Stalls and Flushes

Friday is optional (not on HW or Exams)
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**

Clock cycle

<table>
<thead>
<tr>
<th></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>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub $2, $1, $3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and $12, $2, $5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or $13, $6, $2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add $14, $2, $2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw $15, 100($2)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

tail shows when register is written

head shows when register is read
**Forward** values from pipeline registers so later instructions can use the correct value


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

Can we resolve this data hazard by forwarding data from...

a) LW execute (cycle 3) to AND execute (cycle 4)
   - LW memory (cycle 4) to AND execute (cycle 4)
   c) LW memory (cycle 4) to AND memory (cycle 5)
   d) None of the above
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).

- 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 \texttt{lw} 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.
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>
```

Clock cycle

1  2  3  4  5  6  7  8
Step 1: Force the two instructions after \( \text{lw} \) to remain in ID and IF stages for one extra cycle

- This is easily accomplished.
  - 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

```
1000 lw $2, 20($3)
1004 and -> nop
1008 and $12, $2, $5
1006 or $13, $12, $2
```
Implement the \texttt{nop} in MIPS with \texttt{sll} $0$, $0$, $0$.

<table>
<thead>
<tr>
<th>Name, Mnemonic</th>
<th>Format</th>
<th>Operation</th>
<th>Opcode/Funct (Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift Left Logical</td>
<td>sll</td>
<td>R</td>
<td>$R[rd] = R[rt] \ll \text{shamt}$</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>000000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>
To detect stalls, we need to find dependencies

- Recall how we detected hazards that required forwarding?

  ```
  if (EX/MEM.RegWrite = 1
      and EX/MEM.RegisterRd = ID/EX.RegisterRs)
  then Bypass Rs from EX/MEM stage latch
  ```

  sub $2, $1, $3

  and $12, $2, $5
Which set of pipeline registers should we use to detect that we need to stall?

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

Next instruction

PC
Detect stalls when `lw` finishes the decode stage

What is the stall condition?

```plaintext
if ( 
then stall
```
Detect stalls when \texttt{lw} finishes the decode stage

\texttt{lw} $\$2, 20(\$3)\text{ and }\$12, \$2, \$5$

What is the stall condition?

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

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

<table>
<thead>
<tr>
<th>IM</th>
<th>F</th>
<th>D</th>
<th>E</th>
<th>M</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>DM</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>WB</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>DM</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>WB</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>
Table method for tracking stalls

In which cycle is `add $4, $9, $2` in the WB stage?
Table method for tracking stalls

In which cycle is `add $4, $9, $2` in the WB stage?
Adding hazard detection to the CPU
Adding hazard detection to the CPU
The hazard detection unit

- The hazard detection unit’s inputs are as follows.
  - IF/ID.RegisterRs and IF/ID.RegisterRt, the source registers for the current instruction.
  - ID/EX.MemRead and ID/EX.RegisterRt, to determine if the previous instruction is LW and, if so, which register it will write to.

- By inspecting these values, the detection unit generates three outputs.
  - Two new control signals PCWrite and IF/ID Write, which determine whether the pipeline stalls or continues.
  - A mux select for a new multiplexer, which forces control signals for the current EX and future MEM/WB stages to 0 in case of a stall.
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></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$13, 0 ($11)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$7, $8, $9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$15, $7, $13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In which cycle is add $15, $7, $13 in the WB stage?
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!
Could handle the hazard by stalling until cycle 4 after the branch destination is known.
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.

![Branch Pipeline Diagram](image-url)
If we mispredict the branch, we need to **flush** two incorrect instructions from the pipeline.

Restart execution at the target label.
i>clicker

Which method of handling branches will generally lead to better performance?

a) Stalling
b) Branch prediction and flushes
c) They will give about the same performance
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 \textit{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.