# CS 152 Computer Architecture and Engineering

# Lecture 12 - Advanced Out-of-Order Superscalars

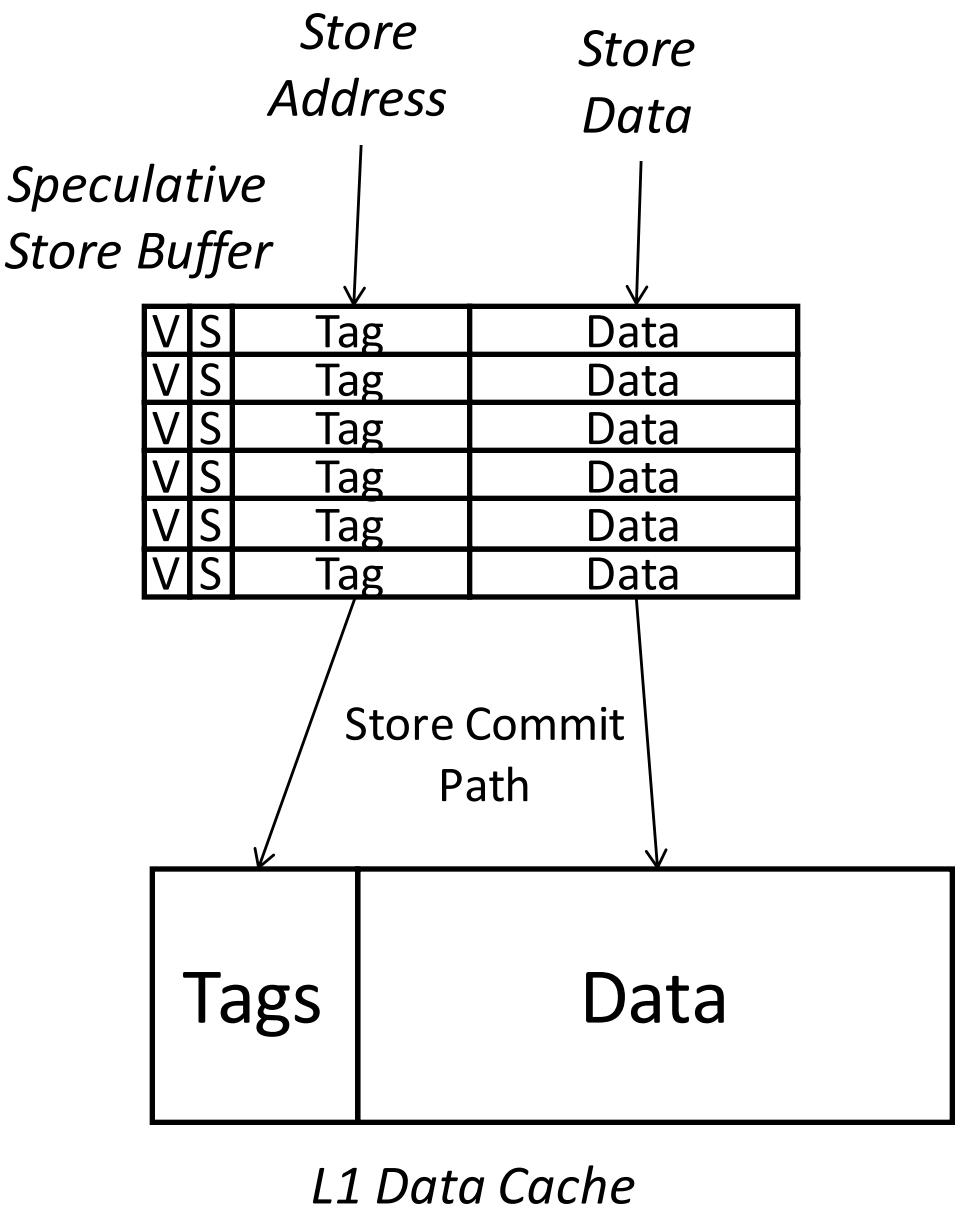
John Wawrzynek
Electrical Engineering and Computer Sciences
University of California at Berkeley

**http://www.eecs.berkeley.edu/~johnw**
**http://inst.eecs.berkeley.edu/~cs152**

# OOO SuperScalar Loads/Stores

- Loads and stores create performance challenges for OOO/superscalar execution
- Do we need to execute loads and stores in sequential order to assure correct function?
  - Many clever schemes to avoid sequential execution
- First, as with scalar processor – decouple stores with dedicated buffer
  - Committing stores from the ROB could hold up slots and slow execution

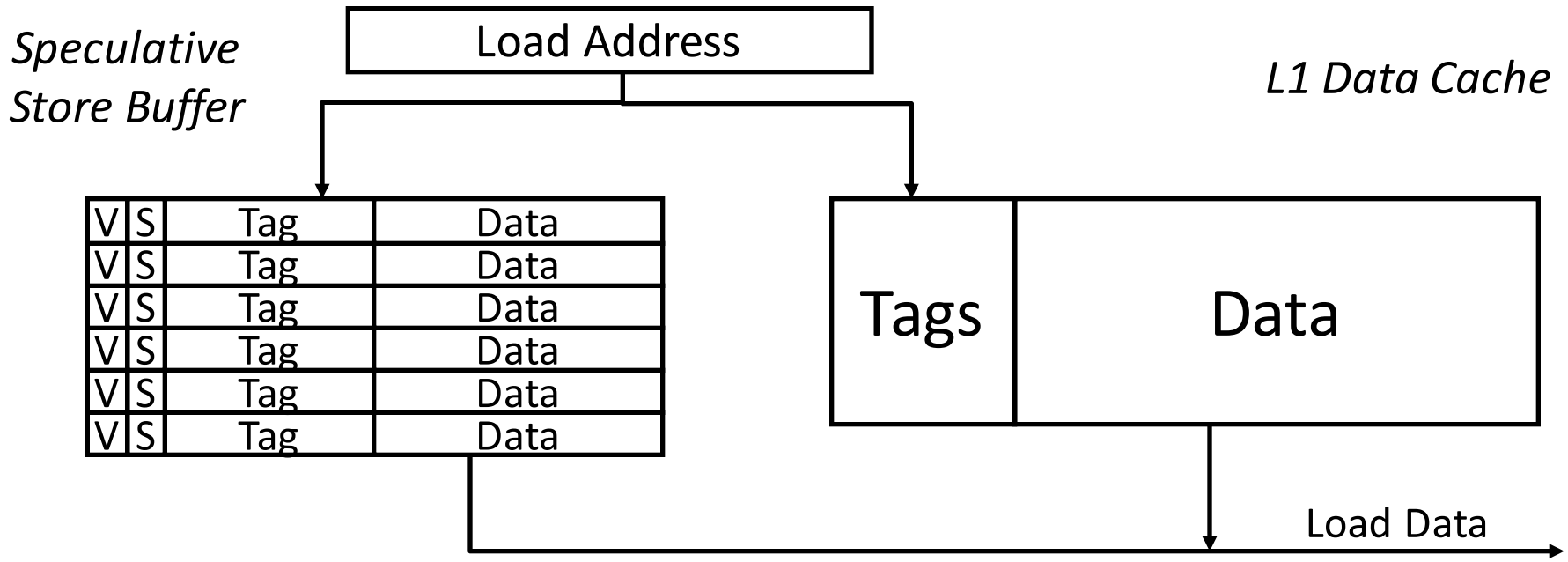
# Speculative Store Buffer




Just like register updates, stores should not modify the memory until after the instruction is committed. A speculative store buffer is a structure introduced to hold speculative store data.

- During decode, store buffer slot allocated in program order
- Stores split into "store address" and "store data" micro-operations
- "Store address" execute writes tag
- "Store data" execute writes data
- Store commits when oldest instruction and both address and data available:
  - clear speculative bit and eventually move data to cache
- On store abort:
  - clear valid bit

# Load bypass from speculative store buffer




- If data in both store buffer and cache, which should we use?

  Speculative store buffer

- If same address in store buffer twice, which should we use?

  Youngest store older than load

# Memory Dependencies

```
sd x1, (x2)
ld x3, (x4)
```

When can we execute the load?

# Approach 1: In-Order Memory Queue

- Execute all loads and stores in program order

=> Load and store cannot issue for execution until all previous loads and stores have completed execution

- Can still execute loads and stores speculatively, and out-of-order with respect to other instructions

- Need a structure to handle memory ordering…

# Approach 2: Conservative O-o-O Load Execution

```
sd x1, (x2)
ld x3, (x4)
```

- Can execute load before store, if addresses known and x4 != x2
- Each load address compared with addresses of all previous uncommitted stores
  - *can use partial conservative check i.e., bottom 12 bits of address, to save hardware*
- Don't execute load if any previous store address not known

*(MIPS R10K, 16-entry address queue)*

# Approach 3: Address Speculation

```
sd x1, (x2)
ld x3, (x4)
```

- Guess that x4 != x2
- Execute load before store address known
- Need to hold all completed but uncommitted load/store addresses in program order
- If subsequently find x4==x2, squash load and *all* following instructions

=> Large penalty for inaccurate address speculation

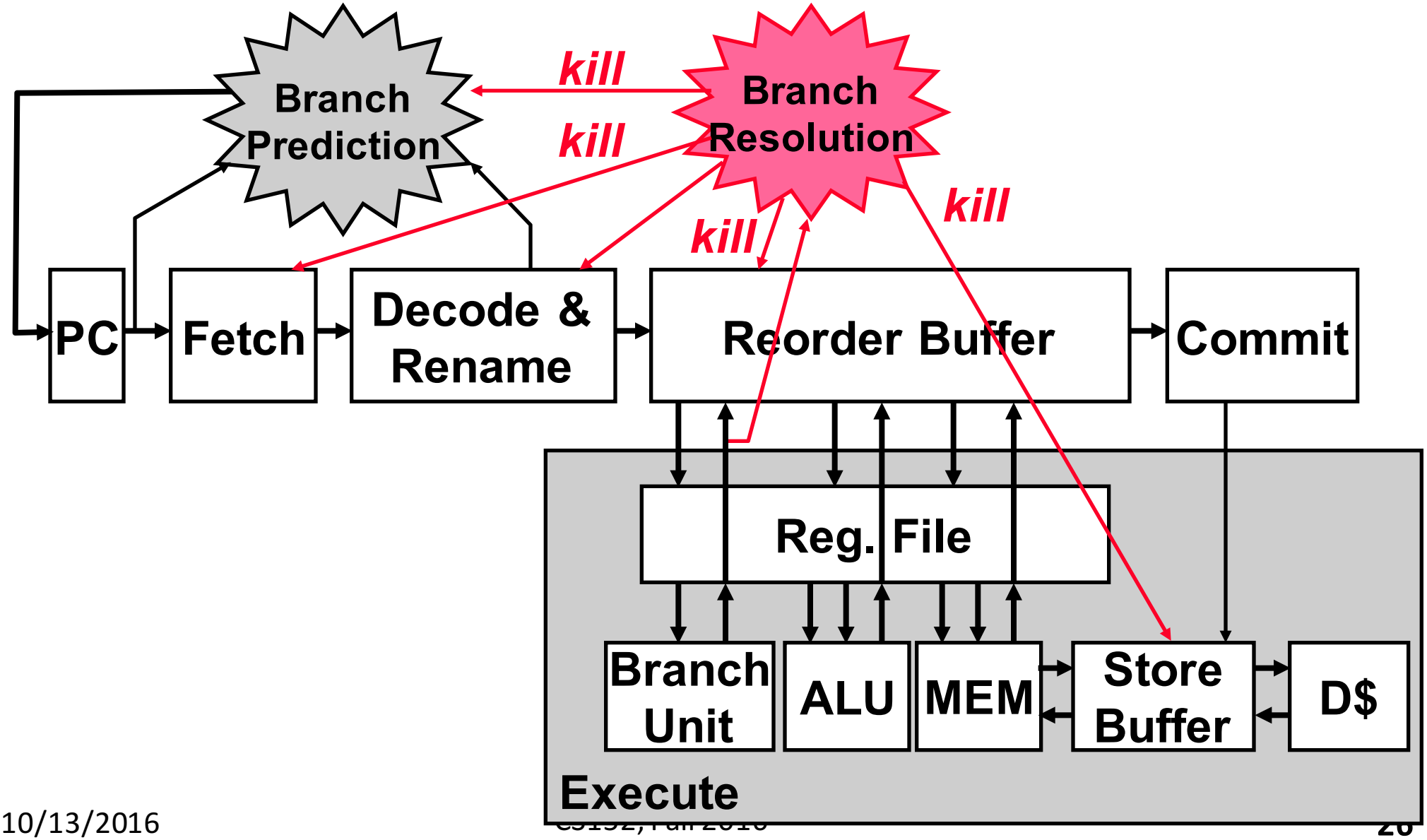
# Approach 4: Memory Dependence Prediction

*(Alpha 21264)*

```
sd x1, (x2)
ld x3, (x4)
```

- Guess that x4 != x2 and execute load before store
- If later find x4==x2, squash load and all following instructions, but mark load instruction as *store-wait*
- Subsequent executions of the same load instruction will wait for all previous stores to complete
- Periodically clear *store-wait* bits

# Unified Physical Register File Pipeline Review



# Instruction Flow in Unified Physical Register File Pipeline

- Fetch
  - Get instruction bits from current guess at PC, place in fetch buffer
  - Update PC using sequential address or branch predictor (BTB)
- Decode/Rename
  - Take instruction from fetch buffer
  - Allocate resources to execute instruction:
    - Destination physical register, if instruction writes a register
    - Entry in reorder buffer to provide in-order commit
    - Entry in issue window to wait for execution
    - Entry in memory buffer, if load or store
  - Decode will stall if resources not available
  - Rename source and destination registers
  - Check source registers for readiness
  - Insert instruction into issue window+reorder buffer+memory buffer

# Memory Instructions

- Split store instruction into two pieces during decode:
  - Address calculation, store-address
  - Data movement, store-data
- Allocate space in program order in memory buffers during decode
- Store instructions:
  - Store-address calculates address and places in store buffer
  - Store-data copies store value into store buffer
  - Store-address and store-data execute independently out of issue window
  - Stores only commit to data cache at commit point
- Load instructions:
  - Load address calculation executes from window
  - Load with completed effective address searches memory buffer
  - Load instruction may have to wait in memory buffer for earlier store ops to resolve

# Issue Stage

- Writebacks from completion phase "wakeup" some instructions by causing their source operands to become ready in issue window
  - In more speculative machines, might wake up waiting loads in memory buffer
- Need to "select" some instructions for issue
  - Arbiter picks a subset of ready instructions for execution
  - Example policies: random, lower-first, oldest-first, critical-first
- Instructions read out from issue window and sent to execution

# Execute Stage

- Read operands from physical register file and/or bypass network from other functional units
- Execute on functional unit
- Write result value to physical register file (or store buffer if store)
- Produce exception status, write to reorder buffer
- Free slot in instruction window

# Commit Stage

- Read completed instructions in-order from reorder buffer
  - (may need to wait for next oldest instruction to complete)
- If exception raised
  - flush pipeline, jump to exception handler
- Otherwise, release resources:
  - Free physical register used by last writer to same architectural register
  - Free reorder buffer slot
  - Free memory reorder buffer slot

# Acknowledgements


- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)

- MIT material derived from course 6.823
- UCB material derived from course CS252