Pipeline

Thoai Nam
Outline

- Pipelining concepts
- The DLX architecture
- A simple DLX pipeline
- Pipeline Hazards and Solution to overcome

Reference:

Computer Architecture: A Quantitative Approach, John L Hennessy & David a Patterson, Chapter 6
A technique to make fast CPUs by overlapping execution of multiple instructions

<table>
<thead>
<tr>
<th>Instruction #</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>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction i</td>
<td>S1</td>
<td>S2</td>
<td>S3</td>
<td>S4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction i+1</td>
<td>S1</td>
<td>S2</td>
<td>S3</td>
<td>S4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction i+2</td>
<td>S1</td>
<td>S2</td>
<td>S3</td>
<td>S4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction i+3</td>
<td>S1</td>
<td>S2</td>
<td>S3</td>
<td>S4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction i+4</td>
<td>S1</td>
<td>S2</td>
<td>S3</td>
<td>S4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
- Pipeline throughput
  - Determined by how often an instruction exists the pipeline
  - Depends on the overhead of clock skew and setup
  - Depends on the time required for the slowest pipe stage

- Pipeline stall
  - Delay the execution of some instructions and all succeeding instructions
  - “Slow down” the pipeline

- Pipeline Designer’s goal
  - Balance the length of pipeline stages
  - Reduce / Avoid pipeline stalls

Concepts (cont’d)
Pipeline speedup = \frac{\text{Average instruction time without pipeline}}{\text{Average instruction time with pipeline}}

\[
= \frac{\text{CPI without pipelining} \times \text{Clock cycle without pipelining}}{\text{CPI with pipelining} \times \text{Clock cycle with pipelining}}
\]

\[
= \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall} \times \text{clock cycles per instruction}}
\]

CPI without pipelining = Ideal CPI \times \text{Pipeline depth}

CPI with pipelining = Ideal CPI + Pipeline stall \times \text{clock cycles per instruction}
The DLX Architecture

- A mythical computer which architecture is based on most frequently used primitives in programs
- Used to demonstrate and study computer architecture organizations and techniques
- A DLX instruction consists of 5 execution stages
  - IF – instruction fetch
  - ID – instruction decode and register fetch
  - EX – execution and effective address calculation
  - MEM – memory access
  - WB – write back
A Simple DLX Pipeline

- Fetch a new instruction on each clock cycle
- An instruction step = a pipe stage

Cycles

<table>
<thead>
<tr>
<th>Instruction #</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>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction i</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+1</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>
</tr>
</tbody>
</table>

- Fetch a new instruction on each clock cycle
- An instruction step = a pipe stage
Pipeline Hazards

- Are situations that prevent the next instruction in the instruction stream from executing during its designated cycles
- Leads to pipeline stalls
- Reduce pipeline performance
- Are classified into 3 types
  - Structural hazards
  - Data hazards
  - Control hazards
Structure Hazard

- Due to resource conflicts
- Instances of structural hazards
  - Some functional unit is not fully pipelined
    » a sequence of instructions that all use that unit cannot be sequentially initiated
  - Some resource has not been duplicated enough. Eg:
    » Has only 1 register-file write port while needing 2 write in a cycle
    » Using a single memory pipeline for data and instruction

- Why we allow this type of hazards?
  - To reduce cost.
  - To reduce the latency of the unit
Data Hazard

- Occurs when the order of access to operands is changed by the pipeline, making data unavailable for next instruction

- Example: consider these 2 instructions
  
  \[
  \text{ADD R1, R2, R3} \quad (R2 + R3 \rightarrow R1) \\
  \text{SUB R4, R1, R5} \quad (R1 - R5 \rightarrow R4)
  \]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Data written here

Data read here → instruction is stalled 2 cycles
Hardware Solution to Data Hazard

- Forwarding (bypassing/short-circuiting) techniques
  - Reduce the delay time between 2 depended instructions
  - The ALU result is fed back to the ALU input latches
  - Forwarding hardware check and forward the necessary result to the ALU input for the 2 next instructions

ADD R1, R2, R3

SUB R4, R1, R5

AND R6, R1, R7

OR R8, R1, R9

XOR R1, R10, R11

No stall
No stall
No stall
Types of Data Hazards

- RAW (Read After Write)
  - Instruction j tries to read a source before instruction i writes it
  - Most common types

- WAR (Write After Read)
  - Instruction j tries to write a destination before instruction i read it to execute
  - Can not happen in DLX pipeline. Why?

- WAW (Write After Write)
  - Instruction j tries to write a operand before instruction i updates it
  - The writes end up in the wrong order

- Is RAR (Read After Read) a hazard?
Software Solution to Data Hazard

- Pipeline scheduling (Instruction scheduling)
  - Use compiler to rearrange the generated code to eliminate hazard.
  
  **Example:**

  **Source code**
  
  \[
  c = a + b \\
  d = e - f
  \]

  **Generated code**
  
  \[
  \begin{align*}
  &\text{LW Ra, a} \\
  &\text{LW Rb, b} \\
  &\text{ADD Rc, Ra, Rb} \\
  &\text{SW c, Rc} \\
  &\text{LW Re, e} \\
  &\text{LW Rf, f} \\
  &\text{SUB Rd, Re, Rf} \\
  &\text{SW d, Rd}
  \end{align*}
  \]

  **Generated and rearranged code** (no hazard)
  
  \[
  \begin{align*}
  &\text{LW Ra, a} \\
  &\text{LW Rb, b} \\
  &\text{LW Re, e} \\
  &\text{ADD Rc, Ra, Rb} \\
  &\text{LW Rf, f} \\
  &\text{SW c, Rc} \\
  &\text{SUB Rd, Re, Rf} \\
  &\text{SW d, Rd}
  \end{align*}
  \]
Control/Branch Hazard

- Occurs when a branch/jump instruction is taken
- Causes great performance loss
- Example:

Unnecessary instruction loaded

<table>
<thead>
<tr>
<th></th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction i+1</td>
<td>IF</td>
<td>stall</td>
<td>stall</td>
<td>IF</td>
<td>ID</td>
</tr>
<tr>
<td>Instruction i+2</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td>IF</td>
<td>ID</td>
</tr>
<tr>
<td>Instruction i+3</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td>IF</td>
<td>ID</td>
</tr>
<tr>
<td>Instruction i+4</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td>IF</td>
<td>ID</td>
</tr>
<tr>
<td>Instruction i+5</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td>IF</td>
<td>ID</td>
</tr>
<tr>
<td>Instruction i+6</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td>IF</td>
<td></td>
</tr>
</tbody>
</table>
Reducing Control Hazard Effects

- Predict whether the branch is taken or not
- Compute the branch target address earlier
- Use many schemes
  - Pipeline freezing
  - Predict-not-taken scheme
  - Predict-taken scheme (N/A in DLX)
  - Delayed branch
Pipeline Freezing

- Hold any instruction after the branch until the branch destination is known
- Simple but not efficient
Predict-Not-Taken Scheme

- Predict the branch as not taken and allow execution to continue
  - Must not change the machine state till the branch outcome is known
- If the branch is not taken: no penalty
- If the branch is taken:
  - Restart the fetch at the branch target
  - Stall one cycle
## Predict-Not-Taken Scheme (cont’d)

**Example**

<table>
<thead>
<tr>
<th>Taken branch instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction i+1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction i+2</td>
<td>stall</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
</tr>
<tr>
<td>Instruction i+3</td>
<td>stall</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
</tr>
<tr>
<td>Instruction i+4</td>
<td>stall</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
</tr>
</tbody>
</table>

Instruction Fetch restarted

Right instruction fetched
Branch Delayed

- Change the order of execution so that the next instruction is always valid and useful
- “From before” approach

```
ADD R1, R2, R3
If R2=0 then
  Delay slot
```

becomes

```
If R2=0 then
  ADD R1, R2, R3
```
Branch Delayed (cont’d)

- “From target” approach

SUB R4, R5, R6
ADD R1, R2, R3
If R1=0 then
Delay slot

becomes

ADD R1, R2, R3
If R1=0 then
SUB R4, R5, R6
“From fall through” approach

ADD R1, R2, R3
If R1=0 then
Delay slot
SUB R4, R5, R6

becomes

ADD R1, R2, R3
If R1=0 then
SUB R4, R5, R6