Pipeline Control Hazards
CSC 211 – December 3, 2020
Datapath Lab Q&A

We added a mux that wasn't in the description - is that okay?

Yes - there are many reasonable ways to implement branches.
Data Hazards, Continued
Consider This Program:

A lw $t0, 0($a0)

B addi $t0, $t0, 1

C lw $t1, 0($a1)

D addi $t1, $t1, 1

E and $t2, $t0, $t1

How long will this program take to run if we avoid data hazards by...

(a) statically scheduling instructions

(b) stalling

(c) forwarding and stalling
An Adversarial Input

We've seen some cases where pipeline stalls are the only option for avoiding a data hazard. While we usually think about making programs run faster, it's useful to think about a worst-case input.

Write a five-instruction MIPS assembly program that runs as slowly as possible. You should maximize the number of pipeline stalls, even with forwarding. Allowing instruction reordering should not improve your program's performance.

```
 lw $t0, 0($s0)
lw $t1, 0($t0)
lw $t2, 0($t1)
lw $t3, 0($t2)
lw $t4, 0($t3)
```
Control Hazards
Show the pipeline execution for:

addi $s0, $zero, 0
j somewhere
addi $s0, $s0, 1
addi $s1, $s1, 1
...
somewhere:
addi $s2, $s2, 1
FIGURE 4.46 The pipelined datapath of Figure 4.41 with the control signals identified. This datapath borrows the control signals from Section 4.4.2. The 32-bit output of the ALU is used as input to the MemWrite stage. The instruction address is driven to the ALU control unit, and the ALU control unit generates the signals Add, Alusrc, and RegWrite to control the ALU. The MemWrite stage is used to write the data back to memory.
Show the pipeline execution for:

```
addi $s0, $zero, 0
beq $s0, $zero, somewhere
addi $s0, $s0, 1
addi $s1, $s1, 1
...
somewhere:
    addi $s2, $s2, 1
```

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
FIGURE 4.46  The pipelined datapath of Figure 4.41 with the control signals identified. This datapath borrows the control signals from the three-stage pipelined datapath of Figure 4.39.
Control Hazard Basics

What causes a control hazard?

The datapath does not know which instruction to fetch in a specific clock cycle. It may not even know that a branch or jump is executing.

How can a processor deal with a control hazard?

Branch Prediction - the datapath guesses whether a branch is taken or not, then proceeds assuming its guess is correct.

Delay Slot - change the rules so programmers have to solve the issue.

Redesign pipeline - move branch decisions earlier in the pipeline.

Stall - just wait until we know the branch outcome.
Branch Predictor Strategies

What are reasonable strategies for predicting whether a branch will be taken?

- Predict always taken
- Predict never taken
- User-supplied prediction

Last branch outcome (for any branch)
Last branch outcome (for this specific branch)

Saturating counter

What are the advantages or downsides of each strategy?

- Static prediction: simple, but not smart
- Last branch outcome - a bit smarter, doesn't deal with conditionals inside loops
- Saturating counter - may be most accurate, most complex
Consider this code snippet:

\[
[s0 = 10]^{\text{initial state}}
\]

```assembly
top:
    addi $s0, $s0, -1  
    bne $s0, $zero, top  
    nop  
end:
```

What is the branch miss rate for...

1. Predict *always taken*
   \[
   \text{miss rate} = \frac{1}{10} = 10\%
   \]

2. Predict *never taken*
   \[
   \text{miss rate} = \frac{9}{10} = 90\%
   \]

3. A two-bit saturating counter  
   *(initially in the weak taken state)*
   \[
   \text{miss rate} = \frac{1}{10} = 10\%
   \]

Actual branch outcome:

```
T T T T T T T T N
```
Consider this code snippet:

```assembly
$s0 = 10
while:
    beq $s0, $zero, end
    nop
    addi $s0, $s0, -1
    j while
    nop
end:
```

What is the branch miss rate for...

1. Predict *always taken*

2. Predict *never taken*

3. A two-bit saturating counter *(initially in the weak taken state)*