Pipeline Hazards

- Conflicts that prevent the instruction A from executing within its pipeline stage during a given clock cycle:
  1. **Structural hazards** – instruction A requires a resource occupied by some previous instruction
  2. **Data hazards** – a source operand of instruction A is the output of some previous instruction and is not ready
  3. **Control hazards** – pipelining branches and other instructions that change PC conflicts with execution of later instructions (since we don’t know where to read these instructions from)

Performance Degradation Due to Hazards

\[
\text{Speedup} = \frac{T_{\text{per instruction, unpipelined}}}{T_{\text{per instruction, pipelined}}}
\]

\[
\text{CPI}_{\text{unpipelined}} = \frac{\text{CPI}_{\text{unpipelined}}}{\text{CPI}_{\text{pipelined}}} = \frac{1}{\text{Stall/Cycle}_{\text{per instruction}}}
\]

Structural Hazards

- Resource conflicts:
  - E.g. using same memory for data and instructions will create a hazard whenever we have STORE or LOAD. Can you think of another structural hazard?
  - Instructions are stalled until resource becomes available
  - Conflicts can be avoided through resource duplication
  - If structural hazards are not that frequent it may be cheaper to allow them

Example

Suppose that data references constitute 40% of the mix, and that the ideal CPI of the pipelined processor (if we didn’t have the hazard) 1. Assume that the processor with the structural hazard (data and instructions are stored in the same memory) has a clock rate that is 1.05 times higher than the clock rate of the processor where we avoid the hazard through resource duplication. Is the pipeline with or without structural hazard faster, and by how much?
Data Hazards

- Source operand of the current instruction is the output of a previous instruction and is not ready

DADD R1, R2, R3
DSUB R4, R1, R5
AND R6, R1, R7
OR R8, R1, R9
XOR R10, R1, R11

Forwarding

DADD R1, R2, R3
DSUB R4, R1, R5
AND R6, R1, R7
OR R8, R1, R9
XOR R10, R1, R11

Data Hazards

DADD R1, R2, R3
LD R4, 0(R1)
SD R4, 12(R1)

Forwarding

DADD R1, R2, R3
LD R4, 0(R1)
Forwarding can be generalized as passing the result from one functional unit to another unit that needs it. This is done through pipeline registers, not directly.

Data Hazards Requiring Stalls
- Load followed by ALU instructions that use the result
  - LD R1, 0(R2)
  - DSUB R4, R1, R5
  - AND R6, R1, R7
  - OR R8, R1, R9

A piece of hardware called pipeline interlock is added to detect a hazard and stall the pipeline.
Control Hazards

- Conditional branches
  - If the branch is taken (condition is true) next instruction should be fetched from the target address
  - We have necessary data at the end of ID stage (assuming branch target calculation is done in ID stage!!)
  - The next instruction needs the data at the beginning of IF stage

Flush Pipeline

- Freeze pipeline until branch outcome is known

<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>Branch</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>Instruction i+1</td>
<td>IF</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction i+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>

Predict Not Taken

- Treat every branch as not taken
- If taken, turn next instruction into noop

<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>Branch (not taken)</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>Instruction i+1</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>Instruction i+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>

<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>Branch (taken)</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>Instruction i+1</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch target</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>

Delayed Branches

- Compiler reorders instructions so that the next instruction after the branch is:
  - Useful in most cases
  - Harmless both in taken and not taken case

<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>Branch (not taken)</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>Delay instruction</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>Instruction i+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>

<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>Branch (taken)</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>Delay instruction</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>Branch target</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>
Delayed Branch

- From before:
  - DADD R1, R2, R3
  - DSUB R4, R5, R6
  - If R2=0 then
    - DADD R1, R2, R3
    - DSUB R4, R5, R6
- From target:
  - DADD R1, R2, R3
  - DSUB R4, R5, R6
  - If R1=0 then
    - OR R7, R8, R9
    - DSUB R4, R5, R6
- From fall-through:
  - DADD R1, R2, R3
  - DSUB R4, R5, R6
  - If R1=0 then
    - OR R7, R8, R9
    - DSUB R4, R5, R6

Nullifying Branches

- Like delayed branches, but they may be harmful if we mispredicted
- Compiler predicts the outcome of a branch and selects delay instruction
- Compiler adds the prediction data into branch instruction
- If branch behaves as predicted the delay instruction is executed, otherwise it is turned into noop

Performance Degradation Due to Control Hazards

\[
\text{Speedup} = \frac{n}{1 + \text{Branch\_frequency} \times \text{Branch\_penalty}}
\]

Example

Assume a deeper pipeline where it takes 3 pipeline stages until branch target is known and 1 more to evaluate branch condition. Find the effective addition to CPI assuming the following frequencies of branches and the following branch penalties:

<table>
<thead>
<tr>
<th>Branch Type</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unconditional branch</td>
<td>4%</td>
</tr>
<tr>
<td>Conditional branch, untaken</td>
<td>6%</td>
</tr>
<tr>
<td>Conditional branch, taken</td>
<td>10%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pipeline Stage</th>
<th>Unconditional</th>
<th>Untaken</th>
<th>Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flush</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Predict taken</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Predict untaken</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>