IC220 Set #19:
Laundry, Co-dependency, and other Hazards
of Modern (Architecture) Life

Return to Chapter 4

Midnight Laundry

<table>
<thead>
<tr>
<th>Task order</th>
<th>6 PM</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>1</th>
<th>2 AM</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Smarty Laundry

Pipelining

- Improve performance by increasing instruction throughput

Ideal speedup is number of stages in the pipeline. Do we achieve this?
Basic Idea

Pipeched Datapath
Pipeline Diagrams

Clock cycle: 1 2 3 4 5 6 7

add $s0, $s1, $s2
sub $a1, $s2, $a3
add $t0, $t1, $t2

Assumptions:
• Reads to memory or register file in 2nd half of clock cycle
• Writes to memory or register file in 1st half of clock cycle

What could go wrong?

Problem: Dependencies

• Problem with starting next instruction before first is finished

Clock cycle: 1 2 3 4 5 6 7 8

sub $s0, $s1, $s2
and $a1, $s0, $a3
add $t0, $t1, $s0
or $t2, $s0, $s0

Dependencies that “go backward in time” are ________________

Will the “or” instruction work properly?
Solution: Forwarding

Use temporary results, don’t wait for them to be written

Clock cycle: 1 2 3 4 5 6 7 8

sub $s0, $s1, $s2

and $a1, $s0, $a3

add $t0, $t1, $s0

or $t2, $s0, $s0

Where do we need this?
Will this deal with all hazards?

Problem?

Clock cycle: 1 2 3 4 5 6 7

lw $t0, 0($s1)

sub $a1, $t0, $a3

add $a2, $t0, $t2

Forwarding not enough...
When an instruction tries to __________ a register following a __________ to the same register.
Solution: “Stall” later instruction until result is ready

<table>
<thead>
<tr>
<th>Clock cycle:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $t0, 0($s1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub $a1, $t0, $a3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add $a2, $t0, $t2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Why does the stall start after ID stage?

Assumptions

- For exercises/exams/everything assume...
  - The MIPS 5-stage pipeline
  - That we have forwarding

...unless told otherwise
Exercise #1 – Pipeline diagrams

• Draw a pipeline stage diagram for the following sequence of instructions. Start at cycle #1. You don’t need fancy pictures – just text for each stage: ID, MEM, etc.
  add $s1, $s3, $s4
  lw $v0, 0($a0)
  sub $t0, $t1, $t2

• What is the total number of cycles needed to complete this sequence?
• What is the ALU doing during cycle #4?
• When does the sub instruction writeback its result?
• When does the lw instruction access memory?

Exercise #2 – Data hazards

• Consider this code:
  1. add $s1, $s3, $s4
  2. add $v0, $s1, $s3
  3. sub $t0, $v0, $t2
  4. and $a0, $v0, $s1

  1. Draw lines showing all the data dependencies in this code

  2. Which of these dependencies do not need forwarding to avoid stalling?
Exercise #3 – Data hazards

• Draw a pipeline diagram for this code. Show stalls where needed.
  1. add $s1, $s3, $s4
  2. lw $v0, 0($s1)
  3. sub $v0, $v0, $s1

Exercise #4 – More Data hazards

• Draw a pipeline diagram for this code. Show stalls where needed.
  1. lw $s1, 0($t0)
  2. lw $v0, 0($s1)
  3. sw $v0, 4($s1)
  4. sw $t0, 0($t1)
The Pipeline Paradox

- Pipelining does not ____________ the execution time of any ____________ instruction.

- But by _______________ instruction execution, it can greatly improve performance by _______________ the _______________.

Control Hazards

- What might be a problem with pipelining the following code?

  ```assembly
  beq $a0, $a1, Else
  lw $v0, 0($s1)
  sw $v0, 4($s1)
  Else: add $a1, $a2, $a3
  ```

- What other kinds of instructions would cause this problem?
Control Hazard Strategy #1: Predict not taken

- What if we are wrong?

- Assume branch target and decision known at end of ID cycle. Show a pipeline diagram for when branch is taken.
  
  ```
  beq $a0, $a1, Else
  lw  $v0, 0($s1)
  sw  $v0, 4($s1)
  Else: add $a1, $a2, $a3
  ```

Control Hazard Strategies

1. Predict not taken
   One cycle penalty when we are wrong – not so bad
   Penalty gets bigger with longer pipelines – bigger problem

2.

3.
Branch Prediction

With more sophistication can get 90-95% accuracy
Good prediction key to enabling more advanced pipelining techniques!

Code Scheduling to Improve Performance

- Can we avoid stalls by rescheduling?

```plaintext
lw $t0, 0($t1)
add $t2, $t0, $t2
lw $t3, 4($t1)
add $t4, $t3, $t4
```

- Dynamic Pipeline Scheduling
  - Hardware chooses which instructions to execute next
  - Will execute instructions out of order (e.g., doesn’t wait for a dependency to be resolved, but rather keeps going!)
  - Speculates on branches and keeps the pipeline full (may need to rollback if prediction incorrect)
Dynamic Pipeline Scheduling

- Let hardware choose which instruction to execute next (might execute instructions out of program order)
- Why might hardware do better job than programmer/compiler?

<table>
<thead>
<tr>
<th>Example #1</th>
<th>Example #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $t0, 0($t1)</td>
<td>sw $s0, 0($s3)</td>
</tr>
<tr>
<td>add $t2, $t0, $t2</td>
<td>lw $t0, 0($t1)</td>
</tr>
<tr>
<td>lw $t3, 4($t1)</td>
<td>add $t2, $t0, $t2</td>
</tr>
<tr>
<td>add $t4, $t3, $t4</td>
<td></td>
</tr>
</tbody>
</table>

Exercise #1

- Can you rewrite this code to eliminate stalls?
  1. lw $s1, 0($t0)
  2. lw $v0, 0($s1)
  3. sw $v0, 4($s1)
  4. add $t0, $t1, $t2
Exercise #2

- Show a pipeline diagram for the following code, assuming:
  - The branch is predicted not taken
  - The branch actually is taken

```assembly
lw $t1, 0($t0)  
beq $s1, $s2, Label2  
sub $v0, $v1, $v2  
Label2: add $t0, $t1, $t2
```

Exercise #3 – True or False?

1. A pipelined implementation will have a faster clock rate than a comparable single cycle implementation

2. Pipelining increases performance by splitting up each instruction into stages, thereby decreasing the time needed to execute each instruction.

3. “Backwards” branches are likely to not be taken