Quiz 2 Review

6.823 Spring 2016

Suvinay Subramanian
Topics Snapshot

» Out-of-order execution
  - Reorder buffer (ROB)
  - Register renaming
  - Branch prediction
  - Store, load buffer

» Multi-threading
  - Coarse-grained, fine-grained, SMT

» Reliability
  - ACE, AVF
Out-of-order Execution

» Instruction-level parallelism (ILP)
  - Execute as many instructions as possible concurrently
  - Dynamic scheduling of instructions
  - Maximize throughput, hide latency (ALU/Memory)

» Why is this not easy?
  - Dependencies between instructions
Flavors of Dependencies

» Data dependency
  - RAW, WAR, WAW

» Control dependency
  - Branches

» Structural dependency
  - Only one ALU
Recipes

1. Stall
2. Bypass
3. Speculate  Modern processors do this a lot!
4. Add hardware
5. Indirection (for false dependencies)
6. Do something else
Resolving Dependencies

» Data dependency
  - RAW, WAR, WAW
    => Scoreboard (L8), Register renaming (L9), ROB (L11), Store buffer (L12)

» Control dependency
  - Branches
    => Branch prediction (L10)

» Structural dependency
  - Only one ALU
    => Superscalar execution (L8, L11)
Reorder Buffer

» Hardware structure that:
   - tracks dependencies between instructions
   - causes correct sequencing of dependent instructions

Enables independent sequences of instructions to proceed concurrently

<table>
<thead>
<tr>
<th>Use</th>
<th>Ex</th>
<th>Op</th>
<th>p1</th>
<th>PR1</th>
<th>p2</th>
<th>PR2</th>
<th>Rd</th>
<th>PRd</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Register Renaming

Eliminates false dependencies by allocating a new register on each write

**Anti-dependence**

\[
\begin{align*}
    r_3 & \leftarrow (r_1) \text{ op } (r_2) \\
    r_1 & \leftarrow (r_4) \text{ op } (r_5)
\end{align*}
\]

Write-after-Read (WAR) hazard

**Output-dependence**

\[
\begin{align*}
    r_3 & \leftarrow (r_1) \text{ op } (r_2) \\
    r_3 & \leftarrow (r_6) \text{ op } (r_7)
\end{align*}
\]

Write-after-Write (WAW) hazard
Register Renaming

Eliminates false dependencies by allocating a new register on each write

Free List | P8 | P12

<table>
<thead>
<tr>
<th>Reg</th>
<th>Value</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>8000</td>
<td>1</td>
</tr>
<tr>
<td>P3</td>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>P4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P5</td>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>P6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Rename Table

<table>
<thead>
<tr>
<th>Register</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>P6</td>
</tr>
<tr>
<td>R2</td>
<td>P2</td>
</tr>
<tr>
<td>R3</td>
<td>P3</td>
</tr>
<tr>
<td>R4</td>
<td>P5</td>
</tr>
<tr>
<td>R5</td>
<td></td>
</tr>
</tbody>
</table>
Store, Load Buffer

» Handle dependencies that arise through memory

» Allow aggressive load scheduling; stores don’t constrain load scheduling

\[
\text{st r1, 4(r2)} \\
\text{ld r3, 8(r4)}
\]

When is the load dependent on the store?

\[
\text{When (r2 + 4) == (r4 + 8)}
\]

Do we know this issue when the instruction is decoded? No
**Store Buffer**

- **On store execute:**
  - mark valid and speculative; save tag, data and instruction number.

- **On store commit:**
  - clear speculative bit and eventually move data to cache

- **On store abort:**
  - clear valid bit

---

<table>
<thead>
<tr>
<th>Store Address</th>
<th>Tags</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store Commit Path</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Enables data forwarding
- Handles OoO stores
- Handles speculative stores

- One entry per store
- Written by stores
- Searched by loads
- Writes to data cache

6.823 Spring 2016
Load Buffer

» On load execute:
   - mark entry valid, and instruction number and tag of data.

» On load commit:
   - clear valid bit

» On load abort:
   - clear valid bit

» One entry per load
   » Enables aggressive load scheduling

» Written by loads
   » Detects ordering violations

» Searched by stores

Speculative Load Buffer

<table>
<thead>
<tr>
<th>Load Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
</tr>
<tr>
<td>Inum</td>
</tr>
<tr>
<td>Tag</td>
</tr>
<tr>
<td>Inum</td>
</tr>
<tr>
<td>Tag</td>
</tr>
<tr>
<td>Inum</td>
</tr>
<tr>
<td>Tag</td>
</tr>
<tr>
<td>Inum</td>
</tr>
<tr>
<td>Tag</td>
</tr>
</tbody>
</table>
Branch Prediction

Speculation on control flow dependencies

» 1-bit predictor, 2-bit predictor
» Global history, local history
» 2-level predictor
Branch Prediction

Branch Target Buffer (BTB)

- store target PC of branch/jmp instruction seen last time
- get address earlier than predictor/decode stage in pipeline

BHT in later pipeline stage corrects when BTB misses a predicted taken branch

BTB/BHT only updated after branch resolves in E stage
Speculative Value Management

» When do we do speculation?
   - Branch prediction
   - Assume no exceptions/interrupts
   - Assume no memory dependency
   - ...

» How do we manage speculative values?
   - Greedy (or eager) update
     - Update value in place
     - Maintain log of old values to use for recovery
   - Lazy update
     - Buffer the new value and leave old value in place
     - Replace the old value on commit
Types of Speculative Values

» Branch Prediction
   - history registers, prediction counters etc.

» Register values

» Memory values
Out-of-order Execution

» Dynamic scheduling of instructions
  - Dynamically builds a restricted data-flow graph of program

» Allows us to tolerate long-latency operations (memory, ALU)
  - Execute “around” long-latency operations

» Adds design complexity, area, power
Topics Snapshot

» Out-of-order execution
  - Reorder buffer (ROB)
  - Register renaming
  - Branch prediction
  - Store, load buffer

» Multi-threading
  - Coarse-grained, fine-grained, SMT

» Reliability
  - ACE, AVF
Multithreading

» Take instructions from different programs (or threads) – guaranteed to be independent.

» Coarse-grained multithreading
  Fine-grained multithreading
  Simultaneous multithreading (SMT)
Reliability

» What happens if a bit flips?
   - Silent Data Corruption (SDC)
   - Detected Unrecoverable Error (DUE)

» What structures are vulnerable?
   - Architectural Vulnerability Factor (AVF)
ACE Lifetime Analysis

» Divide up the lifetime of a bit during a program execution into ACE and un-ACE intervals.

» Start with conservative assumption: Entire period is ACE. Systematically reason about which intervals are un-ACE.

» AVF is the fraction of lifetime a bit is ACE. Typically report per-structure AVF.
Good luck!