Implementation of forwarding

**Problem:** Data dependences

- add $4, $5, $6
- sub $1, $2, $4
- lw $3, 300($4)
- and $7, $8, $4

**Solution:** Data forwarding

<table>
<thead>
<tr>
<th>IF stage</th>
<th>ID stage</th>
<th>EX stage</th>
<th>MEM stage</th>
<th>WB stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycle 3</td>
<td>lw $3, 300($4)</td>
<td>sub $1, $2, $4</td>
<td>add $4, $5, $6</td>
<td></td>
</tr>
<tr>
<td>Cycle 4</td>
<td>and $7, $8, $4</td>
<td>lw $3, 300($4)</td>
<td>sub $1, $2, $4</td>
<td>add $4, $5, $6</td>
</tr>
<tr>
<td>Cycle 5</td>
<td>and $7, $8, $4</td>
<td>lw $3, 300($4)</td>
<td>sub $1, $2, $4</td>
<td>add $4, $5, $6</td>
</tr>
</tbody>
</table>

Forwarding from EX/MEM to ID/EX

- The *sub* reads the wrong value from $4 (stored in ID/EX)
- The *add* has the correct value to be written in $4 (stored in EX/MEM)

- The *sub* can replace the wrong value read from $4 with the correct one (from EX/MEM) before feeding it to the ALU.
- Why two muxes??
Forwarding from MEM/WB to ID/EX

- The `and` reads the wrong value from $3$ (stored in ID/EX)
- The `add` has the correct value to be written in $3$ (stored in MEM/WB)

- The `and` can use the correct value that will be written in $3$ (from MEM/WB)

Combining the two forwarding datapaths

The default is to select the value from ID/EX buffer

Select the value from the MEM/WB buffer if
- the instruction in MEM/WB will write into a register $X$
- the instruction in ID/EX did read from register $X$

Select the value from the EX/MEM buffer if
- the instruction in EX/MEM will write into a register $X$
- the instruction in ID/EX did read from register $X$
Combining the forwarding datapaths

• Set forward control signal, A, to 10 if
  • EX/MEM.RegisterWrite, and
  • EX/MEM.Rd = ID/EX.Rs, and
  • ID/EX.Rs != 0

• Set forward control signal, B, to 10 if
  • EX/MEM.RegisterWrite, and
  • EX/MEM.Rd = ID/EX.Rt, and
  • ID/EX.Rt != 0

Combining the forwarding datapaths

• Set forward control signal, A, to 01 if
  • MEM/WB.RegisterWrite, and
  • MEM/WB.Rd = ID/EX.Rs, and
  • ID/EX.Rs != 0

• Set forward control signal, B, to 01 if
  • MEM/WB.RegisterWrite, and
  • MEM/WB.Rd = ID/EX.Rt, and
  • ID/EX.Rt != 0
Forwarding may not be enough

- Load word can still cause a hazard:
  - an instruction tries to read a register following a load instruction that writes to the same register.
- Thus, we need a hazard detection unit to "stall" the load instruction

<table>
<thead>
<tr>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
<th>Cycle 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF stage</td>
<td>ID stage</td>
<td>EX stage</td>
<td>MEM stage</td>
</tr>
<tr>
<td>and $7, $8, $3</td>
<td>lw $3, 300($4)</td>
<td>and $7, $8, $3</td>
<td>lw $3, 300($4)</td>
</tr>
</tbody>
</table>

"and" stalls during cycle 4 and picks up the correct value of $3 from MEM/WB at the beginning of cycle 5

"lw" has the correct value of $3 in MEM/WB buffer at the end of cycle 4

Example of the need for stalling the pipe

- The \textit{and} reads the wrong value from $3$ (stored in ID/EX)
- The \textit{lw} does not have the value to be written in $3$
- We have to make sure that the \textit{lw} and the \textit{and} are separated by at least one instruction

- If the \textit{and} and the \textit{lw} are not separated by at least one instruction, then we need to insert a bubble at run time.
Inserting a bubble (stalling the pipe)

How to detect a hazard:
- the instruction in ID/EX is a lw instruction (will read from memory)
- the instruction in ID/EX will write into a register $X
- the instruction in IF/ID will read from register $X

What to do when a hazard is detected:
- freeze the contents of the IF/ID buffer and the PC
- insert a no-op into the ID/EX buffer

Hazard Detection (stalling) Unit

- Stall by letting an instruction that won’t write anything go forward
**Branch/Control Hazards**

- When we decide to branch, other instructions are in the pipeline!

<table>
<thead>
<tr>
<th>IF stage</th>
<th>ID stage</th>
<th>EX stage</th>
<th>MEM stage</th>
<th>WB stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycle 1</td>
<td>add $4, $5, $6</td>
<td>add $4, $5, $6</td>
<td>add $4, $5, $6</td>
<td>add $4, $5, $6</td>
</tr>
<tr>
<td>Cycle 2</td>
<td>beq $1, $2, 10</td>
<td>beq $1, $2, 10</td>
<td>beq $1, $2, 10</td>
<td>beq $1, $2, 10</td>
</tr>
<tr>
<td>Cycle 3</td>
<td>lw $3, 300($0)</td>
<td>lw $3, 300($0)</td>
<td>lw $3, 300($0)</td>
<td>lw $3, 300($0)</td>
</tr>
<tr>
<td>Cycle 4</td>
<td>sub $7, $8, $9</td>
<td>sub $7, $8, $9</td>
<td>sub $7, $8, $9</td>
<td>sub $7, $8, $9</td>
</tr>
<tr>
<td>Cycle 5</td>
<td>lw $3, 300($0) or and $3, $2, $1</td>
<td>lw $3, 300($0) or and $3, $2, $1</td>
<td>lw $3, 300($0) or and $3, $2, $1</td>
<td>lw $3, 300($0) or and $3, $2, $1</td>
</tr>
</tbody>
</table>

- We are predicting “branch not taken”
  - need to add hardware for flushing instructions if we are wrong

**Resolving the branches**

Branch condition and address resolved in the EX stage

Branch address can be resolved in ID stage.
Branch condition can also be resolved in the ID stage if we use a comparator.

- why would this help?
- Does this have any effect on the cycle time?
Resolving the branches

The mux at the input of PC selects the branch PC when
- the control indicates that the instruction in the ID stage is a \textit{beq}
- the zero output of the comparator is true

When the mux selects the branch PC, then input a no-op to the IF/ID buffer (flush)

Flushing Instructions (creating a bubble)