Pipelined instruction Execution

Pipelining and ISA Design

- MIPS Instruction Set designed for pipelining
- All instructions are 32-bits
  - Easier to fetch and decode in one cycle
  - x86: 1- to 17-byte instructions
    (x86 HW actually translates to internal RISC instructions!)
- Few and regular instruction formats, 2 source register fields always in same place
  - Can decode and read registers in one step
- Memory operands only in Loads and Stores
  - Can calculate address 3rd stage, access memory 4th stage
- Alignment of memory operands
  - Memory access takes only one cycle
Pipelined Control

• Every instruction must take same number of steps, also called pipeline “stages”, so some will go idle sometimes
1. Instruction Fetch
2. Decode/Register Read
3. Execute
4. Memory
5. Write Back

• Use datapath figure below to represent pipeline

**Graphical Pipeline Representa:**

*Time (clock cycles)*

**Instr. Order**

<table>
<thead>
<tr>
<th>Load</th>
<th>Add</th>
<th>Store</th>
<th>Sub</th>
<th>Or</th>
</tr>
</thead>
</table>

(In Reg, right half highlight read, left half write)
## Pipeline Performance

- Assume time for stages is:
  - 100ps for register read or write
  - 200ps for other stages

- What is pipelined clock rate?
  - Compare pipelined datapath with single-cycle datapath

<table>
<thead>
<tr>
<th>Instr</th>
<th>Instr fetch</th>
<th>Register read</th>
<th>ALU op</th>
<th>Memory access</th>
<th>Register write</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td>100 ps</td>
<td>800ps</td>
</tr>
<tr>
<td>sw</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td></td>
<td>700ps</td>
</tr>
<tr>
<td>R-format</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td>100 ps</td>
<td>600ps</td>
</tr>
<tr>
<td>beq</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td></td>
<td>500ps</td>
</tr>
</tbody>
</table>
Pipeline Speedup

• If all stages are balanced
  • i.e., all take the same time
  • Time between instructions_{pipelined} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}
• If not balanced, speedup is less
• Speedup due to increased throughput
  • Latency (time for each instruction) does not decrease

Hazards

Situations that prevent starting the next logical instruction in the next clock cycle
1. Structural hazards
   – Required resource is busy
2. Data hazard
   – Need to wait for previous instruction to complete its data read/write (e.g., pair of socks in different loads)
3. Control hazard
   – Deciding on control action depends on previous instruction (e.g., how much detergent based on how clean prior load turns out)
1. Structural Hazards

- Conflict for use of a resource
- In MIPS pipeline with a single memory
  - Load/Store requires memory access for data
  - Instruction fetch would have to *stall* for that cycle
    - Causes a pipeline “bubble”
- Hence, pipelined datapaths require separate instruction/data memories
  - In reality, provide separate L1 I$ and L1 D$

1. Structural Hazard #1: Single Memory

<table>
<thead>
<tr>
<th>Instr Order</th>
<th>Instr 1</th>
<th>Instr 2</th>
<th>Instr 3</th>
<th>Instr 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>IS</td>
<td>IS</td>
<td>IS</td>
<td>IS</td>
</tr>
<tr>
<td></td>
<td>Reg</td>
<td>Reg</td>
<td>Reg</td>
<td>Reg</td>
</tr>
<tr>
<td></td>
<td>D$</td>
<td>D$</td>
<td>D$</td>
<td>D$</td>
</tr>
<tr>
<td></td>
<td>Reg</td>
<td>Reg</td>
<td>Reg</td>
<td>Reg</td>
</tr>
</tbody>
</table>

Read same memory twice in same clock cycle
1. Structural Hazard #2: Registers (1/2)

Can we read and write to registers simultaneously?

1. Structural Hazard #2: Registers (2/2)

- Two different solutions have been used:
  1) RegFile access is *VERY* fast: takes less than half the time of ALU stage
     - Write to Registers during first half of each clock cycle
     - Read from Registers during second half of each clock cycle
  2) Build RegFile with independent read and write ports
- Result: can perform Read and Write during same clock cycle
Data Hazards (1/2)

Consider the following sequence of instructions

- add $t0, $t1, $t2
- sub $t4, $t0, $t3
- and $t5, $t0, $t6
- or $t7, $t0, $t8
- xor $t9, $t0, $t10

Data Hazards (2/2)

- Data-flow backward in time are hazards

**Time (clock cycles)**

---

**Instr. Order**

- add $t0,$t1,$t2
- sub $t4,$t0,$t3
- and $t5,$t0,$t6
- or $t7,$t0,$t8
- xor $t9,$t0,$t10

---
Data Hazard Solution: Forwarding

- Forward result from one stage to another

\[
\begin{align*}
\text{add } & \text{ } \texttt{st0}, \texttt{t1}, \texttt{t2} \\
\text{sub } & \text{ } \texttt{st4}, \texttt{st0}, \texttt{t3} \\
\text{and } & \text{ } \texttt{st5}, \texttt{st0}, \texttt{t6} \\
\text{or } & \text{ } \texttt{st7}, \texttt{st0}, \texttt{t8} \\
\text{xor } & \text{ } \texttt{st9}, \texttt{st0}, \texttt{t10} \\
\end{align*}
\]

“\text{x}” hazard solved by register hardware

Data Hazard: Load/Use (1/4)

- Dataflow backwards in time are hazards

\[
\begin{align*}
\text{lw } & \text{ } \texttt{st0}, \texttt{0}(\texttt{t1}) \\
\text{sub } & \text{ } \texttt{st3}, \texttt{st0}, \texttt{t2} \\
\end{align*}
\]

- Can’t solve all cases with forwarding
- Must stall instruction dependent on load, then forward (more hardware)
Data Hazard: Load/Use (2/4)

Hardware stalls pipeline (Called “interlock”)

- `lw $t0, 0($t1)`
- `sub $t3,$t0,$t2`
- `and $t5,$t0,$t4`
- `or $t7,$t0,$t6`

Not in MIPS: (MIPS = Microprocessor without Interlocked Pipeline Stages)

Data Hazard: Load/Use (3/4)

- Instruction slot after a load is called “load delay slot”
- If that instruction uses the result of the load, then the hardware interlock will stall it for one cycle.
- Alternative: If the compiler puts an unrelated instruction in that slot, then no stall
- Letting the hardware stall the instruction in the delay slot is equivalent to putting a nop in the slot (except the latter uses more code space)
Data Hazard: Load/Use (4/4)

- Stall is equivalent to `nop`

\[
lw \ $t0, \ 0($t1) \\
\text{nop} \\
\text{sub} \ $t3,$t0,$t2 \\
\text{and} \ $t5,$t0,$t4 \\
or \ $t7,$t0,$t6
\]

Data Hazards: Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for \( A = B + E; \ C = B + F; \)

\[
\text{lw} \ $t1, \ 0($t0) \\
\text{lw} \ $t2, \ 4($t0) \\
\text{add} \ $t3, \ $t1, \ $t2 \\
\text{sw} \ $t3, \ 12($t0) \\
\text{lw} \ $t4, \ 8($t0) \\
\text{add} \ $t3, \ $t1, \ $t2 \\
\text{sw} \ $t3, \ 12($t0) \\
\text{lw} \ $t4, \ 8($t0) \\
\text{add} \ $t5, \ $t1, \ $t4 \\
\text{sw} \ $t5, \ 16($t0)
\]

- 13 cycles
- 11 cycles
## Data Hazards: Code Scheduling

<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>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>lw $t1, 0($t0)</strong></td>
<td>F</td>
<td>D</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>lw $t2, 4($t0)</strong></td>
<td>F</td>
<td>D</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>add $t3, $t1, $t2</strong></td>
<td>F</td>
<td>D</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>sw $t3, 12($t0)</strong></td>
<td>F</td>
<td>D</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>lw $t4, 8($t0)</strong></td>
<td>F</td>
<td>D</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>add $t5, $t1, $t4</strong></td>
<td>F</td>
<td>D</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>sw $t5, 16($t0)</strong></td>
<td>F</td>
<td>D</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></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>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>lw $t1, 0($t0)</strong></td>
<td>F</td>
<td>D</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>lw $t2, 4($t0)</strong></td>
<td>F</td>
<td>D</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>lw $t4, 8($t0)</strong></td>
<td>F</td>
<td>D</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>add $t3, $t1, $t2</strong></td>
<td>F</td>
<td>D</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>sw $t3, 12($t0)</strong></td>
<td>F</td>
<td>D</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>add $t5, $t1, $t4</strong></td>
<td>F</td>
<td>D</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>sw $t5, 16($t0)</strong></td>
<td>F</td>
<td>D</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>