T12 Advanced Processors: 
Memory Disambiguation

School of Electrical and Computer Engineering
Cornell University

revision: 2020-11-11-15-46

1 Adding Memory Instructions to an OOO Processor 2
2 In-Order Load/Store Issue with Unified Stores 6
3 In-Order Load/Store Issue with Split Stores 8
4 Out-of-Order Load/Store Issue 9

Copyright © 2018 Christopher Batten, Christina Delimitrou. All rights reserved. This handout was originally prepared by Prof. Christopher Batten at Cornell University for ECE 4750 / CS 4420. It has since been updated by Prof. Christina Delimitrou. Download and use of this handout is permitted for individual educational non-commercial purposes only. Redistribution either in part or in whole via both commercial or non-commercial means requires written permission.
1. Adding Memory Instructions to an OOO Processor

• Adding memory instructions to I2OE microarchitecture
  – Add M pipe in parallel to X and Y pipe
  – Commit point is in D so no problem with writing memory in M pipe
  – Early commit point can be difficult to achieve in practice

• Adding memory instructions to I2OL/IO2L microarchitectures
  – Must wait to do stores after commit point (in C stage)
  – Do not want to wait until C stage to handle loads

• Add finished-store buffer (FSB) in parallel to ROB
  – Sometimes called the "store queue"
  – Allocate entries in FSB in-order in D stage (like ROB)
  – Write entries in FSB out-of-order in W stage (like ROB)
  – Deallocate entries from FSB in-order in C stage (like ROB)

• L0: generate load address
• L1: access data cache to load data
• S: pass along store data, generate store address
• W (load): write load data into PRF and clear pending bit in ROB
• W (store): write store address and store data into FSB and clear pending bit in ROB
• C (store): send write request out to memory and wait for write ack
1. Adding Memory Instructions to an OOO Processor

Data Structures: FSB

- Finished-Store Buffer (FSB)
  - \( v \): valid bit
  - \( \text{addr} \): generated store address
  - \( \text{data} \): store data

Example Execution Diagrams

```
<table>
<thead>
<tr>
<th></th>
<th>0</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>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>lw x1, 0(x2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>lw x3, 0(x4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>mul x5, x1, x3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>sw x5, 0(x6)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Aside: Example Execution Diagrams

Can we avoid stalling entire pipeline on a store miss?
1. Adding Memory Instructions to an OOO Processor

Without R stage, stall in C stalls all younger instructions

\[
\begin{array}{ccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
\hline
a: & \text{opA} & & & & & & & & & & & & \\
b: & \text{sw x1, 0(x2)} & & & & & & & & & & & & \\
c: & \text{opB} & & & & & & & & & & & & \\
d: & \text{opC} & & & & & & & & & & & & \\
e: & \text{opD} & & & & & & & & & & & & \\
\end{array}
\]

With R stage, stall due to cache miss is decoupled from C stage

\[
\begin{array}{ccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
\hline
a: & \text{opA} & & & & & & & & & & & & \\
b: & \text{sw x1, 0(x2)} & & & & & & & & & & & & \\
c: & \text{opB} & & & & & & & & & & & & \\
d: & \text{opC} & & & & & & & & & & & & \\
e: & \text{opD} & & & & & & & & & & & & \\
\end{array}
\]

WAW dependencies assuming IO issue

\[
\begin{array}{ccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
\hline
a: & \text{sw x1, 0(x2)} & & & & & & & & & & & & \\
b: & \text{sw x3, 0(x4)} & & & & & & & & & & & & \\
\end{array}
\]

Assume R[x2] == R[x4]

WAW dependencies assuming OOO issue

\[
\begin{array}{ccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
\hline
a: & \text{sw x1, 0(x2)} & & & & & & & & & & & & \\
b: & \text{sw x3, 0(x4)} & & & & & & & & & & & & \\
\end{array}
\]
1. Adding Memory Instructions to an OOO Processor

### WAR dependencies assuming IO issue

<p>| | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>a: lw x1, 0(x2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b: sw x3, 0(x4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assume R[x2] == R[x4]

### WAR dependencies assuming OOO issue

<p>| | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>a: lw x1, 0(x2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b: sw x3, 0(x4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assume R[x2] == R[x4]

### RAW dependencies assuming IO issue

<p>| | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>a: sw x1, 0(x2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b: lw x3, 0(x4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assume R[x2] == R[x4]

### RAW dependencies assuming OOO issue

<p>| | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>a: sw x1, 0(x2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b: lw x3, 0(x4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assume R[x2] == R[x4]
2. In-Order Load/Store Issue with Unified Stores

- Integer IQ supports out-of-order issue
- Memory IQ only supports in-order issue
- Two IQs can act as distributed IQ to facilitate superscalar execution
- Detecting potential RAW hazards
  - L0 stage searches FSB addresses (could also do this in L1)
  - Also search CSB if we are using an extra R stage for retirement
  - If no match in FSB then no RAW dependency exists, load can continue
  - If match in FSB then RAW dependency exists with in-flight store
- **Stall** to resolve RAW dependency
  - Stall load in L0 stage until store commits
  - Address comparison can be conservative to simplify hardware
- **Bypass/Forward** to resolve RAW dependency
  - Bypass data from FSB into end of L0
  - Need to bypass from *youngest* store in FSB
  - Address comparison must be exact to avoid bypassing incorrect value
2. In-Order Load/Store Issue with Unified Stores

Example with RAW Dependency

<table>
<thead>
<tr>
<th></th>
<th>0</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>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>sw</td>
<td>x1</td>
<td>O(x2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>lw</td>
<td>x3</td>
<td>O(x4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>add</td>
<td>x5</td>
<td>x6</td>
<td>x7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>lw</td>
<td>x8</td>
<td>O(x9)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assume R[x2] == R[x4] == R[x9]

- Inst b searches FSB in L0 and finds no match, but need to aggressively bypass store address/data from W stage
- Inst d searches FSB in L0 and finds match, bypasses data from FSB

Example without RAW Dependency

<table>
<thead>
<tr>
<th></th>
<th>0</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>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>lw</td>
<td>x1</td>
<td>O(x2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>mul</td>
<td>x3</td>
<td>x1</td>
<td>x4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>mul</td>
<td>x5</td>
<td>x3</td>
<td>x6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>sw</td>
<td>x5</td>
<td>O(x7)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>lw</td>
<td>x8</td>
<td>O(x9)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assume R[x2] != R[x7] != R[x9]

- Inst e is stuck behind store due to inorder issue ...
- ... but there is no RAW dependency between d and e
- ... and we know the addresses earlier!
3. In-Order Load/Store Issue with Split Stores

- **Key Idea:** split stores into store-data and store-addr micro-ops
  - Potentially split stores in D and merge store in W
  - FSB needs a valid bit for address and a valid bit for data

- **In D stage for a store**
  - If store data is not pending, then enqueue store in in-order memory IQ
  - If store data is pending, split store into two micro-ops: store-data micro-op goes in integer IQ and store-addr micro-op goes in mem IQ

- **In I stage for store micro-ops**
  - Store-data micro-ops use X-pipe
  - Store-addr micro-ops use S-pipe

- **In W stage for store micro-ops**
  - Store-data micro-op writes data field and sets data valid bit
  - Store-addr micro-op writes address field and sets address valid bit

- **In C stage for stores**
  - When store is at head of ROB, can only commit if both valid bits set

- **What if L0 finds an address match in FSB, but data not valid?**
  - Stall load in L0 if address match, but data not valid
  - Enable re-issue by keeping load in mem IQ until there is no match
Example without RAW Dependency

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>lw x1, 0(r2)</td>
</tr>
<tr>
<td>1</td>
<td>mul x3, x1, r4</td>
</tr>
<tr>
<td>2</td>
<td>mul x5, x3, r6</td>
</tr>
<tr>
<td>3</td>
<td>sw x5, 0(x7)</td>
</tr>
<tr>
<td>4</td>
<td>f: lw x8, 0(x9)</td>
</tr>
</tbody>
</table>

Assume R[r2] != R[r7] != R[r9]

- Inst f checks address in L0, finds no match, and can continue
- Assume D can put micro-ops into int and mem IQ in same cycle

4. Out-of-Order Load/Store Issue

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>sw x1, 0(x2)</td>
</tr>
<tr>
<td>1</td>
<td>lw x3, 0(x4)</td>
</tr>
</tbody>
</table>

Assume R[x2] == R[x4]

- Checking FSB in L0 will not help, store address is not in the FSB yet!
- Speculatively issue loads assuming no RAW hazard
  - Check later to see if RAW hazard has occurred
  - Squash all instructions after load and restart if detect hazard
4. Out-of-Order Load/Store Issue

- Only one IQ required (combining with split stores still possible)
  - Searching FSB more complicated
  - Need “age” logic to track which stores are older vs younger than the load in L0 searching the FSB
  - Stall/bypass from “youngest older” store

- Add finished-load buffer (FLB)
  - Sometimes called the “load queue”
  - FLB holds address of loads that have finished but not committed
  - Allocate entries in FLB in-order in D stage (like ROB)
  - Write entries in FLB out-of-order in W stage (like ROB)
  - Deallocate entries from FLB in-order in C stage (like ROB)

- Checking for RAW hazards
  - Store in S stage searches the FLB
  - Need “age” logic to track which loads are older vs younger than the store in S searching the FLB
  - If store finds an address match for a younger load, then there has been a memory RAW hazard (memory dependence violation)
  - Mark the corresponding load, when that load commits, squash all instructions in the pipeline, and re-execute from load

- FSB (store queue) and FLB (load queue) sometimes combined into a single complex data-structure called the load-store queue (LSQ)
4. Out-of-Order Load/Store Issue

Loads checking FSB

Assume R[x2] == R[x4] == R[x6]

Stores checking FLB

Assume R[x2] == R[x4] == R[x6]

Complex example