Lecture 4: Introduction to Pipelining
Pipelining

- Laundry Example
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 40 minutes
- “Folder” takes 20 minutes
Sequential Laundry

- Sequential laundry takes 6 hours for 4 loads
• Pipelined laundry takes 3.5 hours for 4 loads
Pipelining: Observations

- Pipelining doesn’t help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to “fill” pipeline and time to “drain” it reduces speedup
5 Steps of DLX Datapath

Figure 3.1
Pipelined DLX Datapath

Figure 3.4
Visualizing Pipelining

Figure 3.3

Time (clock cycles)
Limits to Pipelining

• **Hazards** prevent next instruction from executing during its designated clock cycle
  – **Structural hazards**: HW cannot support this combination of instructions
  – **Data hazards**: Instruction depends on result of prior instruction still in the pipeline
  – **Control hazards**: Pipelining of branches & other instructions that change the PC

• **Common solution is to stall** the pipeline until the hazard is resolved, inserting one or more “bubbles” in the pipeline
One Memory Port/Structural Hazards

Figure 3.6

Time (clock cycles)

Load

Instr 1

Instr 2

Instr 3

Instr 4
One Memory Port/Structural Hazards

Figure 3.7

- Load
- Instr 1
- Instr 2
- Instr 3

*Time (clock cycles)*
Speed Up Equation for Pipelining

\[
\text{Speedup from pipelining} = \frac{\text{Ave Instr Time unpipelined}}{\text{Ave Instr Time pipelined}}
\]
\[
= \frac{\text{CPI}_{\text{unpipelined}} \times \text{Clock Cycle}_{\text{unpipelined}}}{\text{CPI}_{\text{pipelined}} \times \text{Clock Cycle}_{\text{pipelined}}}
\]
\[
= \frac{\text{CPI}_{\text{unpipelined}}}{\text{CPI}_{\text{pipelined}}} \times \frac{\text{Clock Cycle}_{\text{unpipelined}}}{\text{Clock Cycle}_{\text{pipelined}}}
\]

\[
\text{Ideal CPI} = \frac{\text{CPI}_{\text{unpipelined}}}{\text{Pipeline depth}}
\]

\[
\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth} \times \text{Clock Cycle}_{\text{unpipelined}}}{\text{CPI}_{\text{pipelined}} \times \text{Clock Cycle}_{\text{pipelined}}}
\]
Speed Up Equation for Pipelining

\[
\text{CPI}_{\text{pipelined}} = \text{Ideal CPI} + \text{Pipeline stall clock cycles per instr}
\]

\[
\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \times \frac{\text{Clock Cycle}_{\text{unpipelined}}}{\text{Clock Cycle}_{\text{pipelined}}}
\]

\[
\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Clock Cycle}_{\text{unpipelined}}}{\text{Clock Cycle}_{\text{pipelined}}}
\]
Example: Dual-port vs. Single-port

- Machine A: Dual ported memory
- Machine B: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate
- Ideal CPI = 1 for both
- Loads are 40% of instructions executed

\[
\text{SpeedUp}_A = \frac{\text{Pipeline Depth}}{1 + 0} \times \left(\frac{\text{clock}_{\text{unpipe}}}{\text{clock}_{\text{pipe}}}\right) \\
= \text{Pipeline Depth}
\]

\[
\text{SpeedUp}_B = \frac{\text{Pipeline Depth}}{1 + 0.4 \times 1} \times \left(\frac{\text{clock}_{\text{unpipe}}}{\left(\frac{\text{clock}_{\text{unpipe}}}{1.05}\right)}\right) \\
= \frac{\text{Pipeline Depth}}{1.4} \times 1.05 \\
= 0.75 \times \text{Pipeline Depth}
\]

\[
\frac{\text{SpeedUp}_A}{\text{SpeedUp}_B} = \frac{\text{Pipeline Depth}}{(0.75 \times \text{Pipeline Depth})} = 1.33
\]

- Machine A is 1.33 times faster
Data Hazard on R1

Figure 3.9

```
add r1, r2, r3
sub r4, r1, r3
and r6, r1, r7
or r8, r1, r9
xor r10, r1, r11
```
Three Generic Data Hazards

Instr\(_i\) followed by Instr\(_j\)

• Read After Write (RAW)
  Instr\(_j\) tries to read operand before Instr\(_i\) writes it
Three Generic Data Hazards

$\text{Instr}_i$ followed by $\text{Instr}_j$

- **Write After Read (WAR)**
  $\text{Instr}_j$ tries to write operand before $\text{Instr}_i$ reads it

- Can’t happen in DLX 5 stage pipeline because:
  - All instructions take 5 stages,
  - Reads are always in stage 2, and
  - Writes are always in stage 5
Three Generic Data Hazards

Instr\textsubscript{i} followed by Instr\textsubscript{j}

- **Write After Write (WAW)**
  Instr\textsubscript{j} tries to write operand before Instr\textsubscript{i} writes it
  - Leaves wrong result (Instr\textsubscript{i} not Instr\textsubscript{j})

- Can’t happen in DLX 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Writes are always in stage 5

- Will see WAR and WAW in later more complicated pipes
Forwarding to Avoid Data Hazard

Figure 3.10

- add r1, r2, r3
- sub r4, r1, r3
- and r6, r1, r7
- or r8, r1, r9
- xor r10, r1, r11
HW Change for Forwarding

Figure 3.20
Data Hazard Even with Forwarding

Figure 3.12

Time (clock cycles)

lw r1, 0(r2)
sub r4, r1, r6
and r6, r1, r7
or r8, r1, r9
Data Hazard Even with Forwarding

Figure 3.13

Instruction Order

lw r1, 0(r2)
sub r4, r1, r6
and r6, r1, r7
or r8, r1, r9
Software Scheduling to Avoid Load Hazards

Try producing fast code for

\[
\begin{align*}
a &= b + c; \\
d &= e - f;
\end{align*}
\]
assuming a, b, c, d, e, and f in memory.

<table>
<thead>
<tr>
<th>Slow code:</th>
<th>Fast code:</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW Rb,b</td>
<td>LW Rb,b</td>
</tr>
<tr>
<td>LW Rc,c</td>
<td>LW Rc,c</td>
</tr>
<tr>
<td>ADD Ra,Rb,Rc</td>
<td>LW Re,e</td>
</tr>
<tr>
<td>SW a,Ra</td>
<td>ADD Ra,Rb,Rc</td>
</tr>
<tr>
<td>LW Re,e</td>
<td>LW Rf,f</td>
</tr>
<tr>
<td>LW Rf,f</td>
<td>SW a,Ra</td>
</tr>
<tr>
<td>SUB Rd,Re,Rf</td>
<td>SUB Rd,Re,Rf</td>
</tr>
<tr>
<td>SW d,Rd</td>
<td>SW d,Rd</td>
</tr>
</tbody>
</table>
Compiler Avoiding Load Stalls

<table>
<thead>
<tr>
<th></th>
<th>scheduled</th>
<th>unscheduled</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>31%</td>
<td>54%</td>
</tr>
<tr>
<td>spice</td>
<td>14%</td>
<td>42%</td>
</tr>
<tr>
<td>tex</td>
<td>25%</td>
<td>65%</td>
</tr>
</tbody>
</table>
Pipelining Summary

• Just overlap tasks, and easy if tasks are independent
• Speed Up vs Pipeline Depth; if ideal CPI is 1, then:

\[
\text{Speedup} = \frac{\text{Pipeline Depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Clock Cycle Unpipelined}}{\text{Clock Cycle Pipelined}}
\]

• Hazards limit performance on computers:
  – Structural: need more HW resources
  – Data: need forwarding, compiler scheduling
  – Control: discuss next time