

ECE 4750 Computer Architecture, Fall 2022

Topic 9: Advanced Processors Memory Disambiguation

School of Electrical and Computer Engineering
Cornell University

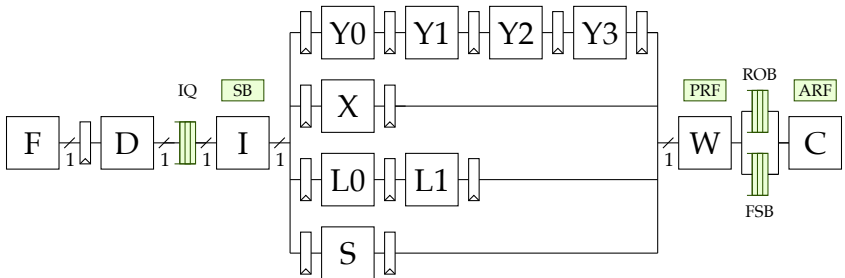
revision: 2022-11-16-13-50

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 © 2022 Christopher Batten. All rights reserved. This handout was prepared by Prof. Christopher Batten at Cornell University for ECE 4750 Computer Architecture. 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

Data Structures: FSB

- Finished-Store Buffer (FSB)
 - **v**: valid bit
 - **addr**: generated store address
 - **data**: store data

Finished Store Buffer

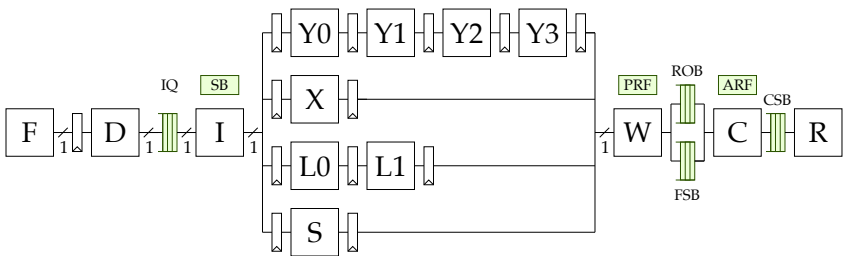
v	addr	data

Example Execution Diagrams

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a: lw x1, 0(x2)															
b: lw x3, 0(x4)															
c: mul x5, x1, x3															
d: sw x5, 0(x6)															

Aside: Example Execution Diagrams

Can we avoid stalling entire pipeline on a store miss?



Without R stage, stall in C stalls all younger instructions

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a : opA															
b : sw x1, 0(x2)															
c : opB															
d : opC															
e : opD															

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

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a : opA															
b : sw x1, 0(x2)															
c : opB															
d : opC															
e : opD															

WAW dependencies assuming IO issue

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a : sw x1, 0(x2)															
b : sw x3, 0(x4)															

Assume $R[x2] == R[x4]$

WAW dependencies assuming OOO issue

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a : sw x1, 0(x2)															
b : sw x3, 0(x4)															

WAR dependencies assuming IO issue

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a: lw x1, 0(x2)															
b: sw x3, 0(x4)															

Assume $R[x2] == R[x4]$ **WAR dependencies assuming OOO issue**

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a: lw x1, 0(x2)															
b: sw x3, 0(x4)															

Assume $R[x2] == R[x4]$ **RAW dependencies assuming IO issue**

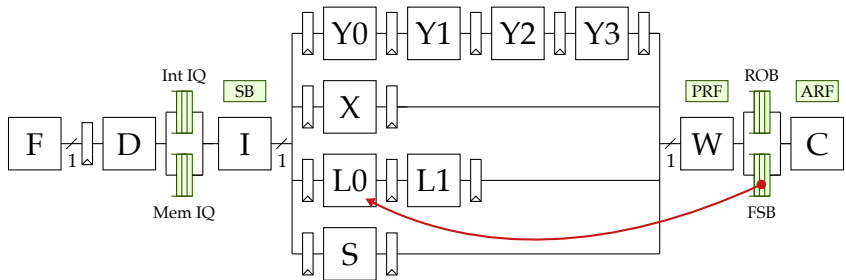
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a: sw x1, 0(x2)															
b: lw x3, 0(x4)															

Assume $R[x2] == R[x4]$ **RAW dependencies assuming OOO issue**

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a: sw x1, 0(x2)															
b: lw x3, 0(x4)															

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

Example with RAW Dependency

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a: sw x1, 0(x2)															
b: lw x3, 0(x4)															
c: add x5, x6, x7															
d: lw x8, 0(x9)															

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

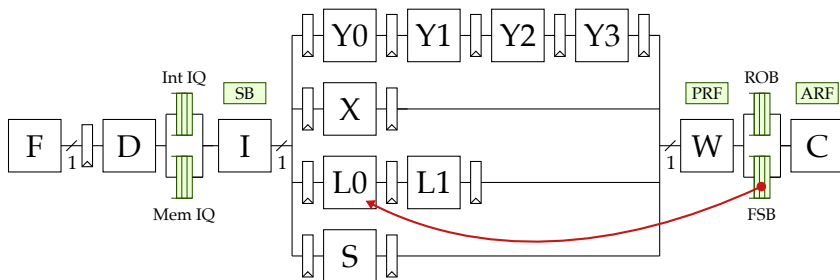
Example without RAW Dependency

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a: lw x1, 0(x2)															
b: mul x3, x1, x4															
c: mul x5, x3, x6															
d: sw x5, 0(x7)															
e: lw x8, 0(x9)															

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

- Inst e is stuck behind store due to in-order 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

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
a: lw x1, 0(r2)																	
b: mul x3, x1, r4																	
c: mul x5, x3, r6																	
d: sw x5, 0(x7)																	
e:																	
f: lw x8, 0(x9)																	

Assume $R[r2] \neq R[r7] \neq R[r9]$

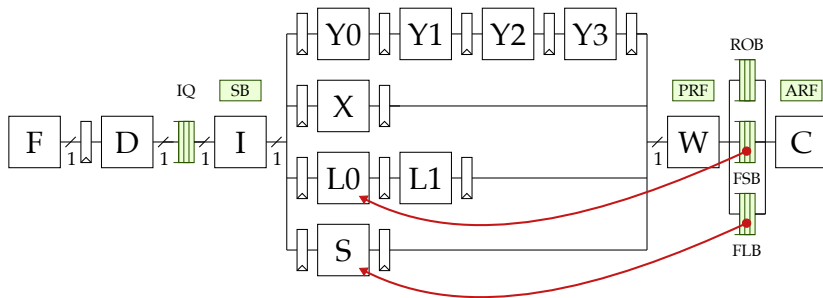
- Inst f checks address in L0, finds no match, and can continue
- Assume D can put micro-ops into int and mem IQ in same cycle

4. Out-of-Order Load/Store Issue

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a: sw x1, 0(x2)															
b: lw x3, 0(x4)															

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



- 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)

Loads checking FSB

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a: sw x1, 0(x2)															
b: lw x3, 0(x4)															
c: sw x5, 0(x6)															

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

Stores checking FLB

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
a: lw x1, 0(x2)																		
b: sw x3, 0(x4)																		
c: lw x5, 0(x6)																		
d: addi x7, x5, 1																		
e: addi x8, x7, 1																		

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

Complex example

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
a: sw x1, 0(x2)																						
b: sw x3, 0(x4)																						
c: lw x5, 0(x6)																						
d: sw x7, 0(x8)																						
e: addi x9, x5, 1																						
f: addi x10, x9, 1																						

Assume $R[x2] == R[x4] == R[x6] == R[x8]$