Instruction Scheduling

Last week

- Register allocation

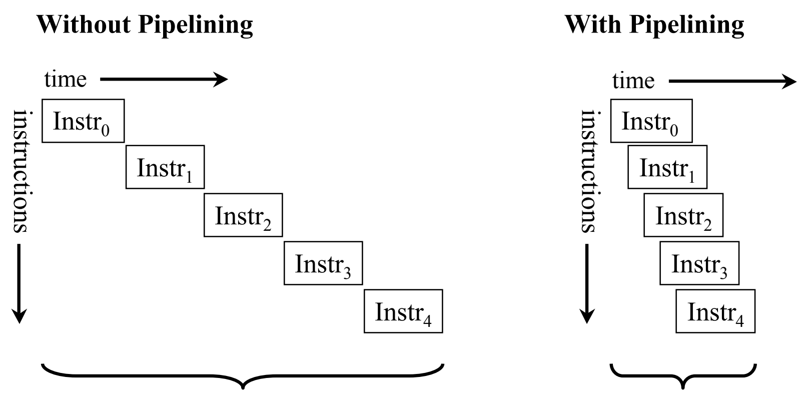
Today

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

Background: Pipelining Basics

Idea

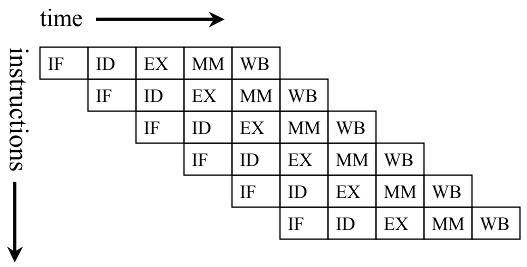
- Begin executing an instruction **before** completing the previous one



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



IF	ID	EX	MM	WB					
	IF	ID	EX	MM	WB				
		IF	ID	EX	MM	WB			
			IF	ID	EX	MM	WB		
				IF	ID	EX	MM	WB	
					IF	ID	EX	MM	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)
- Issue 2 (Pentium), 4 (UltraSPARC) or more (dead Compaq EV8) instructions per cycle
- 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

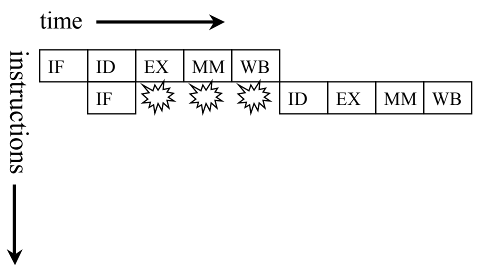
– Stall (insert bubbles into pipeline)

Stalls (Data Hazards)

Code

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

Pipeline picture

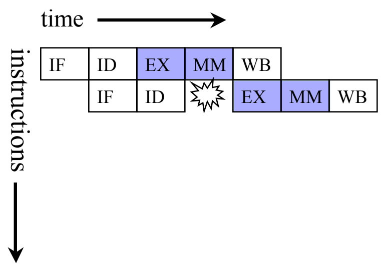


Stalls (Structural Hazards)

Code

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

Pipeline Picture

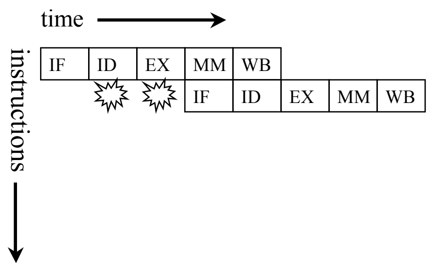


Stalls (Control Hazards)

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

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

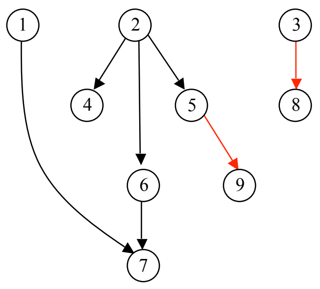
- Nodes represent instructions
- Edges (s_1,s_2) represent dependences between instructions
  - Instruction s_1 must execute before s_2
- Sometimes called **data dependence graph** or **data-flow graph**

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

Consider these questions

– 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...

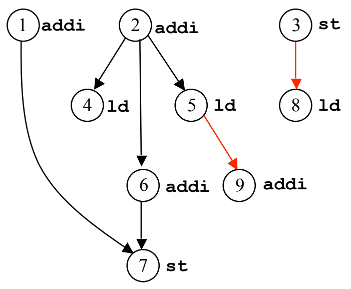
- It does not interlock with the previously scheduled instruction (avoid stalls)
- It interlocks with its successors in the dependence graph (may enable successors to be scheduled without stall)
- It has many successors in the graph (may enable successors to be scheduled with greater flexibility)
- It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

Scheduling Algorithm

```
Build dependence graph G
Candidates ← set of all roots (nodes with no in-edges) in G
while Candidates ≠ ∅
      Select instruction s from Candidates          {Using heuristics—in order}
      Schedule s
      Candidates ← Candidates – s
      Candidates ← Candidates ∪ "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
6  addi  $sp,8,$sp
5  ld    $r4,-8($sp)
7  st    0($sp),$r2
4  ld    $r3,-4($sp)
8  ld    $r5,a
9  addi  $r4,1,$r4
```

Hazards in new schedule

(8,1)

Scheduling Example (cont)

Original code

```
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
```

```
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
```

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

- Register renaming
- Scheduling loads

Deal with data hazards

- Loop unrolling
- Software pipelining
- Predication and speculation

Deal with control hazards

Register Renaming

Idea

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

Example

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

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

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

- Register renaming

Phase ordering problem

Next Time

Lecture

- More instruction scheduling
  - scheduling loads
  - loop unrolling
  - software pipelining