Stalls and Flushes

Pick up handout
Exam 4 is alive.
Wed lecture is optional
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**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Clock cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub $2, $1, $3</td>
<td>1</td>
</tr>
<tr>
<td>and $12, $2, $5</td>
<td>2</td>
</tr>
<tr>
<td>or $13, $6, $2</td>
<td>3</td>
</tr>
<tr>
<td>add $14, $2, $2</td>
<td>4</td>
</tr>
<tr>
<td>sw $15, 100($2)</td>
<td>5</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

- 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

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

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

lw $2, 20($3)

and $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).

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!

\begin{itemize}
  \item 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.
\end{itemize}
Stalling delays the entire pipeline

- Delay the second instruction -> delay the third instruction too.
  - Why? (two reasons)
Step 1: Force the two instructions after `lw` to remain in ID and IF stages for one extra cycle

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

- 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.

\[ \text{lw } \$2, 20(\$3) \]
\[ \text{and } \$12, \$2, \$5 \]
\[ \text{or } \$13, \$12, \$2 \]
Stalling converts some instructions to nop ("no operation") instructions

\[
\begin{align*}
\text{not} &\quad \text{stall} \\
\text{lw} &\quad $2, 20($3) \\
\text{and} &\quad \rightarrow \text{nop} \\
\text{stall} &\quad \text{and} \quad $12, $2, $5 \\
\text{or} &\quad $13, $12, $2
\end{align*}
\]
To detect stalls, we need to find dependencies

- Recall how we detected hazards that required forwarding?

  if \((\text{EX/MEM.RegWrite} = 1 \text{ and } \text{EX/MEM.RegisterRd} = \text{ID/EX.RegisterRs})\) 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)
- Detect stall in ID
- Consumer and $12, $2, $5
- or $13, $12, $2

Next instruction
Detect stalls when \( \text{lw} \) finishes the decode stage

What is the stall condition?

\[
\text{if (id/ex, mem-read) \&\& ((id/ex, rd = if/id, rs) \| (id/ex, rd = if/id, rt))}
\]

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

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

What is the stall condition?

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

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

<table>
<thead>
<tr>
<th>Clock Cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>IM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td>IM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reg</td>
<td></td>
<td>Reg</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>id/ex</td>
<td></td>
<td>id/ex</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ex/mem</td>
<td></td>
<td>ex/mem</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DM</td>
<td></td>
<td>DM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mem/wb</td>
<td></td>
<td>mem/wb</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reg</td>
<td></td>
<td>Reg</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IM</th>
<th>D</th>
<th>E</th>
<th>M</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>lw</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>IM</td>
<td>D</td>
<td>E</td>
<td>M</td>
<td>WB</td>
<td></td>
</tr>
<tr>
<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>
</tr>
<tr>
<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>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IM</th>
<th>D</th>
<th>E</th>
<th>M</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IM</th>
<th>D</th>
<th>E</th>
<th>M</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$2$
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?

| lw    | $13, 0 ($11) |
| add   | $7, $8, $9   |
| add   | $15, $7, $13 |

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.
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 \texttt{nop}.

A flush is required when the instruction in EX is a BEQ and “zero” is 1.
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>\texttt{sll}</td>
<td>R</td>
<td>\texttt{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>\texttt{000000}</td>
<td>\texttt{0000}</td>
<td>\texttt{0000}</td>
<td>\texttt{0000}</td>
<td>\texttt{0000}</td>
<td>\texttt{0000}</td>
</tr>
</tbody>
</table>
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.