Computer Architecture:
ILP: Pipelining

Berk Sunar       Thomas Eisenbarth
ECE 505
Outline

• Pipelining
  – 5 stages of RISC/MIPS pipeline
  – Types of hazards
Last Time:
Review of Instruction Set Architecture

Review of Instruction Set Architecture

1. Classification of ISA
2. How to analyze Instruction Set architecture
3. Example: MIPS ISA
Last Time: MIPS Architecture

- RISC Architecture (Reduced Instruction Set Computer)
- Load-Store Architecture
- 32 Registers, with R0 is always 0
- All Instructions are one size (32 bits)
- Three classes:
  - ALU Instructions
  - Load and Store Instructions
  - Conditional Instructions

<table>
<thead>
<tr>
<th>OP</th>
<th>R3</th>
<th>R2</th>
<th>R1</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OP</th>
<th>R2</th>
<th>R1</th>
<th>IMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Pipelining: Concept

- All objects go through same stages
- No shared resources between any two stages
- Latency of all stages is equal
  \(\rightarrow\) Stages can be processed independently of each other

- Conditions often hold for industry assembly lines
  However, instructions depend on each other causing various hazards
Pipelining for Processors

**Assumption**: instruction processing can be broken into \( N \) sequential stages

**Idea**: overlap different instructions at different stages

Sequential execution:

```
Inst 1.1  Inst 1.2  Inst 1.3  Inst 2.1  Inst 2.2  Inst 2.3
```

Pipelined execution:

```
Inst 1.1  Inst 1.2  Inst 1.3  Inst 2.1  Inst 2.2  Inst 2.3
```

+ increase resource utilization: fewer stages sitting idle
+ increase completion rate (throughput): up to 1 in \( 1/N \) time

- Almost all modern processors since 1970 are pipelined
Five parts of instruction execution:

- **Instruction Fetch (IF):** Fetch instruction from memory
- **Instruction Decode (ID):** Decode the instruction and read the registers; Prepare for branch
- **Execution (EX):** Execute the operation (ALU) or ld/st address/branch outcome
- **Memory Access (MEM):** Load/Store to Memory
- **Write-Back (WB):** Write result (ALU or ld) into a register
Five stages of instruction execution (pipeline depth is 5)

- **Instruction Fetch (IF):** Fetch instruction from memory
- **Instruction Decode (ID):** Decode the instruction and read the registers; Prepare for branch
- **Execution (EX):** Execute the operation (ALU) or ld/st address/branch outcome
- **Memory Access (MEM):** Load/Store to Memory
- **Write-Back (WB):** Write result (ALU or ld) into a register

→ Stages divided by pipeline registers
Compared to non-pipelined case:
- Better throughput: an instruction finishes every cycle
- Same latency per instruction: each still takes 5 cycles
5-Stages of classic RISC

• Typical Instructions
  \text{DADD R1, R2, R3}
  \text{IF – ID – EX – MEM – WB}

• Store Instructions
  \text{SW R3, 500(R4)}
  \text{IF – ID – EX – MEM}

• Branch Instructions
  \text{BEQZ R4, name}
  \text{IF – ID}
Why Pipelining?

DADD R1, R2, R3
DSUB R4, R5, R6
AND R7, R8, R9
Pipelining

- Instruction Level Parallelism

DADD R1, R2, R3
DSUB R4, R5, R6
AND R7, R8, R9
Performance Limits

• Instruction latency is ideally the same.
• Instruction throughput increases.
• **Ideally:**
  Speedup = Pipeline depth (N)

– Practically, instruction latency is worse
  • Not all instructions require 5 stages
  • Clock runs at the slowest stage (effect of imbalanced stages)
  • The delay of the new registers
  • Clock Skew
  • [[ HAZARDS ]]
Technology Assumptions

Let instruction execution require $N$ stages, each takes $t\downarrow n$

- Pipelined processor has speedup $\leq N$

Assuming each stage takes similar time $t\downarrow n$ is ok given

- A small amount of very fast memory (caches) backed up by a large, slower memory
- Fast ALU (at least for integers)
- Multiported Register files (slower!)

- Note that some commercial designs have over 30 pipeline stages to do an integer add!
Performance Limits

Example

1 ns clock cycle
35% Branches (2 cycles)
25% Store operations (4 cycles)
40% ALU operations (5 cycles)

Pipelining adds 0.2 ns of overhead to the clock

Calculate the speedup from Pipelining
Performance Limits

• Example
  Unpipelined Architecture:
  Average instruction execution time = 
  \[ \text{Clock cycle} \times \text{Average CPI} \]
  \[= 1 \text{ ns} \times (0.35 \times 2) + (0.25 \times 4) + (0.4 \times 5) \]
  \[= 3.7 \text{ ns} \]

  Pipelined Architecture:
  Average instruction execution time = 1.2 ns

  Speedup = 3.7 / 1.2 = 3.08 times (instead of 5!)
Instructions Interact With Each Other in Pipeline

• **Structural Hazard:** An instruction in the pipeline needs a resource being used by another instruction in the pipeline

• **Data Hazard:** An instruction depends on a data value produced by an earlier instruction

• **Control Hazard:** Whether or not an instruction should be executed depends on a control decision made by an earlier instruction
Overview of Structural Hazards

• Structural hazards occur when two instructions need the same hardware resource at the same time
• Approaches to resolving structural hazards
  – **Schedule:** Programmer/compiler explicitly avoids scheduling instructions that would create structural hazards
  – **Stall:** Hardware includes control logic that stalls until earlier instruction is no longer using contended resource
  – **Duplicate:** Add more hardware to design so that each instruction can access independent resources at the same time
• Simple 5-stage MIPS pipeline has no structural hazards specifically because ISA was designed that way
Structural Hazards

Assuming using a generic memory for both instruction and data
Structural Hazards

Solutions:

• Use separate registers (extra cost), or

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Clock Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction $i$</td>
<td>1 2 3 4 5 6 7 8 9 10</td>
</tr>
<tr>
<td>Instruction $i+1$</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>Instruction $i+2$</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>Instruction $i+3$</td>
<td>stall IF ID EX MEM WB</td>
</tr>
<tr>
<td>Instruction $i+4$</td>
<td>IF ID EX MEM WB</td>
</tr>
</tbody>
</table>
Performance Hit

• Ideally without stalls:

  Speedup = Pipeline depth (N)

– Ideally with stalls:

  \[
  \text{Speedup} = \frac{\text{Pipeline depth (N)}}{1 + \text{Average number of stalls per instruction}}
  \]

- Practically with stalls?
Data Hazards

• Is there any Problem?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>DADD</td>
<td>R1, R2, R3</td>
</tr>
<tr>
<td>DSUB</td>
<td>R4, R1, R5</td>
</tr>
<tr>
<td>AND</td>
<td>R6, R1, R7</td>
</tr>
<tr>
<td>OR</td>
<td>R8, R1, R9</td>
</tr>
<tr>
<td>XOR</td>
<td>R10, R1, R11</td>
</tr>
</tbody>
</table>
Data Hazards

• Typically,

Any Simple Solution?
Data Hazards

• Solution: Forwarding
Data Hazards

• How about this?

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>R1, 0(R2)</td>
</tr>
<tr>
<td>DSUB</td>
<td>R4, R1, R5</td>
</tr>
<tr>
<td>AND</td>
<td>R6, R1, R7</td>
</tr>
<tr>
<td>OR</td>
<td>R8, R1, R9</td>
</tr>
</tbody>
</table>

Why it is different?
Data Hazards

• **Forwarding can solve **\textbf{AND}, but not **\textbf{DSUB}
Data Hazards

- This problem requires stall

Before

<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>LD R1, 0(R2)</td>
<td></td>
<td></td>
<td></td>
<td>MEM</td>
<td></td>
</tr>
<tr>
<td>DSUB R4, R1, R5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>AND R6, R1, R7</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>OR R8, R1, R9</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

After

<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>LD R1, 0(R2)</td>
<td></td>
<td></td>
<td></td>
<td>MEM</td>
<td></td>
</tr>
<tr>
<td>DSUB R4, R1, R5</td>
<td>IF</td>
<td>ID</td>
<td>stall</td>
<td>EX</td>
<td>MEM</td>
</tr>
<tr>
<td>AND R6, R1, R7</td>
<td>IF</td>
<td>ID</td>
<td>stall</td>
<td>ID</td>
<td>EX</td>
</tr>
<tr>
<td>OR R8, R1, R9</td>
<td>stall</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
</tr>
</tbody>
</table>
Data Hazards Summary

Data hazards occur when one instruction depends on a data value produced by a preceding instruction still in the pipeline

Approaches to resolving data hazards

- **Schedule:** Programmer explicitly avoids scheduling instructions that would create data hazards
- **Bypass:** Hardware datapath allows values to be sent to an earlier stage before preceding instruction has left the pipeline
- **Stall:** Hardware includes control logic that freezes earlier stages until preceding instruction has finished producing data value
- **Speculate:** Guess that there is not a problem, if incorrect kill speculative instruction and restart
Outlook

Next Time:

More on Pipelining and Hazards