EECS 470 Lab 7

LSQ Tips

Department of Electrical Engineering and Computer Science
College of Engineering
University of Michigan

Thursday, March 21st, 2024

Overview

Project Deadlines

Project

- Milestone 2 due Thursday, March 28th
    - Goal: everything except memory working
    - Run `mult_no_lsq.s` and have correct .wb output
    - Another 1-page progress report, with a top level architectural diagram
    - Past experience suggests it takes 7-10 days to wire your pipeline together and debug *after* writing all individual modules
- Milestone 3 due Thursday, April 11th
    - Short report on your progress
    - Optional meeting depending on project status

High Level

Why is Memory Annoying?

- Memory operations can have a huge latency!
- Memory operations don't work well with speculation
  - Making memory state update out of order would get very complicated...
  - Can't take back a store once it's gone
  - Loads may issue out of order (and may need to grab old value)

2 Main Goals:

1. Hide memory latency (cache is mostly responsible for this)
   - See lab 6 for this
2. Ensure correctness (here is where the LSQ comes in!)

Correctness

Why is Correctness a Concern?

- Loads are dependent on stores that write to the same address
  - ANY unresolved store before a load creates a possible dependency since we don't know where it is writing
- Without stores, loads out of order would cause no issue
  - Might impact performance through invalidating cache lines needed for earlier loads
  - But loads themselves can never cause correctness issues
- Dependencies don't impact anything when all operations are in order

Design Space

Design Space

- There is a huge range of complexity with the load-store queue
  - Have to trade off between complexity and performance
  - Find the optimizations that will give you the most bang for your buck
- This guide aims to give a high-level overview of your options

Implementation

- Memory operations are significantly harder to debug
  - Incorrect values could have been written 1,000's of cycles prior
- Simply stalling on corner cases can reduce the number of bugs significantly
  - Ex: byte-level forwarding

Speculation and Forwarding

Options

- **No speculation or forwarding**: Must stall all loads or stores until the last prior store reaches the head of the ROB
- **Forwarding:** Keep track of the values and addresses the stores write to, and forward their data to any later loads if the addresses match
  - Requires adding a store queue, but not a load queue
  - Have to be careful about stores to bytes, half-words, and words
- **Speculation:** Let loads go ahead before prior stores know their addresses
  - But have to save addresses and check that they don't match prior stores resolved later
  - Requires adding a load queue and creating new infrastructure to squash the incorrect load and all dependent instructions

Speculation and Forwarding

Complexity

- Now we will go over many options for your LSQ architecture, ranging from simplest to most complex
- Be aware that this is a huge range of complexity, and each option will take drastically different amounts of time

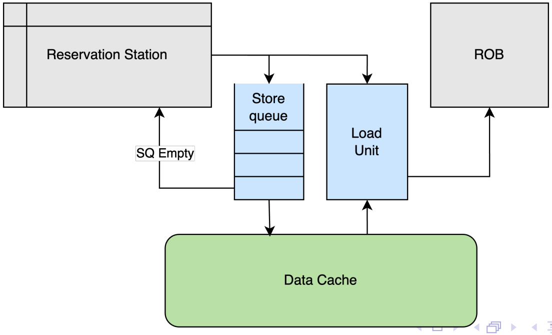
General Notes

- Stores may not write to cache immediately due to memory bus contention
- Stores are added to store queue on dispatch to ensure they are in order
  - Same for load queue if present

No Forwarding or Speculation (But Out-of-Order Loads)

Approach

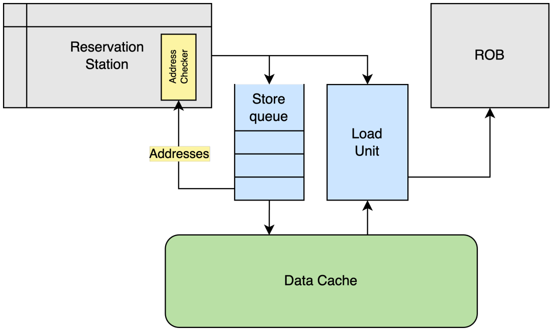
- Only issue a load if there is no older store is in the pipeline
- Add stalling logic in RS and a notification line for when the last store completes
- Can't have a load in progress if an older store has not completed



No forwarding or speculation Optimization

Approach

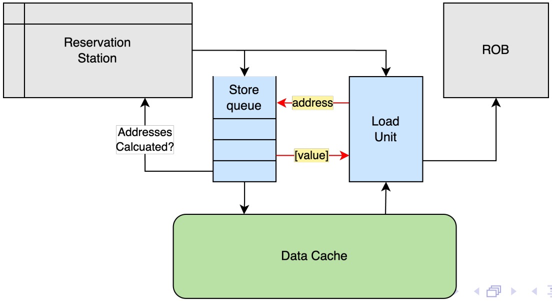
- Only issue load if all older stores have calculated their address
  - Can interleave loads + stores, but only when stores have their addresses
- Don't issue a load whose address matches an in-flight store
  - Stall in RS until store has finished writing to D-cache



Adding forwarding (no speculation)

Approach

- Only issue load if all older stores have calculated their address
- If load address matches store in store queue, forward the value from the store queue
- If there is not a match in the store queue, grab the value from memory as usual



Forwarding at the Byte Level

Approach

- Now, account for byte offsets in address "matches"
- 8 possible ways to forward
  - Word can get data from 4 byte and 2 half word addresses
  - Half word can each forward from 2 byte addresses
- Example: lw is issuing at address 1000, and in the store queue there is an older sb at 1002, and an older sb at 1003
  - The lw would grab both of those bytes through forwarding, and get the other 2 bytes (1000 and 1001) from memory
  - Without byte-level resolution, the lw would have to stall until all the sb finish executing

Word	[3:0]			
Half	[3:2]		[1:0]	
Byte	[3]	[2]	[1]	[0]

Adding Speculation

Approach

- Issue loads whenever values are ready; assume no conflicting stores
  - Don't care if older stores have not calculated their addresses
- Still check the older stores that do have their address, and forward if addresses align
- If a store calculates its address and finds that a younger load has executed with a matching address, squash accordingly
- Now requires an ordered **load queue**
  - Must record load *and* store positions at dispatch

Benefits

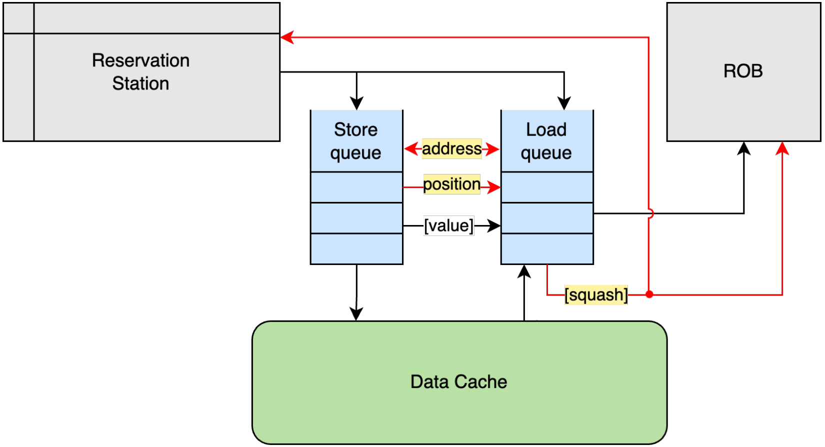
- Can now mask more of the memory latency of loads

Adding Speculation

Two Flavors of Speculation

- **Load-queue internal speculation**: Send memory requests before prior stores have resolved, but don't let the instruction leave the LSQ until safe to do so
  - Don't need to change other modules
- **CDB speculation**: speculated loads can go onto the CDB
  - Requires changes outside LSQ to squash loads (like mispredicted branches)
  - This is a large jump in complexity

Speculation Digram



NOTE: Only need squash signal if doing CDB speculation

Speculation and Forwarding with "Bad Pair" Caching

Approach

- Improve the performance by making a "bad pairs" table/cache of past problematic load/store pairs
  - Whenever a store-to-load pair causes an exception, add the pair to the table
- If you encounter the same load PC again, stall instead of speculating to avoid the likely squash
- Only issue the load if its associated store has calculated its address and value
- Essentially, this is conflict speculation

Recommended to use this if doing CDB speculation

Final Thoughts

- Each approach requires different levels of complexity
- Speculation is a BIG jump in complexity (load queue and squashing)
- Can significantly simplify logic by stalling hard cases
  - Minimal performance impact if only stalling in rare cases
  - Ex: byte to word forwarding between OoO loads and stores

	Simple	**Forwarding**	**LQ internal Speculation**	**CDB Speculation**
Stall in RS?	Yes	If not speculating	Rare	Rare
Store queue	No	Yes	Yes	Yes
Load queue	No	No	Yes	Yes
External Squashing	No	No	No	Yes