# **Memory Ordering**

#### Introduction

1. Load and store instructions can access the same memory without using the same register

## Memory Access Ordering

- 1. Eliminated control dependencies
  - Branch prediction
- 2. Eliminated false dependencies on registers
  - Register renaming
- 3. Obey raw register dependencies
  - Tomasulo-like scheduling
- 4. How do we handle memory dependencies?
  - Dependence between value written by a store and read by a load if they're accessing the same address

## When does Memory Write Happen?

- 1. Memory write occurs at commit
  - Unsafe to update memory before an instruction commits (could be cancelled)
- 2. When does a load get data? Does it have to happen at commit?
  - Want loads to happen as early as possible
- 3. Load-Store Queue

### Load Store Queue Part 1

- 1. Loads need to happen as soon as possible, store happens at commit
- 2. Load-Store Queue: Structure just like a ROB
  - Insert instructions in order and remove at commit
  - Only contains loads and stores
  - Contains the following fields
    - L/S bit: Designate if instruction is load or store
    - Address: What address is accessed?
    - Value: Value to be stored
    - Complete: Instruction is complete
  - For every load inserted, check if any store matches that address
    - If there is no matching store, go to memory
    - If there is a matching store, do "store-to-load forwarding"
  - Store-to-load forwarding: Copy the value from the store accessing the same address into the load entry for the load

### Load Store Queue Part 2

- 1. As long as the address computed by the store is known at the time of a subsequent load, store-to-load forwarding works well
- 2. However, it is possible that the target of the store is still being computed when the next load is enqueued. In this case, there are three options:
  - Make loads and stores execute in order
    - Guarantees correctness at the expense of performance
  - Wait for all previous store addresses to be known
    - Either go to memory or get the value from the store
  - Just let the load go anyway
    - If no addresses match, let the load go to memory

- If an address does match, wait for that store to finish
- If a store finishes and it turns out that the load retrieved the wrong value from memory, we need to reload the value and rerun any instruction that depended on that result
- 3. Most modern processors take the third approach
  - Accessing the same memory location is an uncommon case, so we get the best performance
  - Only rarely pay the cost of recovering from an incorrect load
  - Modern hardware also tries to predict whether or not a load is likely to access the same address as a previous store and waits in this case

#### Out of Order Load Store Execution

1. Consider the following program:

LOAD R3 = 0(R6) ADD R7 = R3 + R4 STORE R4 -> 0(R7) SUB R1 = R1 - R2 LOAD R8 = 0(R1)

- 2. Suppose the first load is a cache miss and takes many cycles to complete
  - The following ADD and STORE can't execute until this load finishes
  - The SUB finishes quickly and allows the LOAD to complete
- 3. Then, the first LOAD finishes
  - Once the address of the STORE is computed, there are two cases
    - The final load accessed a different memory location, in which case there is no issue
    - The final load accessed this memory location, in which case R8 should contain the value in R4, not whatever stale value was in memory

#### In Order Load Store Execution

- 1. With in-order execution, if there is any load with a previous store that has yet to complete, we stall
- 2. Store is considered done when the instruction is complete, not committed
- 3. Bad performance, but correct

#### Memory Ordering Quiz 0

1. If we do everything in order, when does each instruction read/write to memory? Assume loads take 40 cycles and stores take 1 cycle.

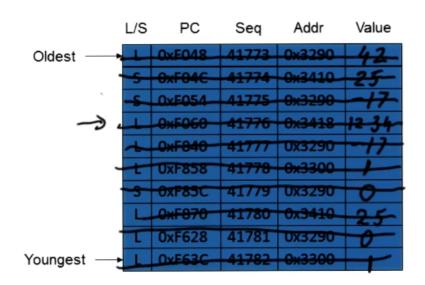
Instruction	-> Memory	Memory ->
LW R1,0(R2)	1	41
SW R1,4(R2)	42	43
LW R1,0(R3)	43	83
SW $R1,4(R3)$	84	85
LW $R1,0(R4)$	85	125
SW R1,4(R4)	125	126

#### Store to Load Forwarding

- 1. Load: Which earlier store do we get our value from?
  - Could be multiple stores going to the same address
- 2. Store: Which later load do I give my value to?
  - Could be multiple loads
- 3. This determination is made in the load-store queue

## LSQ Example

- 1. Every load searches the load-store queue for previous instructions that access the same address (only use the most recent previous instruction)
  - If it finds one, it uses its value
  - Otherwise, it goes to the cache, then memory
- 2. When a load commits, it simply updates the ARF
- 3. When a store commits, it updates the value in the cache/memory



Data Cache		
0x3290	0	
0x3300	- 1	
0x3410	25	
0x3418	12 34	

LSQ Example

### LSQ-ROB-RS

- 1. Issuing a load/store requires:
  - A ROB entry
  - A LSQ entry
- 2. Issuing a non-load/store requires:
  - A ROB entry
  - A RS
- 3. Executing a load/store
  - Compute address
  - Produce value
- 4. For loads, computing the address and producing the value must happen in order
  - A load must broadcast its result to subsequent instructions
  - On commit, we free the ROB and LSQ entries
- 5. For stores, computing the address and producing the value can be out-of-order
  - On commit, we free the ROB and LSQ entries, as well as send the value to memory

# Memory Ordering Quiz 1

1. Consider the following program:

SW R1  $\rightarrow$  O(R2) LW R2  $\leftarrow$  O(R2)

- 2. Does LW access cache or memory?
  - No

# Memory Ordering Quiz 2

1. Consider the following program:

SW R1 -> O(R2) LW R2 <- O(R2)

- 2. The load gets its result from:
  - A result broadcast (false)
  - A reservation station (false)
  - A ROB entry (false)
  - An LSQ entry (true)

# Conclusion

1. Load-store queue is used to track dependencies through memory to ensure correct execution of memory instructions in out-of-order processors