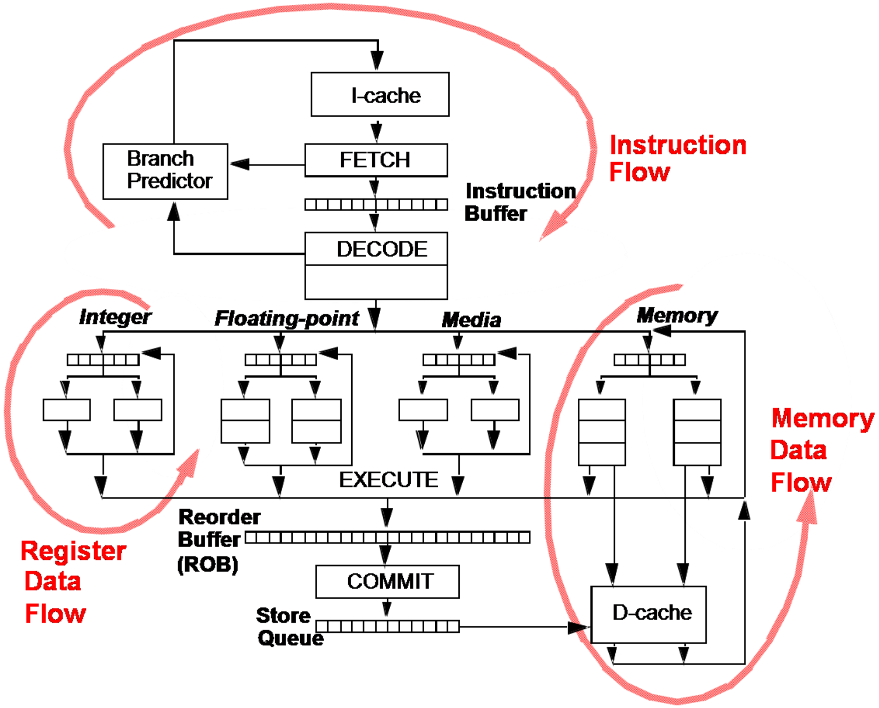
# EECS 470

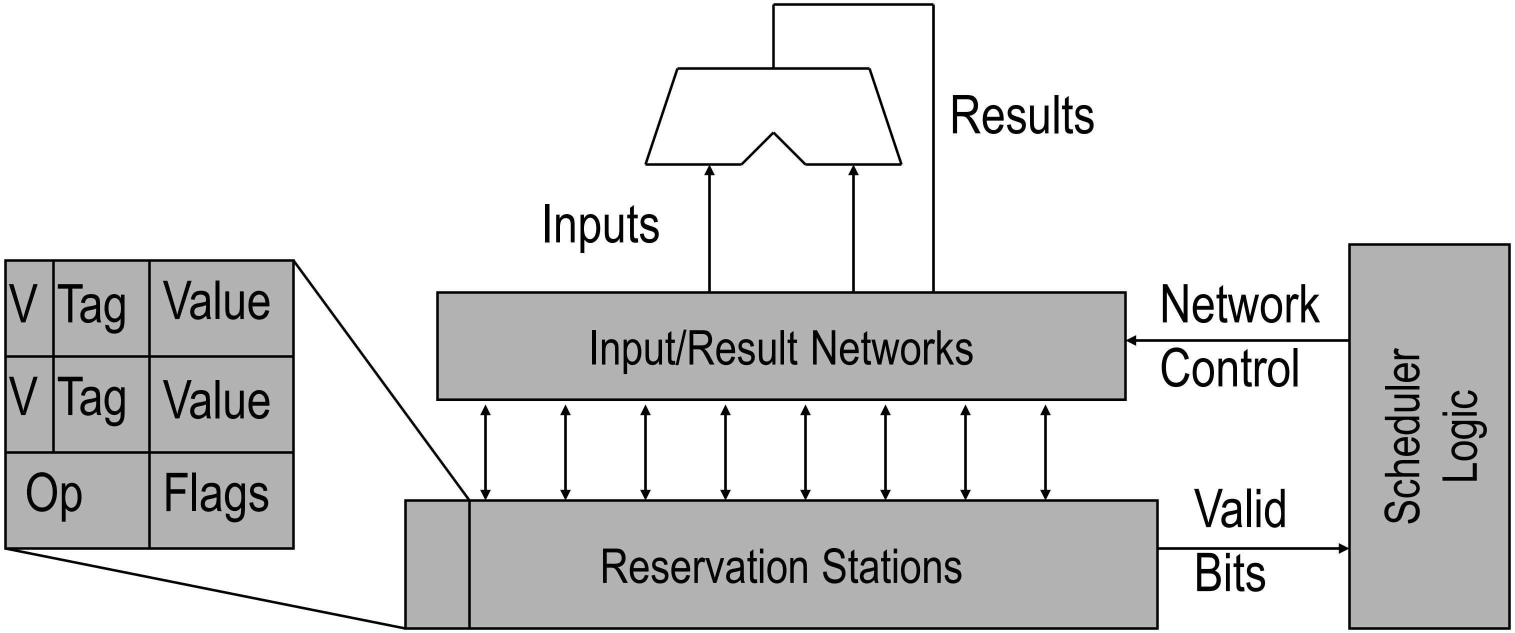
# Memory Speculation

## Winter 2024



Slides developed in part by Profs. Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, Wenisch of Carnegie Mellon University, Purdue University, University of Michigan, Univerity of Pennsylvania, and University of Wisconsin.

# Tomasulo-Style Scheduler Implementation



- Synchronization managed by scheduler logic
- Communication through input/output networks
- Infrastructure geared towards register communication

# Out-of-Order Memory Operations

Scheduling is straightforward in out-of-order…

- Register inputs only
- Register renaming captures all true dependences
- Tags tell you exactly when you can execute

… except with loads and stores

- Speculative stores cannot modify memory
  - Unless you can fix a mis-speculated store somehow!
- Register renaming does not tell you all dependences for loads
  - There are some in memory
- How do loads find older in-flight stores to same address (if any)?

- Issue of finding if addresses match is called **"memory disambiguation"**

# The Good: Register Communication

- Directly specified dependencies (contained in inst)
  - Accurate description of communication
    - no false or missing dependency edges
    - permits realization of dataflow schedule
  - Early description of communication
    - know dependencies upon decode
    - allows scheduler logic to be pipelined without impacting speed of communication
- Small communication name space (32-64 usually)
  - Fast access to communication storage
    - possible to map entire communication space (no tags)
    - possible to bypass communication storage
      - Forwarding

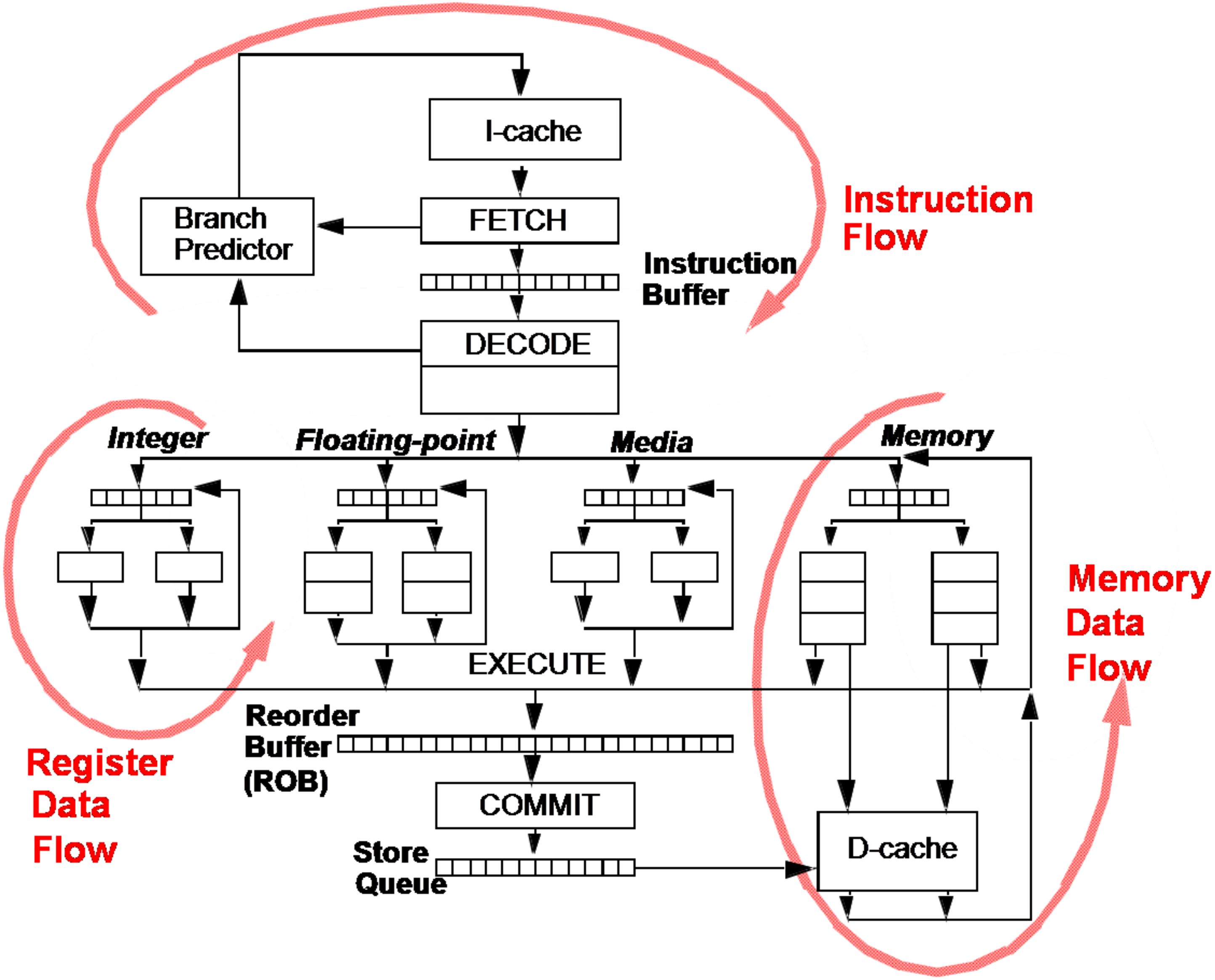
# The Bad (and the ugly): Memory Scheduling

- Recall how we handle dependencies in registers
  - We address false dependencies with register renaming.
  - We address true dependencies with the out-of-order machine
    - CDB broadcast and wakeup
- Why can't we do the same things with memory?
  - What would rename look like for memory?
    - Why might it be hard to pull off?
  - What about wakeup?
    - What's tricky here?
- Cannot directly use the same techniques
  - Indirectly specified memory dependencies
  - Large communication space ($2^{32-64}$ bytes!)
  - Memory latency is variable
    - Complicates scheduling

# Requirements for a Solution

- Accurate description of memory dependencies
  - No (or few) missing or false dependencies
  - Permit realization of dataflow schedule
- Early presentation of dependencies
  - Permit pipelining of scheduler logic
- Fast access to communication space
  - Preferably as fast as register communication (zero cycles)

# The Three Dependency "Flows"



# Implementation

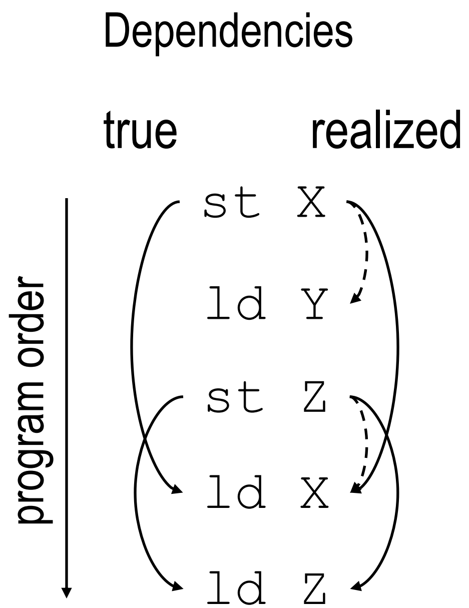
Several hardware realizations:

- Unified LSQ (easier to understand, but nasty hardware)
- Separate LQ* and SQ (more complicated, but fairly elegant)

We'll start with a unified LSQ and move to separate LB and SQ.

*Likely would end up with a load *buffer* (LB) rather than a queue…

# In-order Load/Store Scheduling



- ***Idea:*** Schedule all loads and stores in program order
  - This *cannot* violate true data dependencies (non-speculative)
- Capabilities/limitations:
  - Overly restrictive – likely to add *many* false dependencies
  - Early presentation of dependencies (no addresses)
  - Not fast, all communication through memory structures
- Found in in-order issue pipelines

# Consider the LSQ cases

| Slot | LSQ addresses (top is head) |
|---|---|
| A | Store 0x44 |
| B | Load 0x20 |
| C | Load 0x44 |
| D | Store ???? |
| E | Load 0x44 |
| F | Store 0x20 |
| G | Load 0x20 |
| H | Load 0x60 |

- Which of the loads are we sure we will fulfill via D$/Memory?
- Which of the loads will we fulfill via load-to-store forwarding?
- Which aren't we sure of?
- Identify what to do with each load.

# Unified Load/Store Queue

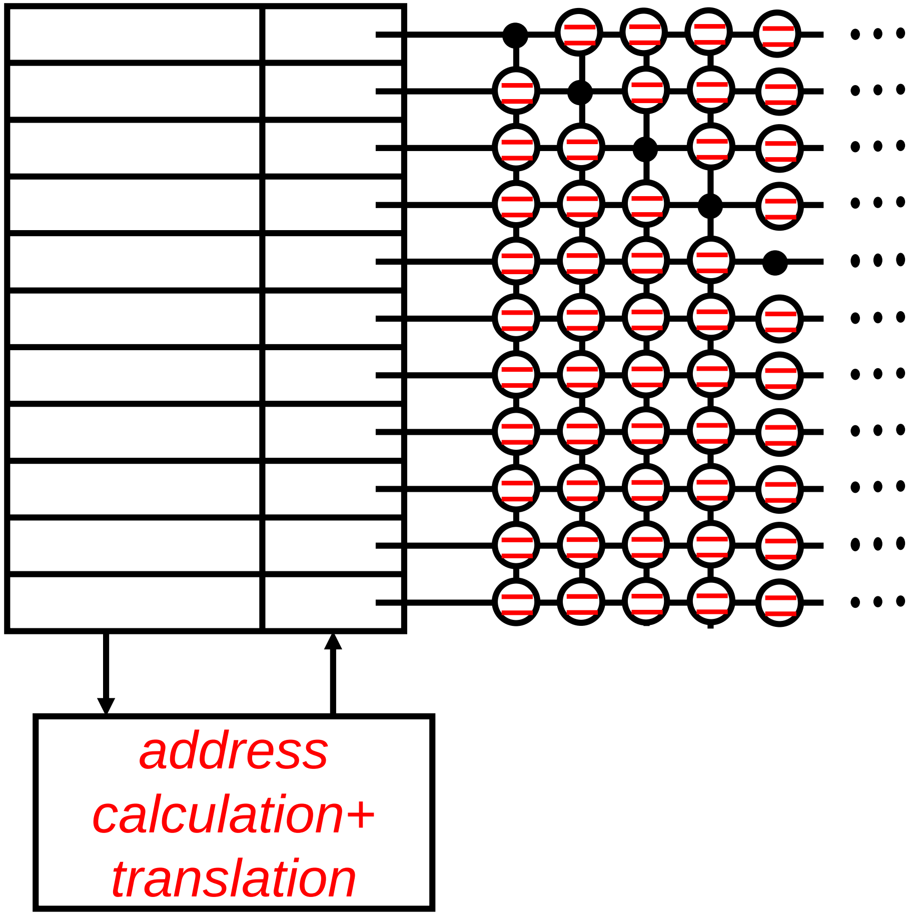
Operates as a circular FIFO

- Allocate on dispatch
- De-allocate on retirement

Calc address in register dataflow order

A NxN comparator matrix detects memory address dependence (also considers relative age of entries)

- Store ops are held until retirement
- Load ops are issued when no dependency exists & all older store addresses known



# Unified Load/Store Queue Questions

When do we search for store-to-load forwarding?

- As soon as we have the load address

What could happen once we have the load address?

- There is a store whose data we'll use
- There is no store whose data we'll use
- We aren't sure which store's data, if any, we'll use.

What should we do for each of those three cases?

# Split LQ and SQ

D$/TLB + structures to handle in-flight loads/stores

Performs four functions

**In-order store retirement**

- Writes stores to D$ in order
- Basic, implemented by store queue (SQ)

**Store-load forwarding**

- Allows loads to read values from older un-retired stores
- Data provided to LQ from SQ.

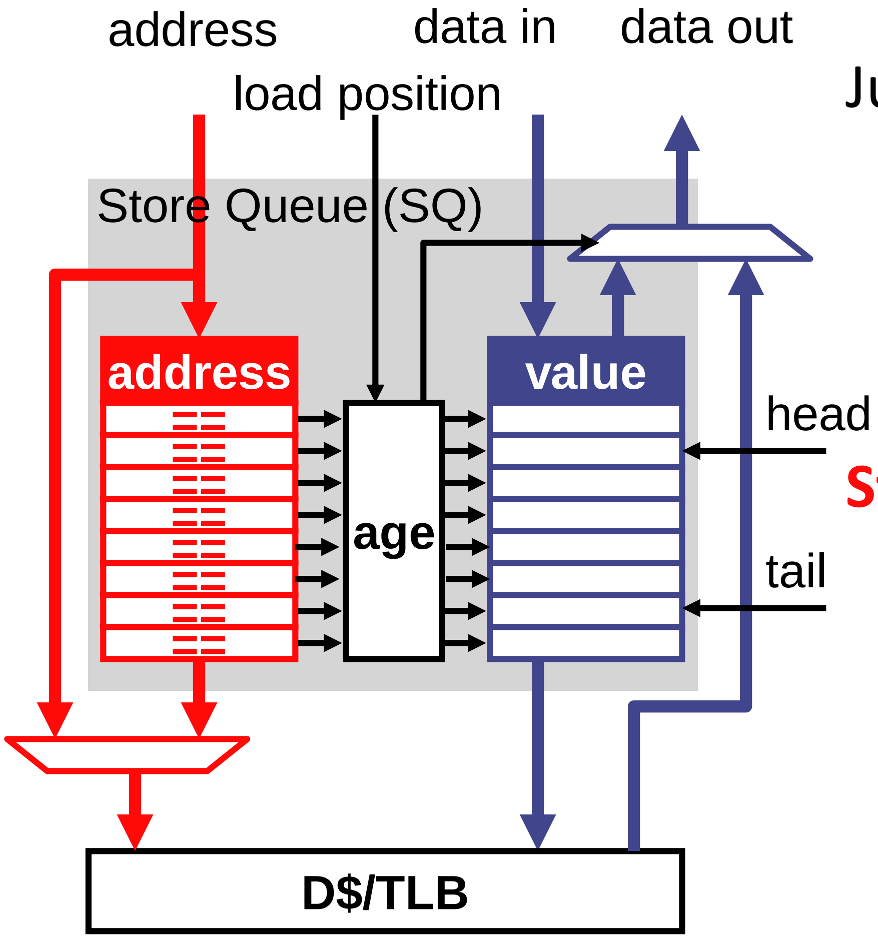
**Memory ordering violation detection**

- Checks load speculation (more later)
- Advanced, implemented by load queue (LQ)

**Memory ordering violation avoidance**

- Advanced, implemented by dependence predictors

# Simple Data Memory FU: D$/TLB + SQ



Just like any other FU

- 2 register inputs (addr, data in)
- 1 register output (data out)
- 1 non-register input (load pos)?

**Store queue (SQ)**

- In-flight store address/value
- In program order (like ROB)
- Addresses associatively searchable
- Size heuristic: 15-20% of ROB

But what does it do?

# When should a load access memory?

When to go to memory?

- Only at the head of the ROB (in order!)
- Only no stores between it and head of LSQ/ROB
- Only when there are no stores of the same address (*or unknown address*) between it and the head of the LSB/ROB.
- Load goes to memory at address calculation, gets fixed if there is a conflicting store

# When should a load go to the CDB?

- Once you know you have the right data
- Once you get some data
  - If you get this wrong, you need to squash it and all(?) following instructions.
- Based on a confidence predictor or other additional information.
  - “Memory dependence prediction”

# How to get data from stores?

- Forwarding:
  - If there is a conflicting store, get the data from it.
    - This isn’t trivial.

# Yet another idea…

- Let’s use a predictor

- Predict if a load is likely to be forwarding from a store
  - If not, get it from D$ and send it to the CDB
  - If so, wait.

- Thoughts on cost/benefit of speculation?
  - How should that impact our predictor?

“Memory dependence prediction”