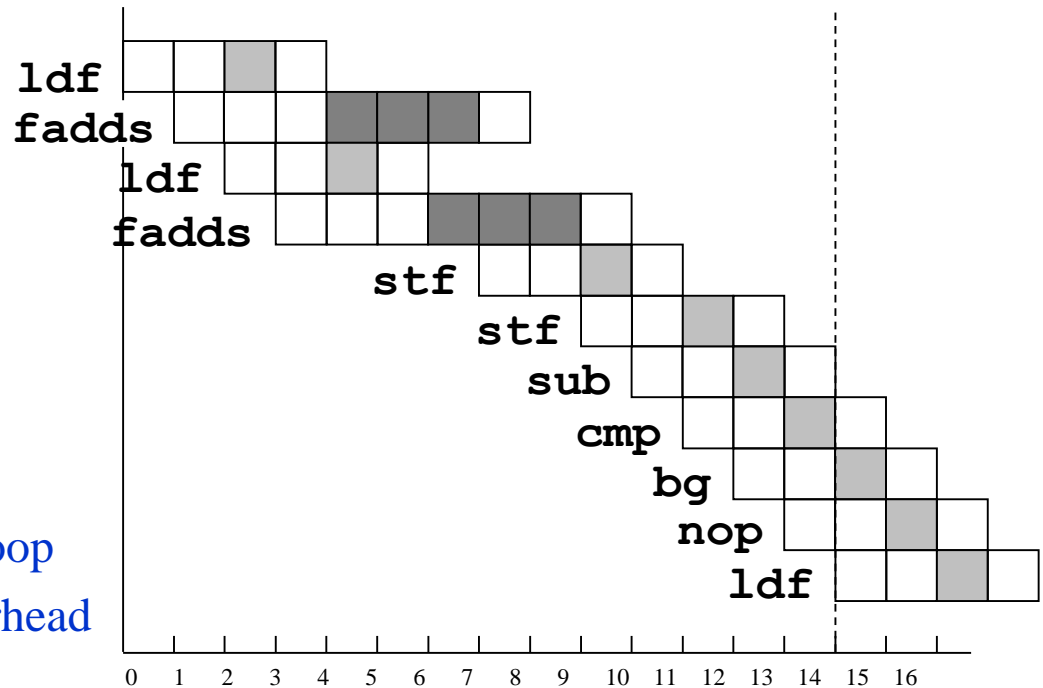
Cycles per iteration: 12

Loop Unrolling Example

Sample loop

```
L: ldf    [r1], f0
    fadds f0, f1, f2
    ldf    [r1-4], f10
    fadds f10, f1, f12
    stf    f2, [r1]
    stf    f12, [r1-4]
    sub    r1, 8, r1
    cmp    r1, 0
    bg     L
    nop
```

Loop
overhead



Cycles per iteration: $14/2 = 7$
(71% speedup!)

The larger window lets us hide the latency of the **fadds** instruction

Phase Ordering Problem

Register allocation

- Tries to reuse registers
- Artificially constrains instruction schedule

Just schedule instructions first?

- Scheduling can dramatically increase register pressure

Classic phase ordering problem

- Tradeoff between memory and parallelism

Approaches

- Consider allocation & scheduling together
- Run allocation & scheduling multiple times
(schedule, allocate, schedule)

Concepts

Instruction scheduling

- Reorder instructions to efficiently use machine resources
- List scheduling

Improving instruction scheduling

- Balanced scheduling
 - Consider characteristics of the program
- Register renaming
- Loop unrolling

Phase ordering problem

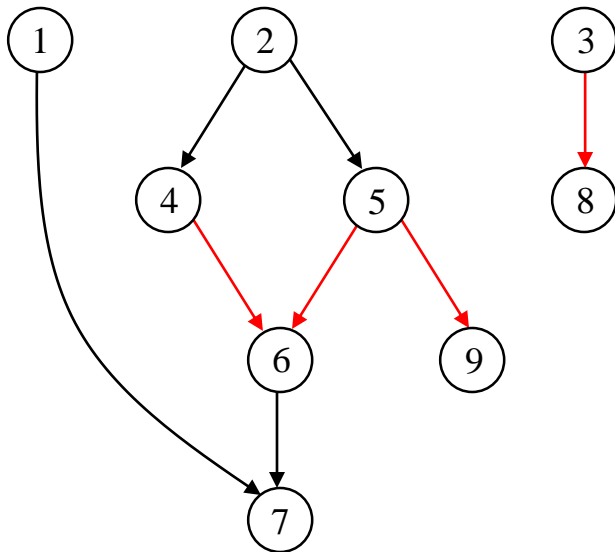
Next Time

Lecture

- More instruction scheduling

Scheduling Example

Dependence Graph



Scheduled Code

```

3  st    a, $r0
2  addi  $sp, 12, $sp
4  ld    $r3, -4($sp)
5  ld    $r4, -8($sp)
8  ld    $r5, a
1  addi  $r2, 1, $r1
6  addi  $sp, 8, $sp
7  st    0($sp), $r2
9  addi  $r4, 1, $r4
  
```

Candidates

```

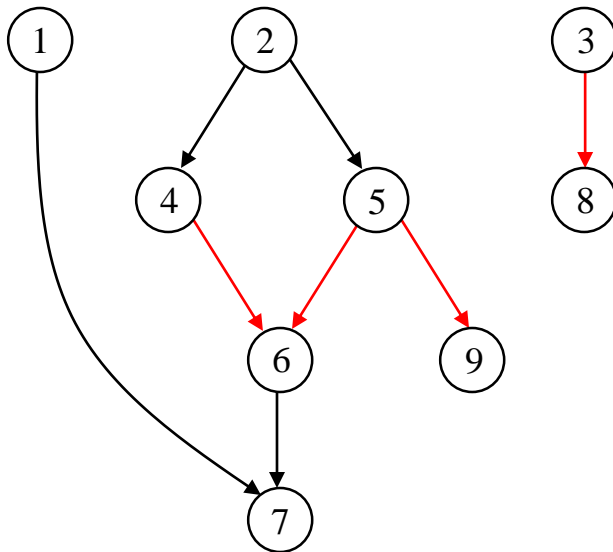
1  addi  $r2, 1, $r1
2  addi  $sp, 8, $sp
3  st    0($sp), $r2
8  ld    $r5, a
9  addi  $r4, 1, $r4
  
```

Hazards in New Schedule

-(8,1)

Scheduling Example

Dependence Graph



Scheduled Code

```
3  st    a, $r0
2  addi  $sp, 12, $sp
4  ld    $r3, -4($sp)
5  ld    $r4, -8($sp)
8  ld    $r5, a
6  addi  $sp, 8, $sp
1  addi  $r2, 1, $r1
7  st    0($sp), $r2
9  addi  $r4, 1, $r4
```

Candidates

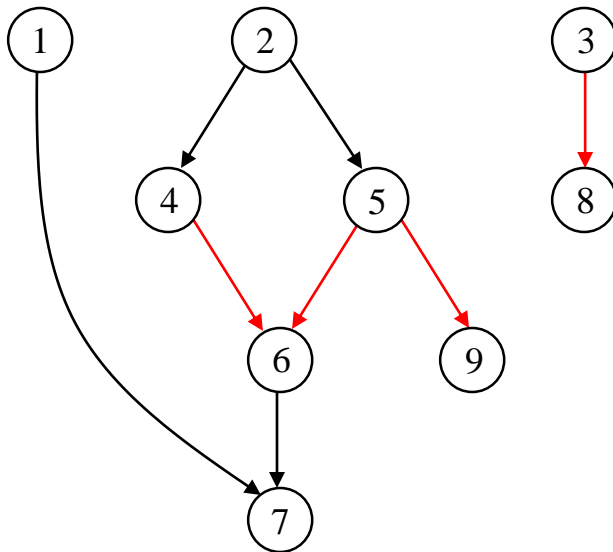
```
1  addi  $r2, 1, $r1
2  addi  $sp, 12, $sp
3  st    0($sp), $r2
8  ld    $r5, a
9  addi  $r4, 1, $r4
```

Hazards in New Schedule

–(8,1)

Scheduling Example

Dependence Graph



Candidates

1 **addi** \$r2,1,\$r1
2 **addi** \$sp,12,\$sp

Scheduled Code

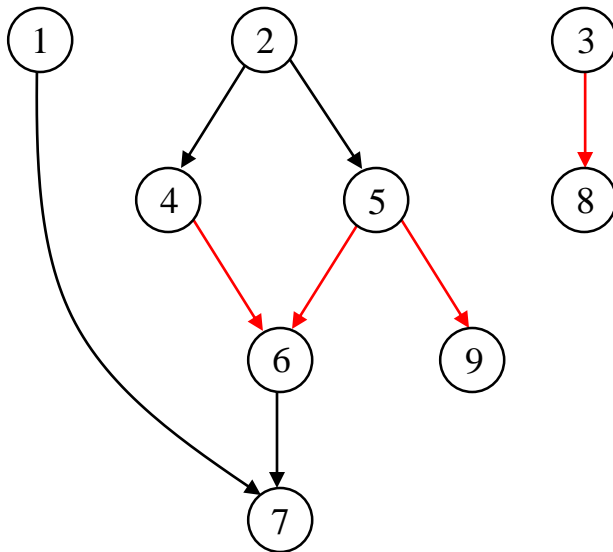
```
3 st a, $r0  
2 addi $sp,12,$sp  
4 ld $r3,-4($sp)  
5 ld $r4,-8($sp)  
1 addi $r2,1,$r1  
6 addi $sp,8,$sp  
8 ld $r5,a  
7 st 0($sp), $r2  
9 addi $r4,1,$r42
```

Hazards in New Schedule

-(8,1)

Scheduling Example

Dependence Graph



Scheduled Code

```
3  st    a, $r0
2  addi  $sp, 12, $sp
4  ld    $r3, -4($sp)
5  ld    $r4, -8($sp)
6  addi  $sp, 8, $sp
1  addi  $r2, 1, $r1
7  st    0($sp), $r2
8  ld    $r5, a
9  addi  $r4, 1, $r4
```

Candidates

```
1  addi  $r2, 1, $r1
2  addi  $sp, 12, $sp
3  st    a, $r0
```

Hazards in New Schedule

–(5,6), (7,8)

Software Pipelining

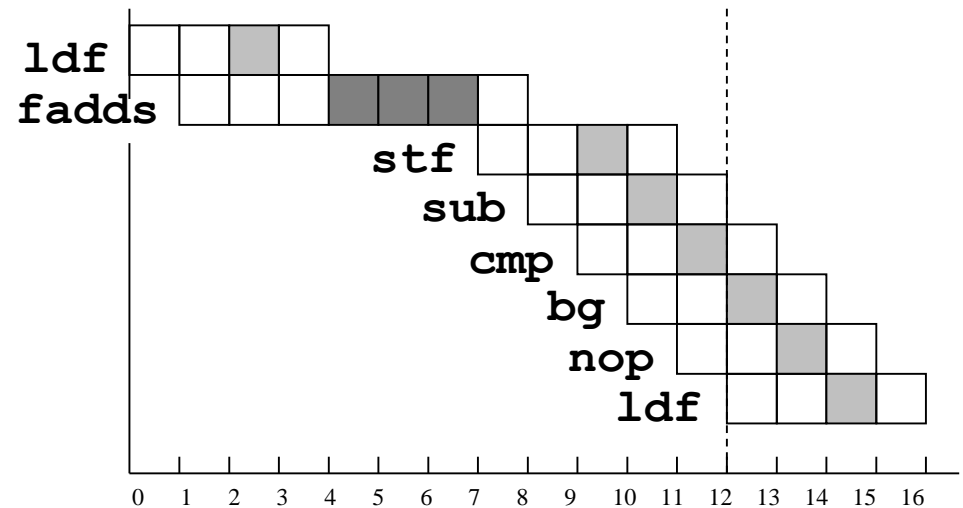
Basic Idea

- Ideally, we could completely unroll loops and have complete freedom in scheduling across iteration boundaries
- Software pipelining is a systematic approach to scheduling across iteration boundaries without doing loop unrolling
- Use control-flow profiles to identify most frequent path through a loop
- If the most frequent path has hazards, try to move some of the long latency instructions to **previous** iterations of the loop
- Three parts of a software pipeline
 - **Kernel**: Steady state execution of the pipeline
 - **Prologue**: Code to fill the pipeline
 - **Epilogue**: Code to empty the pipeline

Software Pipelining Example

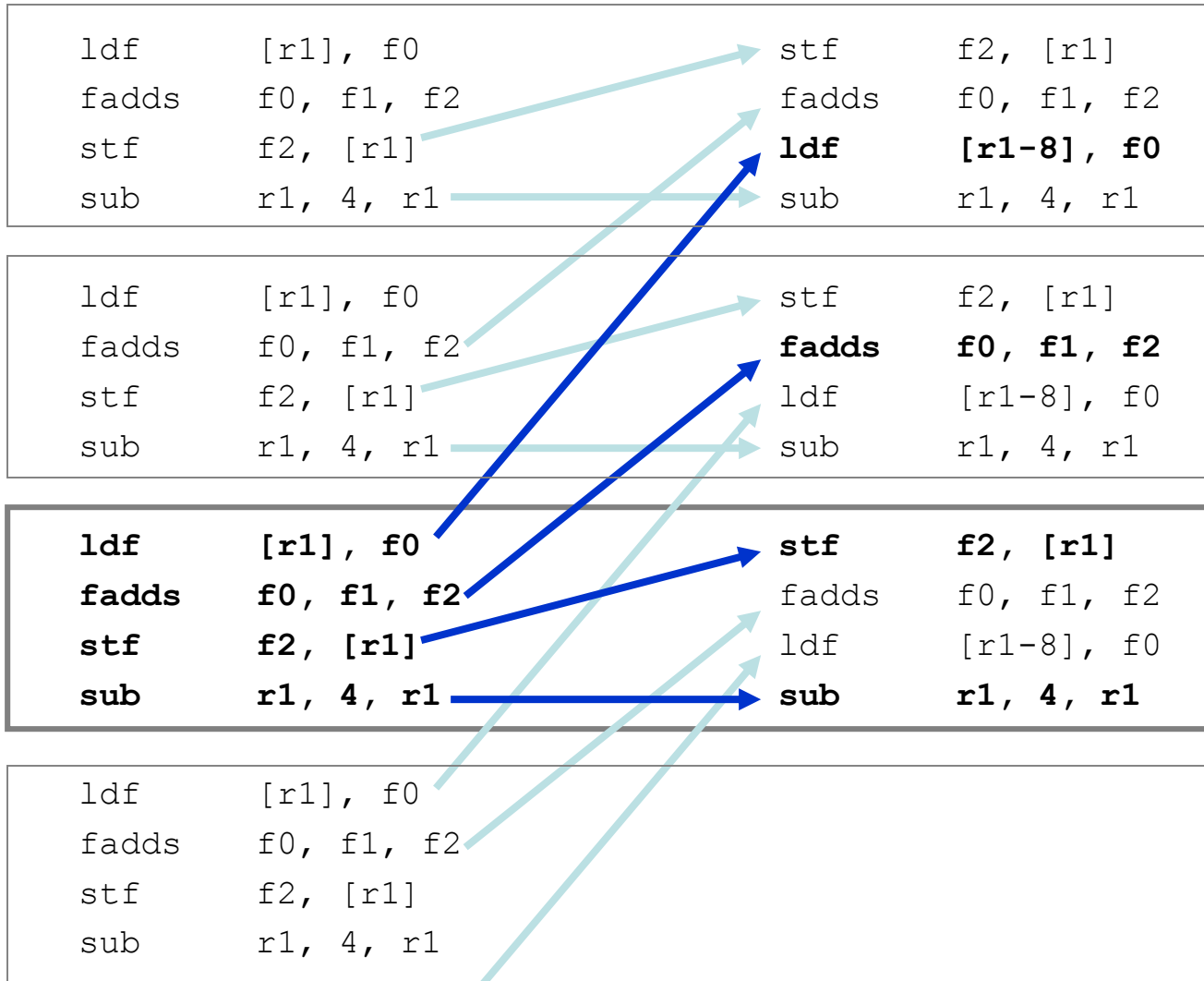
Sample loop (reprise)

```
L: ldf    [r1], f0
   fadds f0, f1, f2
   stf    f2, [r1]
   sub    r1, 4, r1
   cmp    r1, 0
   bg     L
   nop
```



Cycles per iteration: 12

Software Pipelining Example (cont)



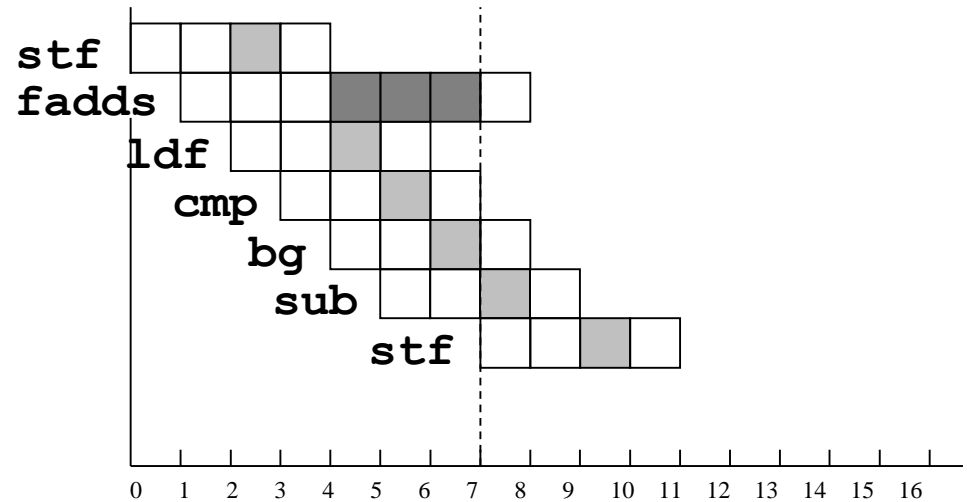
Software Pipelining Example (cont)

Sample loop

```
ldf [r1], f0
fadds f0, f1, f2
ldf [r1-4], f0
```

```
L: stf f2, [r1]
fadds f0, f1, f2
ldf [r1-8], f0
cmp r1, 8
bg L
```

```
sub r1, 4, r1
stf f2, [r1]
sub r1, 4, r1
fadds f0, f1, f2
stf f2, [r1]
```



Cycles per iteration: 7 (71% speedup!)