Handling Out-of-Order Execution of Loads and Stores
Registers versus Memory

- So far, we considered mainly registers as part of state

- What about memory?

- What are the fundamental differences between registers and memory?
  - Register dependences known statically – memory dependences determined dynamically
  - Register state is small – memory state is large
  - Register state is not visible to other threads/processors – memory state is shared between threads/processors (in a shared memory multiprocessor)
Memory Dependence Handling (I)

- Need to obey memory dependences in an out-of-order machine
  - and need to do so while providing high performance

- Observation and Problem: Memory address is not known until a load/store executes

- Corollary 1: Renaming memory addresses is difficult
- Corollary 2: Determining dependence or independence of loads/stores has to be handled after their (partial) execution
- Corollary 3: When a load/store has its address ready, there may be older/younger stores/loads with unknown addresses in the machine
When do you schedule a load instruction in an OOO engine?

- Problem: A younger load can have its address ready before an older store’s address is known
- Known as the memory disambiguation problem or the unknown address problem

Approaches

- **Conservative**: Stall the load until all previous stores have computed their addresses (or even retired from the machine)
- **Aggressive**: Assume load is independent of unknown-address stores and schedule the load right away
- **Intelligent**: Predict (with a more sophisticated predictor) if the load is dependent on any unknown address store
Handling of Store-Load Dependences

- **A load’s dependence status is not known** until all previous store addresses are available.

How does the OOO engine detect dependence of a load instruction on a previous store?
- Option 1: Wait until all previous stores committed (no need to check for address match)
- Option 2: Keep a list of pending stores in a store buffer and check whether load address matches a previous store address

How does the OOO engine treat the scheduling of a load instruction wrt previous stores?
- Option 1: Assume load dependent on all previous stores
- Option 2: Assume load independent of all previous stores
- Option 3: Predict the dependence of a load on an outstanding store
Memory Disambiguation (I)

- Option 1: Assume load is dependent on all previous stores
  + No need for recovery
  -- Too conservative: delays independent loads unnecessarily

- Option 2: Assume load is independent of all previous stores
  + Simple and can be common case: no delay for independent loads
  -- Requires recovery and re-execution of load and dependents on misprediction

- Option 3: Predict the dependence of a load on an outstanding store
  + More accurate. Load store dependences persist over time
  -- Still requires recovery/re-execution on misprediction
    - Alpha 21264: Initially assume load independent, delay loads found to be dependent
Predicting store-load dependences important for performance

Simple predictors (based on past history) can achieve most of the potential performance
Data Forwarding Between Stores and Loads

- We cannot update memory out of program order
  - Need to buffer all store and load instructions in instruction window

- Even if we **know** all addresses of past stores when we generate the address of a load, two questions still remain:
  1. How do we check whether or not it is dependent on a store
  2. How do we forward data to the load if it is dependent on a store

- Modern processors use a LQ (load queue) and a SQ for this
  - Can be combined or separate between loads and stores
  - A load searches the SQ after it computes its address. Why?
  - A store searches the LQ after it computes its address. Why?
Out-of-Order Completion of Memory Ops

- When a store instruction finishes execution, it writes its address and data in its reorder buffer entry (or SQ entry).

- When a later load instruction generates its address, it:
  - searches the SQ with its address
  - accesses memory with its address
  - receives the value from the youngest older instruction that wrote to that address (either from ROB or memory)

- This is a complicated “search logic” implemented as a Content Addressable Memory
  - Content is “memory address” (but also need size and age)
  - Called store-to-load forwarding logic
Store-Load Forwarding Complexity

- **Content Addressable Search** (based on Load Address)

- **Range Search** (based on Address and Size of both the Load and earlier Stores)

- **Age-Based Search** (for last written values)

- **Load data can come from a combination of multiple places**
  - One or more stores in the Store Buffer (SQ)
  - Memory/cache
CYCLE

MUL R1, R2 → R3
ADD R3, R4 → R5
ADD R2, R6 → R7
ADD R8, R9 → R10
MUL R7, R10 → R11
ADD R5, R11 → R5

Register-Alias Table

<table>
<thead>
<tr>
<th>V</th>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SOURCE 1

<table>
<thead>
<tr>
<th>V</th>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SOURCE 2

<table>
<thead>
<tr>
<th>V</th>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Source 1

<table>
<thead>
<tr>
<th>V</th>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td>Value</td>
<td></td>
</tr>
</tbody>
</table>

Source 2

<table>
<thead>
<tr>
<th>V</th>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td>Value</td>
<td></td>
</tr>
</tbody>
</table>
Digital Design & Computer Arch.

Lecture 16b: Handling Out-of-Order Execution of Loads and Stores

Prof. Onur Mutlu

ETH Zürich
Spring 2022
28 April 2022