Quiz 2 Review

6.823 Spring 2017

Mark Jeffrey

Adapted from: Suvinay Subramanian, 2016
Topics Snapshot

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

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

Reorder Buffer

<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>
</tbody>
</table>

Instruction

Source

Destination
Register Renaming

Eliminates false dependencies by allocating a new register on each write

<table>
<thead>
<tr>
<th>Anti-dependence</th>
<th>Output-dependence</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_3 \leftarrow (r_1) \text{ op } (r_2) )</td>
<td>( r_3 \leftarrow (r_6) \text{ op } (r_7) )</td>
</tr>
<tr>
<td>( r_1 \leftarrow (r_4) \text{ op } (r_5) )</td>
<td>Write-after-Write (WAW) hazard</td>
</tr>
<tr>
<td>Write-after-Read (WAR) hazard</td>
<td></td>
</tr>
</tbody>
</table>
Register Renaming

Eliminates false dependencies by allocating a new register on each write
Register Renaming

Eliminates false dependencies by allocating a new register on each write
Store, Load Buffer

» Handle dependencies that arise through memory

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

\[
\text{st } r1, 4(r2) \quad \text{When is the load dependent on the store?}
\]

\[
\text{ld } r3, 8(r4)
\]

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

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

» Enables data forwarding
» Handles OoO stores
» Handles speculative stores

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

L1 Data Cache

<table>
<thead>
<tr>
<th>Store Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
</tr>
<tr>
<td>V</td>
</tr>
<tr>
<td>V</td>
</tr>
<tr>
<td>V</td>
</tr>
<tr>
<td>V</td>
</tr>
<tr>
<td>V</td>
</tr>
<tr>
<td>V</td>
</tr>
</tbody>
</table>
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

» Written by loads

» Searched by stores

» Enables aggressive load scheduling

» Detects ordering violations

Speculative Load Buffer

Load Address

<table>
<thead>
<tr>
<th>Inum</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>V Inum</td>
<td>Tag</td>
</tr>
<tr>
<td>V Inum</td>
<td>Tag</td>
</tr>
<tr>
<td>V Inum</td>
<td>Tag</td>
</tr>
<tr>
<td>V Inum</td>
<td>Tag</td>
</tr>
<tr>
<td>V Inum</td>
<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
Little’s Law

Throughput ($T$) = Number in Flight ($N$) / Latency ($L$)

Example:

4 floating point registers
8 cycles per floating point operation

⇒ ½ issues per cycle!
Topics Snapshot

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

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

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

- Coarse-grained multithreading
  - Fine-grained multithreading
  - Simultaneous multithreading (SMT)
Good luck!