Stalls and Flushes
Today’s Lecture

- **Stalls**
  - Some data dependencies that cannot be resolved with only forwarding
  - Example: Loads
  - Example: Branches

- **Flushes**
  - Some data dependencies cause us to fetch the wrong instructions
  - Flushes remove wrong instructions from the pipeline
Use arrows to show dependencies: Arrows that point backwards reveal data hazards

sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
Forward values from pipeline registers so later instructions can use the correct value

- In clock cycle 4, AND gets R[1]-R[3] from EX/MEM
- In cycle 5, OR gets R[1]-R[3] from MEM/WB

```
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
```
Consider the following sequence of instructions

\[
\text{lw} \quad \$2, \ 20(\$3)
\]

and

\[
\text{and} \quad \$12, \ \$2, \ \$5
\]
When we cannot resolve a hazard with forwarding alone, we need to **stall** the pipeline

- To stall the pipeline, introduce a one-cycle delay into the pipeline, (a bubble).

![Diagram showing pipeline with stall at cycle 5]

- Notice that we’re still using forwarding in cycle 5, to get data from the MEM/WB pipeline register to the ALU.
Without forwarding we would have to stall for two cycles for \( \text{l}w \) to reach writeback.

- In general, you can always stall to avoid hazards—but dependencies are very common in real code, and stalling often can reduce performance by a significant amount.

\[ \text{l}w \; \$2, \; 20(\$3) \]

\[ \text{and} \; \$12, \; \$2, \; \$5 \]
Stalling delays the entire pipeline

- Delay the second instruction -> delay the third instruction too.
- Why? (two reasons)

<table>
<thead>
<tr>
<th>lw</th>
<th>$2, 20($3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>$12, $2, $5</td>
</tr>
<tr>
<td>or</td>
<td>$13, $12, $2</td>
</tr>
</tbody>
</table>
Step 1: Force the two instructions after `lw` to remain in ID and IF stages for one extra cycle.

- Don’t update the PC, so the current IF stage is repeated.
- Don’t update the IF/ID register, so the ID stage is also repeated.
Step 2: Set control signals for EX, MEM, and WB to zero for cycles 4, 5, and 6 respectively.

lw $2, 20($3)

and $12, $2, $5

or $13, $12, $2

No ALU!
No Mem!
No Reg File!
Stalling converts some instructions to nop (“no operation”) instructions

lw  $2, 20($3)

and -> nop

and  $12, $2, $5

or  $13, $12, $2
Implement the \texttt{nop} in MIPS with 
\texttt{sll $0, $0, 0}

<table>
<thead>
<tr>
<th>Name,</th>
<th>Mnemonic</th>
<th>Format</th>
<th>Operation</th>
<th>Opcode/Funct (Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift Left Logical</td>
<td>\texttt{sll}</td>
<td>R</td>
<td>(R[rd] = R[rt] \ll \text{shamt})</td>
<td>0/00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
To detect stalls, we need to find dependencies

- Recall how we detected hazards that required forwarding?

  \[
  \text{if } (\text{EX/MEM.RegWrite} = 1 \\
  \quad \text{and } \text{EX/MEM.RegisterRd} = \text{ID/EX.RegisterRs}) \\
  \text{then Bypass Rs from EX/MEM stage latch}
  \]

\[
\text{sub } \$2, \$1, \$3
\]

\[
\text{and } \$12, \$2, \$5
\]
Which set of pipeline registers should we use to detect that we need to stall?

1. lw  $2, 20($3)

2. and $12, $2, $5

3. or  $13, $12, $2

Next instruction
Detect stalls when $lw$ finishes the decode stage

$lw \quad $2, 20($3) 

and $12, \quad $2, \quad $5$

What is the stall condition?

if ( 

then stall
Detect stalls when \( \text{lw} \) finishes the decode stage

\[
\text{lw} \quad $2, 20($3) \quad \text{and} \quad $12, $2, $5
\]

What is the stall condition?

\[
\text{if (ID/EX.MemRead = 1 and} \\
\quad (\text{ID/EX.RegisterRt = IF/ID.RegisterRs or} \\
\quad \text{ID/EX.RegisterRt = IF/ID.RegisterRt})) \\
\text{then stall}
\]
Table method for tracking stalls

- `lw $2, 20($3)`
- `and $12, $2, $5`
- `or $13, $12, $2`

Clock Cycle:

- F
- D
- E
- M
- WB

**Diagram:**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>IM</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>WB</td>
</tr>
<tr>
<td>if/id</td>
<td>Reg</td>
<td>ld/ex</td>
<td>ex/mem</td>
<td>DM</td>
<td>mem/wb</td>
</tr>
<tr>
<td>Reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Reg</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lw</code></td>
<td>$2, 20($3)</td>
</tr>
<tr>
<td><code>and</code></td>
<td>$12, $2, $5</td>
</tr>
<tr>
<td><code>or</code></td>
<td>$13, $12, $2</td>
</tr>
</tbody>
</table>
Adding hazard detection to the CPU
Generalizing Forwarding/Stalling

- What if data memory access was so slow, we wanted to pipeline it over 2 cycles?

- How many bypass inputs would the muxes in EXE have?
- Which instructions in the following require stalling and/or bypassing?

<table>
<thead>
<tr>
<th>Clock cycle</th>
<th>DM</th>
<th>Reg</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IM</td>
<td>Reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>$13, 0 ($11)</td>
</tr>
<tr>
<td>add</td>
<td>$7, $8, $9</td>
</tr>
<tr>
<td>add</td>
<td>$15, $7, $13</td>
</tr>
</tbody>
</table>
When are branches resolved (when do I know for certain where I am branching to)?
Branch target address and zero flag are determined in the EX stage, creating a control hazard

- Therefore, branch decision cannot be made until the end of the EX stage.
- But we need to know which instruction to fetch next, in order to keep the pipeline running!

```
beq  $2, $3, Label
```
Could handle the hazard by stalling until cycle 4 after the branch destination is known.

beq $2, $3, Label

???
Alternatively, we could guess whether the branch is taken (i.e., **Branch Prediction**)

- Easy to just assume the branch is *not* taken, just increment PC
- If we’re correct, then there is no problem and the pipeline keeps going at full speed.

```
beq    $2, $3, Label
```

```
next instruction 1
```

```
next instruction 2
```
If we mispredict the branch, we need to **flush** two incorrect instructions from the pipeline.

Restart execution at the target label.

**beq** $2$, $3$, **Label**

- **next instruction 1**
- **next instruction 2**

**Label**: ...
On a misprediction, branch prediction wastes two cycles; stalling always wastes those two cycles

- All modern CPUs use branch prediction.
  - Accurate predictions are important for optimal performance.
  - Most CPUs predict branches dynamically—statistics are kept at run-time to determine the likelihood of a branch being taken.

- The pipeline structure also has a big impact on branch prediction.
  - A longer pipeline may require more instructions to be flushed for a misprediction, resulting in more wasted time and lower performance.
  - We must also be careful that instructions do not modify registers or memory before they get flushed.
Flush the pipeline by turning the instructions in ID/EX and IF/ID into `nop`

A flush is required when the instruction in EX is a BEQ and “zero” is 1.
Summary

- Three kinds of hazards conspire to make pipelining difficult.
- **Structural hazards** result from not having enough hardware available to execute multiple instructions simultaneously.
  - These are avoided by adding more functional units (e.g., more adders or memories) or by redesigning the pipeline stages.
- **Data hazards** can occur when instructions need to access registers that haven’t been updated yet.
  - Hazards from R-type instructions can be avoided with forwarding.
  - Loads can result in a “true” hazard, which must stall the pipeline.
- **Control hazards** arise when the CPU cannot determine which instruction to fetch next.
  - We can minimize delays by doing branch tests earlier in the pipeline.
  - We can also take a chance and predict the branch direction, to make the most of a bad situation.