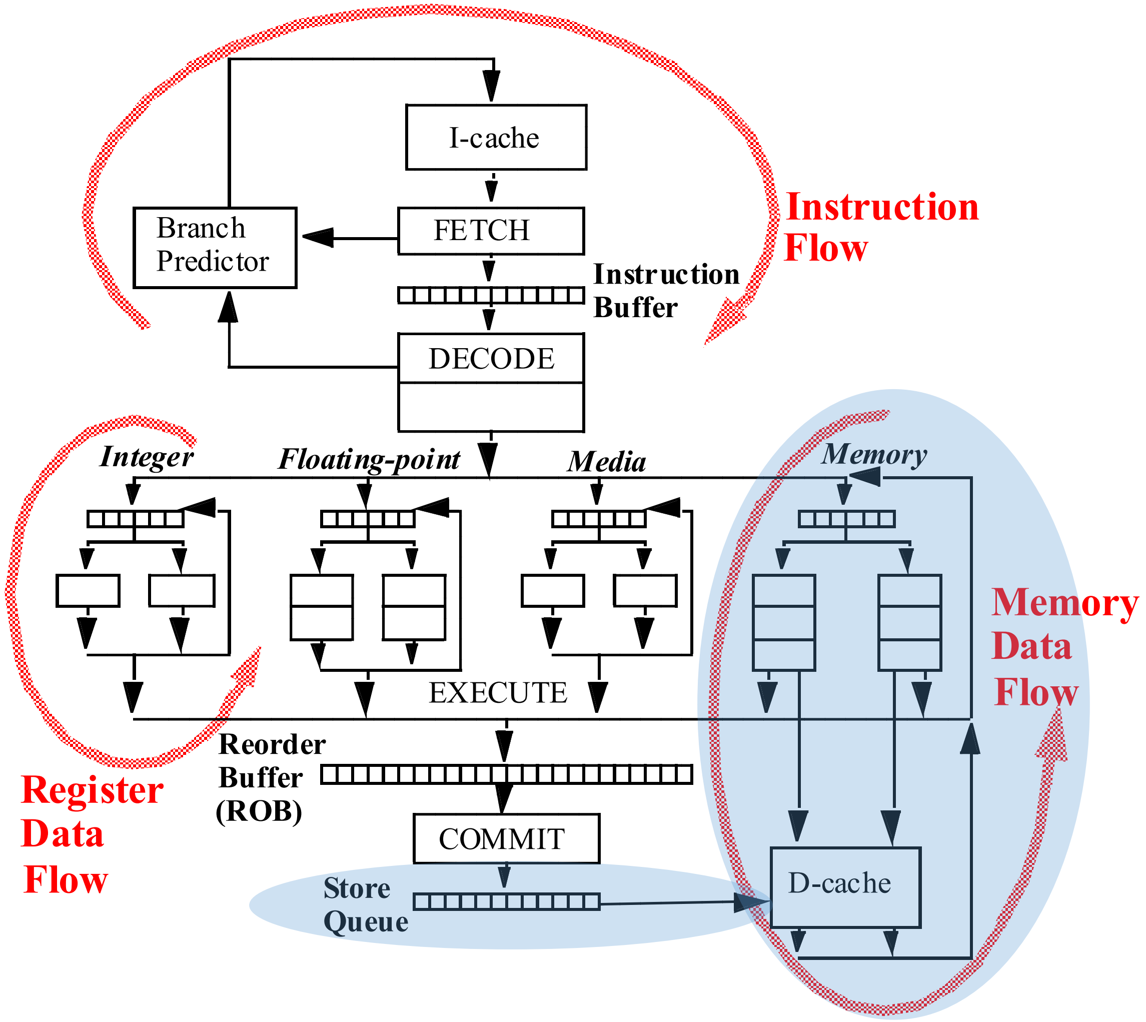
Memory Data Flow in Out-of-Order Pipelines

Nima Honarmand

Big Picture


I-cache
Branch Predictor
FETCH
Instruction Buffer
DECODE
Instruction Flow
Integer
Floating-point
Media
Memory
EXECUTE
Memory Data Flow
Reorder Buffer (ROB)
COMMIT
Register Data Flow
Store Queue
D-cache

OoO and Memory Instructions

- Memory instructions benefit from out-of-order execution just like other ones
- Especially important to execute loads as soon as address is known
  - Loads are at the top of dependence chains
- To enable precise state recovery, stores are sent to D$ after retirement
  - Sufficient to prevent wrong-branch-path stores
- Loads can be issued out-of-order w.r.t. other loads and stores <u>if no dependence</u>

OoO and Memory Instructions

- Memory instructions have same 3 types of dependences as register insts.
  - RAW (true), WAR and WAW (false)
- However, memory-based dependences are dynamic
  - Unlike register-based dependences
  - Often not identifiable by looking at the instructions
  - Depend on program state (can change as the program executes)

Load R3 = 0[R6]	(1) Issue, Cache Miss!	(3) Miss serviced
Add R7 = R3 + R9	(4) Issue	
Store R4 → 0[R7]	(5) Issue	
Sub R1 = R1 – R2	(1) Issue	
Load R8 = 0[R1]	(2) Issue, Cache Hit	

But there was a later load…

- [R1] != [R7] -> Load and Store are independent -> Correct execution
- [R1] == [R7] -> Load and Store are dependent -> Incorrect execution

Basic Concepts

- ***Memory Aliasing***: two memory references involving the same memory location (collision of two memory addresses)
- ***Memory Disambiguation***: determining whether two memory references will alias or not
  - Requires computing effective addresses of both memory references
- We say a memory op ***is performed*** when it is done in D$
  - Loads perform in Execute (X) stage
  - Stores perform in Rertire (R) stage

Scheme 1: In-Order Load/Stores

- Performs all loads/stores in-order with respect to each other
  - However, they can execute out of order with respect to other types of instructions

→ Pessimistically, assuming dependence between all memory operations

Load/Store Queue (LSQ)

- Another HW queue, but just for memory ops
- Loads and store instructions are stored in program order
  - Operates as a circular FIFO
  - Allocate on dispatch
  - De-allocate on retirement
- For each instruction, LSQ contains:
  - "Type": Instruction type (S or L)
  - "Addr": Memory addr
    - Addr is generated in dataflow order and copied to LSQ
  - "Val": Data for stores
    - Val is generated in dataflow order and copied to LSQ
- LSQ can be merged with the RS for memory ops
  - i.e., each entry also contains tags and other RS stuff
  - Implementation detail

Scheme 1: In-Order Load/Stores

- Only the instruction at LSQ head can perform, if ready
  - If load, it can perform whenever ready
  - If store, it can perform if it is also at ROB head and ready
- Stores are held for all previous instructions
  - Since they perform in R stage
- Loads are only held for stores
- Easy to implement but killing most of OoO benefits
  → significant performance hit

Scheme 1 Pipeline

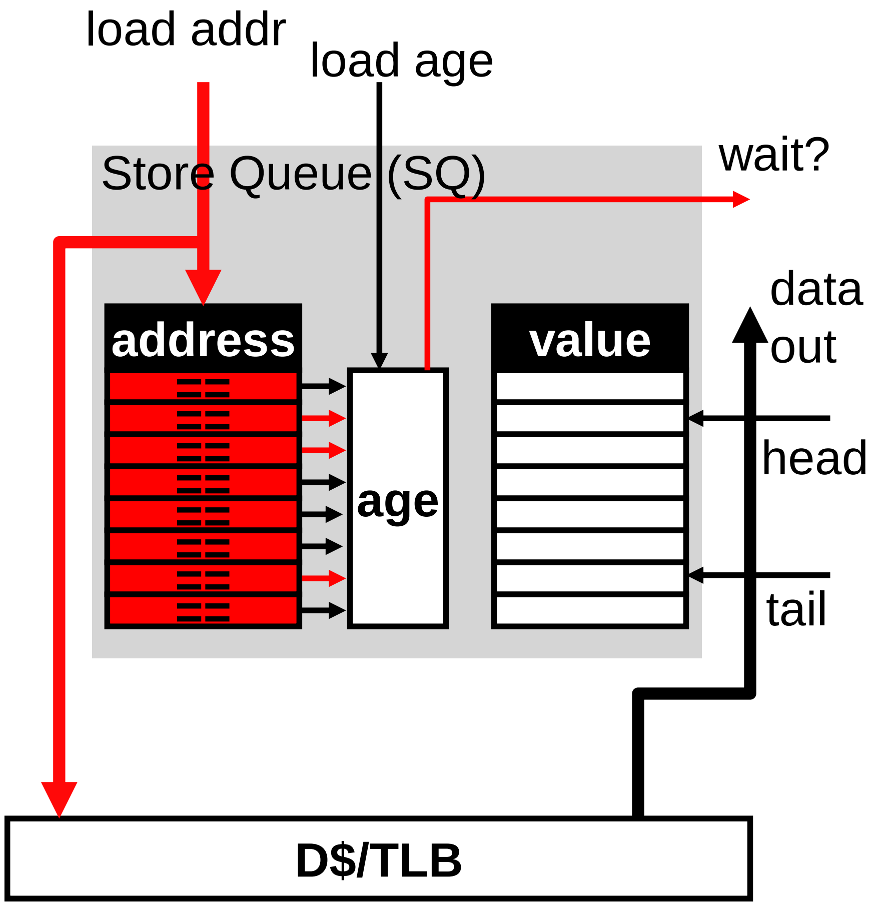
- Stores
  - **Dispatch (D)**
    - Allocate entry at LSQ tail
  - **Execute (X)**
    - Calculate and write address and data into corresponding LSQ entry
  - **Retire (R)**
    - Write address/data from LSQ head to D$, free LSQ head
- Loads
  - **Dispatch (D)**
    - Allocate entry at LSQ tail
  - **Addr Gen (G)**
    - Calculate and write address into corresponding LSQ entry
  - **Execute (X)**
    - Send load to D$ if at the head of LSQ
  - **Retire (R)**
    - Free LSQ head

Scheme 2: Load Bypassing

- Loads can be allowed to bypass older stores (if no aliasing)
  - Requires checking addresses of older stores
  - Addresses of older stores must be known in order to check
- To implement, use separate load queue (LQ) and store queue (SQ)
  - Think of separate RS for loads and stores
- Need to know the relative order of instructions in the queues
  - "Age": new field added to both queues
    - A simple counter incremented during in-order dispatch (for now)

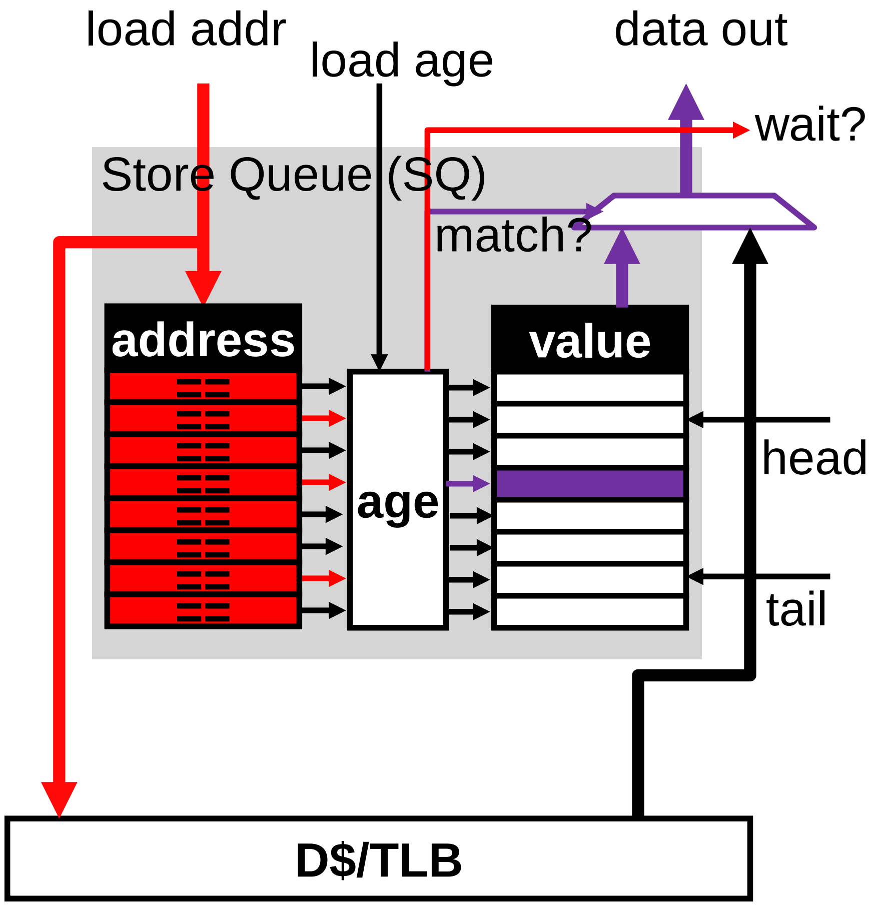
Scheme 2: Load Bypassing

- Loads: for the oldest ready load in LQ, check the addr. of older stores in SQ
  - If any older stores with an *uncomputed* or *matching* addr, load cannot issue
  - To reduce latency, check SQ in parallel with accessing D$
- Requires associative memory (CAM)
- Stores: can always execute when at ROB head



Scheme 3: Load Forwarding + Bypassing

- Loads: can be satisfied from the stores in the store queue on an address match
  - If the store data is available
  - If multiple matches,
    - youngest store older than the load provides the data
- Avoids waiting until the store is sent to the cache
- Stores: can always execute when at ROB head



Schemes 2 & 3 Pipeline

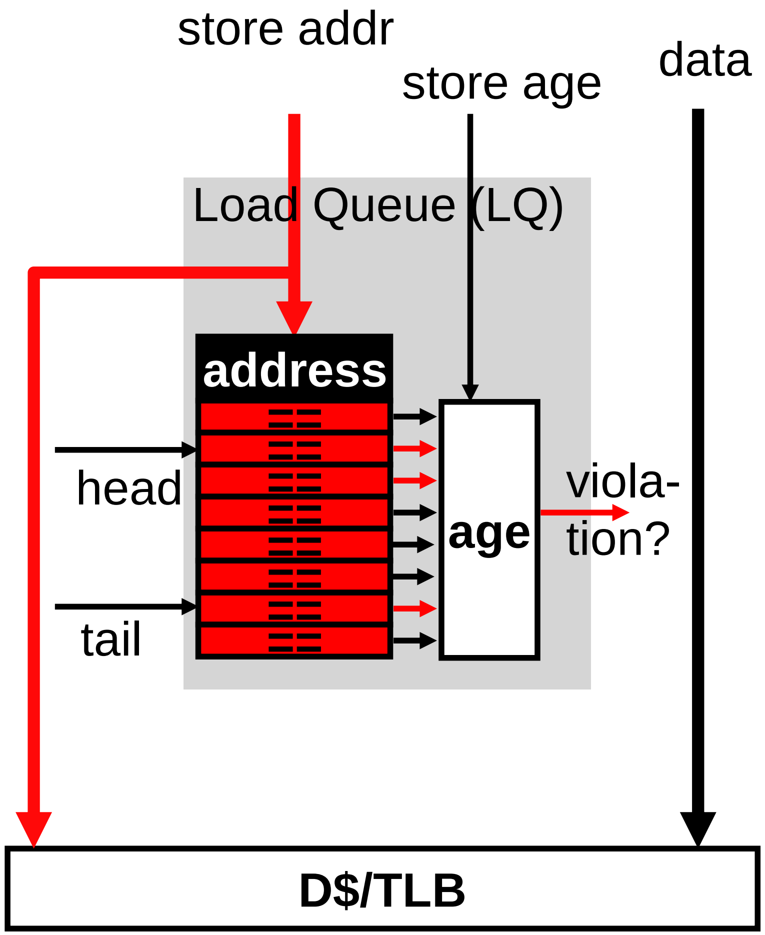
- Stores
  - **Dispatch (D)**
    - Allocate entry at SQ tail and record age
  - **Execute (X)**
    - Calculate and write address and data into corresponding SQ entry
  - **Retire (R)**
    - Write address/data from SQ head to D$, free SQ head
- Loads
  - **Dispatch (D)**
    - Allocate entry at LQ tail and record age
  - **Addr Gen (G)**
    - Calculate and write address into corresponding LQ entry
  - **Execute (X)**
    - Send load to D$ when D$ available and check the SQ for aliasing stores
  - **Retire (R)**
    - Free LQ head

Scheme 4: Loads Execute When Ready

- Drawback of previous schemes:
  - Loads must wait for all older stores to compute their addr.
    - i.e., to “execute”
- Alternative: let the loads go ahead even if older stores exist with uncomputed addr.
  - Most aggressive scheme
- Greatest potential IPC: loads never stall
- A form of speculation: speculate that uncomputed stores are to other addresses
  - Relies on the fact that aliases are rare
  - Potential for incorrect execution
    - Need to be able to “undo” bad loads (mis-speculations)

Detecting Ordering Violations

- Case 1: Older store execs before younger load
  - No problem, HW from Scheme 3 takes care of this
- Case 2: Older store execs after younger load
  - Store scans all younger loads
  - Address match → ordering violation
  - Requires associative search in LQ



Scheme 4 Pipeline

- Stores
  - **Dispatch (D)**
    - Allocate entry at SQ tail and record age
  - **Execute (X)**
    - Calculate and write address and data into corresponding SQ entry
  - **Retire (R)**
    - Write address/data from SQ head to D$, free SQ head
    - Check LQ for potential aliases, initiate “recovery” if necessary
- Loads
  - **Dispatch (D)**
    - Allocate entry at LQ tail and record age
  - **Addr Gen (G)**
    - Calculate and write address into corresponding LQ entry
  - **Execute (X)**
    - Send load to D$ when D$ available and check the SQ for aliasing stores
  - **Retire (R)**
    - Free LQ head

Dealing with Mis-speculations

- Loads are not the only instructions we should worry about
  - Mis-speculated loads propagate wrong values to their dependents
- These must somehow be re-executed
- Easiest: use ROB mechanisms, and flush all instructions after (and including?) the misspeculated load
  - Refetch from the load instruction
  - Load gets forwarded value from store or from D$
  - Correct value propagated when instructions re-execute
- **But** flushing the whole pipeline has high performance overhead
  - Kills ~100 instructions at various stages of execution

Lowering Flush Overhead – Option 1

- ***Selective Re-execution***: re-execute only the dependent instructions
- Ideal case w.r.t. maintaining high IPC
  - No need to re-fetch/re-dispatch/re-rename/re-execute
- Very complicated
  - Need to hunt down only data-dependent instructions
  - Some bad instructions already executed (now in ROB)
  - Some bad instructions didn’t execute yet (still in RS)
- Pentium 4 does something like this (called “replay”)

Lowering Flush Overhead – Option 2

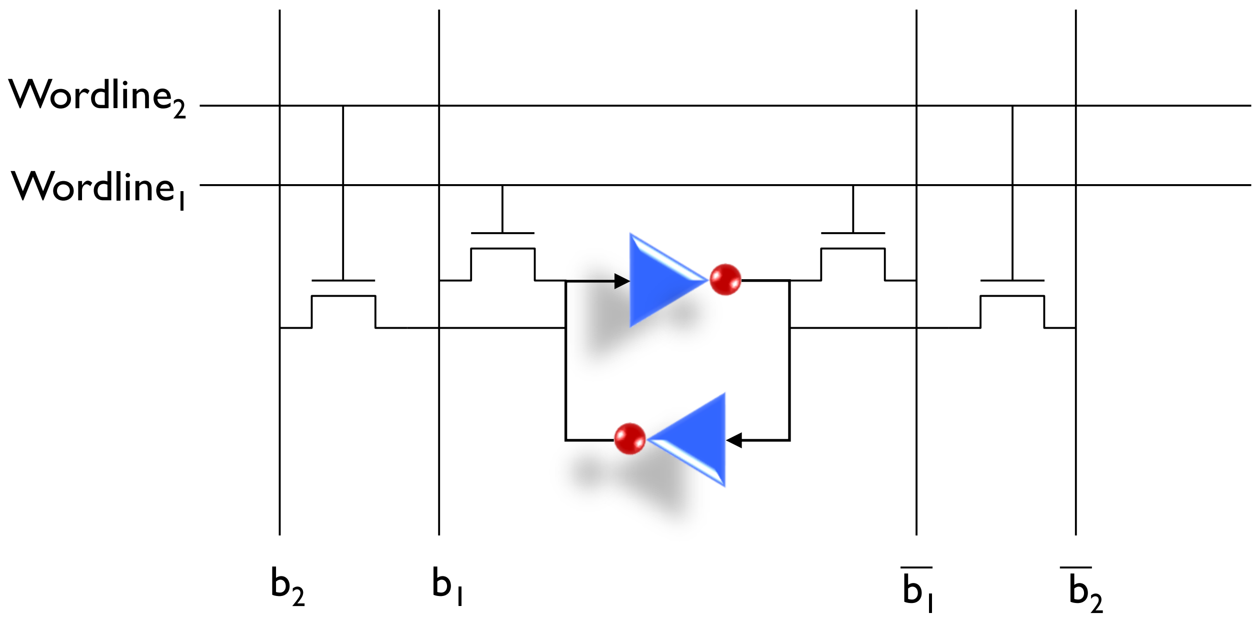
- Observation: loads/stores that cause violations are "stable"
  - Dependences are mostly program based, program doesn't change
- ***Alias Prediction***: predict which load/store pairs are likely to alias
  - Use a hybrid scheme
  - Predict which loads, or load/store pairs will cause violations
    - Use Scheme 3 for those
    - Use Scheme 4 with pipeline flush for the rest

Other Memory-Flow Tricks in OOO Super-Scalars

Multi-Port Caches

- Super-scalars might make multiple parallel cache accesses
  - Core can make multiple L1$ access requests per cycle
    - E.g., 2 simultaneous L1 D$ accesses in Intel processors
  - Multiple cores can access LLC at the same time
- Cache should have multiple access ports
- How to process simultaneous requests on different ports?
  - Design SRAMs with multiple ports
    - Big and power-hungry
  - Split SRAMs into multiple banks
    - Can result in delays, but usually not

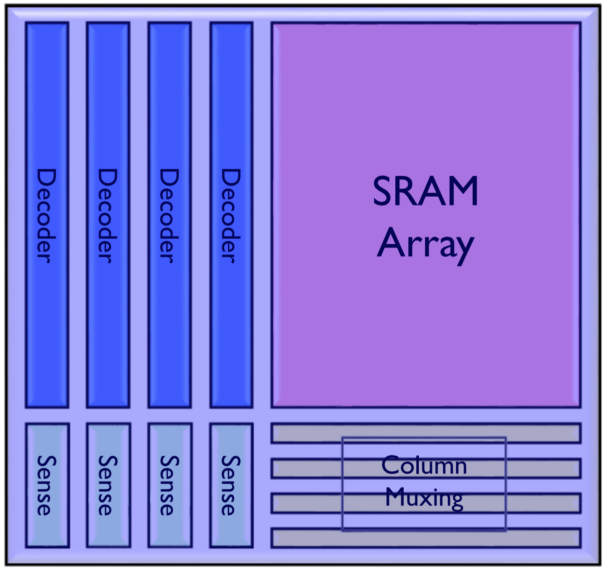
Multi-Port SRAMs

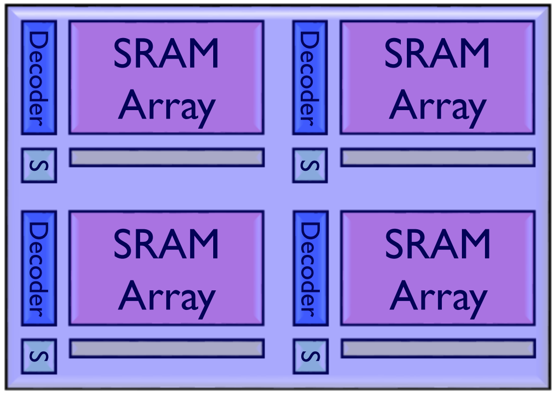


Wordlines = 1 per port
Bitlines = 2 per port

$\Longrightarrow$ Area = $O(\text{ports}^2)$

Multi-Port SRAMs vs. Banked SRAMs





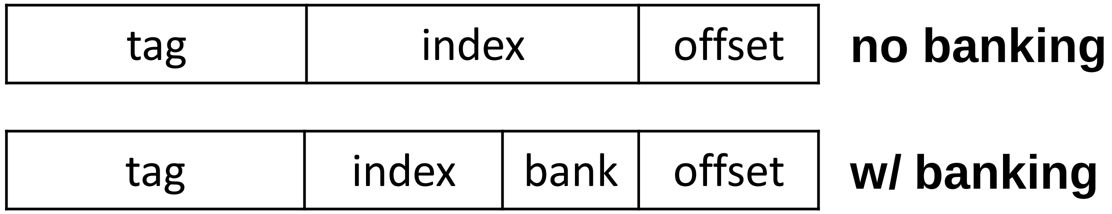
4 ports
Big (and slow)
Guarantees concurrent access

4 banks, 1 port each
Each bank small (and fast)
Conflicts (delays) possible

How to decide which bank to go to?

Bank Conflicts

- Banks are *address interleaved*
  - For block size **b** cache with **N** banks…
  - Bank = (Address / **b**) % **N**
    - Looks more complicated than is: just low-order bits of index



- Banking can provide high bandwidth
- But only if all accesses are to different banks
  - For 4 banks, 2 accesses, chance of conflict is 25%
  - 8 banks a good trade-off between complexity and conflict ratio

Non-Blocking Caches

- So far, we assumed caches stop accepting new requests when there is a cache miss
  - i.e., cache waits until miss is resolved
- **Observation 1**: misses usually happen in bursts; it is helpful to overlap latencies of multiple parallel misses
- **Observation 2**: main memory system can supports a large number of in-flight requests
- **Idea**: let's make caches ***non-blocking***
  - i.e., cache keeps accepting new requests while waiting for misses to be handled

Implementing Non-Blocking Caches (1)

- On a miss:
  - Send the request to main memory, and
  - Put the miss information in a ***Miss Status Holding Register*** (***MSHR***)
    - Instruction tag (ROB#), address, load-or-store, store value, …
- When memory response arrives:
  - Merge memory response data with store value (if store miss) and write to cache
  - Broadcast results on CDB (if load miss)

Implementing Non-Blocking Caches (2)

- If a new load/store request to an already missing line
  - Can merge the new miss into existing MSHR
    - Instead of sending another request to main memory
  - MSHR should be big enough to keep info for multiple pending misses to the same line
- Also, can have several MSHRs to support multiple missing cache lines
  - E.g., 11 at L1 level in current Intel Xeon (server) processors