COMP3710 (Class # 5176)
Special Topics in Computer Science
Computer Microarchitecture

Convener: Shoaib Akram
shoaib.akram@anu.edu.au

Australian
National
University

Plan

Week 5: In-order to out-of-order (OOO) transformation

Week 6: OOO v.1 (CDC 6600 Scoreboard) and OOO v.2 (IBM 360/91)

Week 7: OOO v.3 a.k.a. Physical Register File (PRF) microarchitecture

Week 7: Load/Store queue and the load/store execution lane

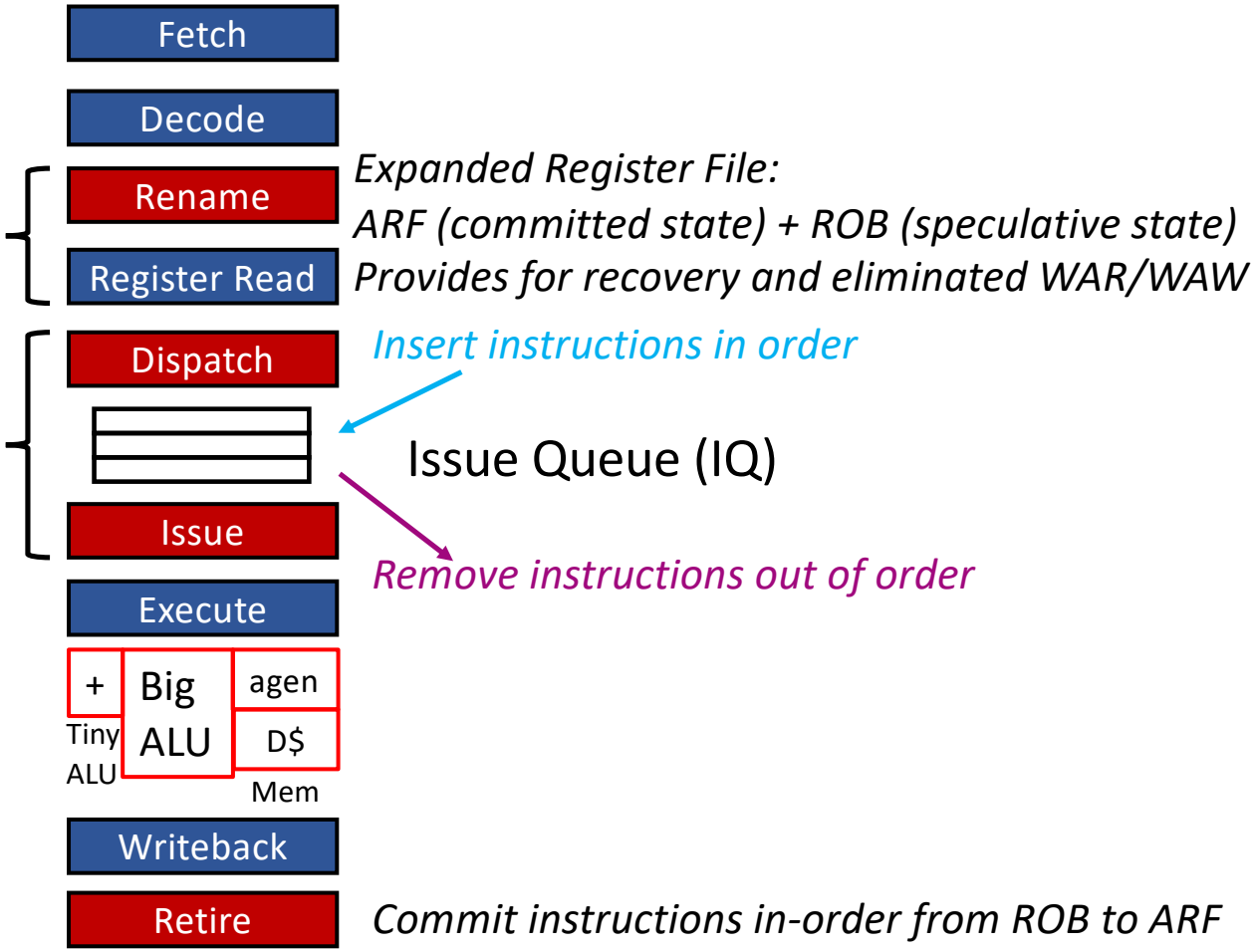
Week 8+9: Cache design/implementation (assignment # 2)

Remaining topics: ★Virtual memory★, SMT, Multicores, DRAM, NVM

Two OOO Designs

ARF+ROB

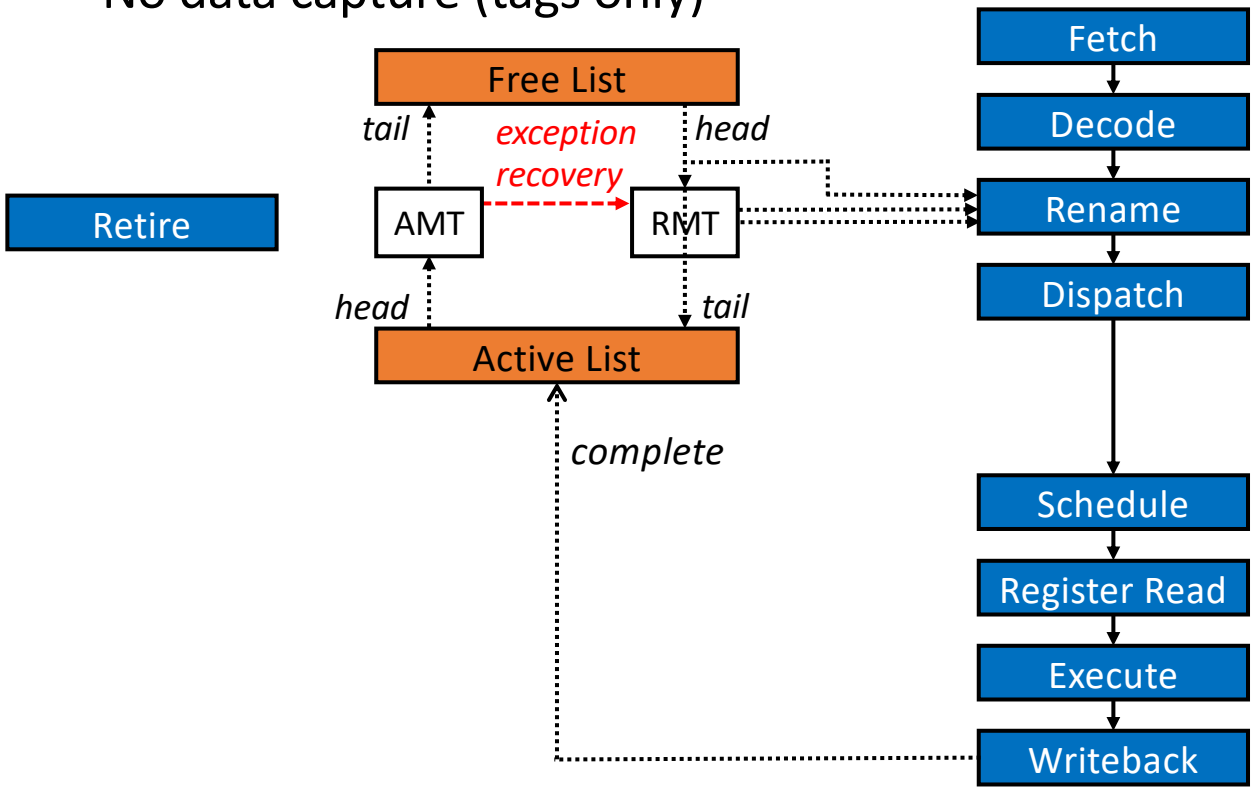
- With data capture
- Capture data in register read stage

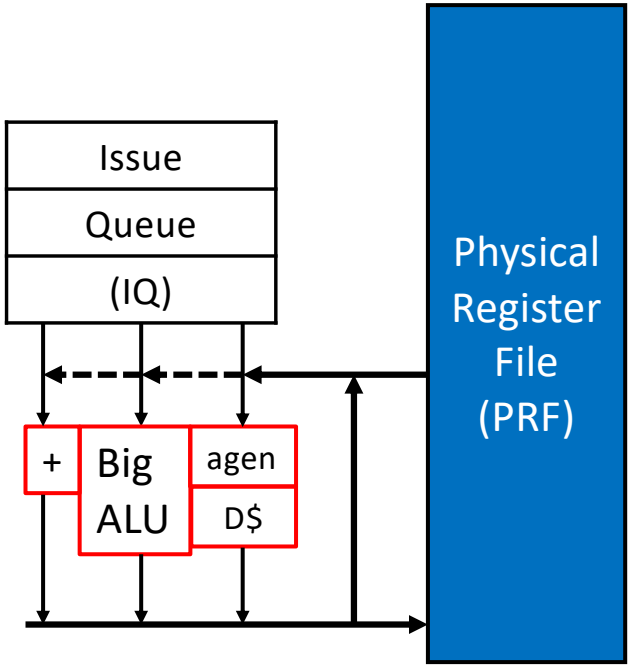


Two OOO Designs

PRF

- No data capture (tags only)



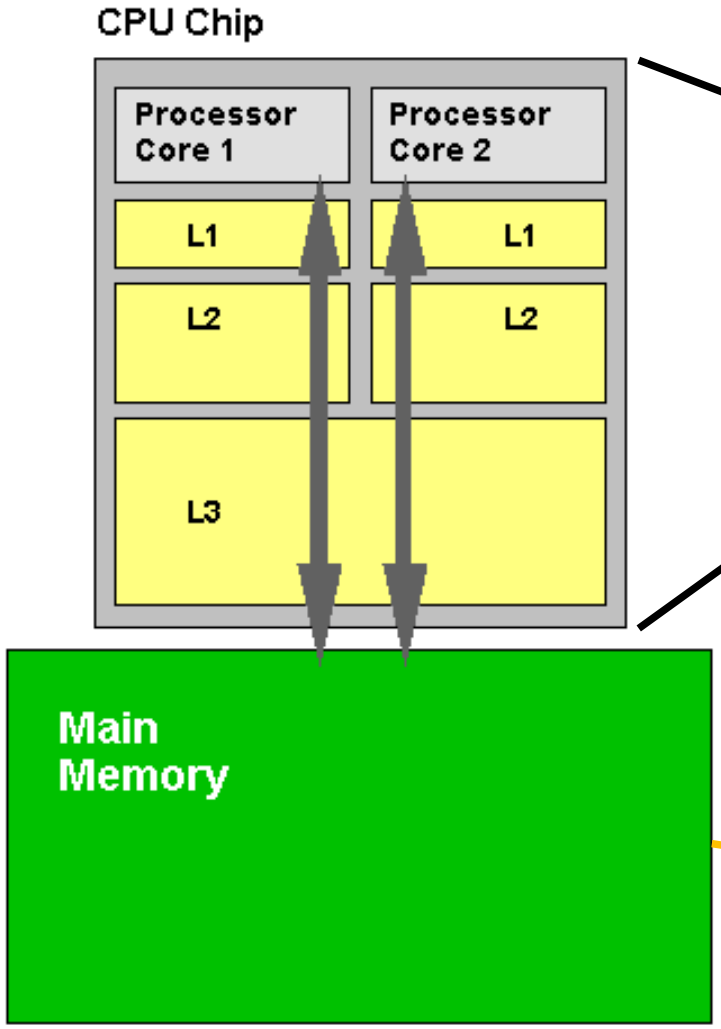


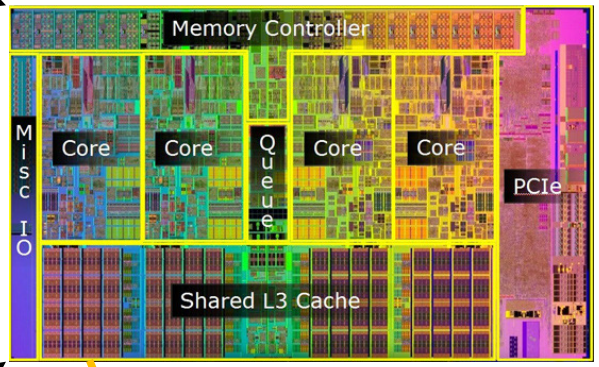
OOO Load/Store Execution

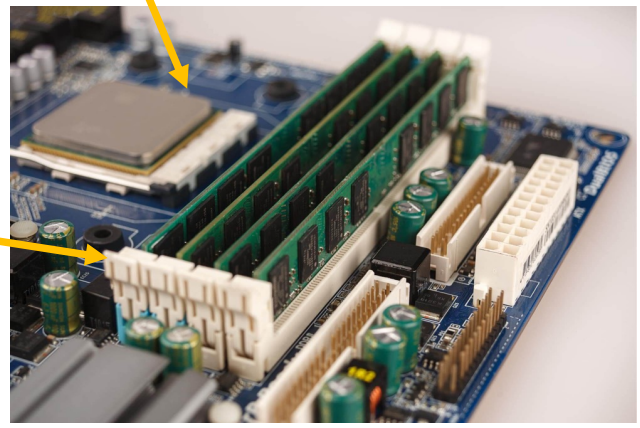
- **Datapath of memory operations consist of**
  - Register file, Issue queue, Active list
    - Core components of the OOO CPU
  - Data cache (D$, SRAM)
    - On-chip and tightly integrated with the CPU
    - Microarchitectural (performance) optimization for speed because memory is too slow and regs too few
  - Main memory (DRAM)
    - Off-chip and slow to access
- **Caches deliver speed but introduce new issues**
  - Must maintain a coherent copy of data in cache and memory

OOO Load/Store Execution

- Multicore CPU
  - Private Level 1 (L1) cache
  - Private Level 2 (L2) cache
  - Shared Level 3 (L3) cache
- Lower in the hierarchy
  - Slower
  - More capacity







OOO Load/Store Execution

- **Memory instructions (load and store) benefit from OOO execution**
  - Loads are at the top of the dependence chains
  - **RISC ISA:** More registers, most variables allocated to regs
  - **CISC ISA:** Fewer registers, many operands from memory
- **Stores are sent to the data cache on retirement**
  - Ensures precise recovery
- **Loads can be issued out of order w.r.t. loads and stores**
  - Ok, if there are no dependences (i.e., different memory addresses)
  - Not Ok if there are dependences

Fundamental Concepts

- **Memory aliasing**
  - Two memory references (loads or stores) involving the same memory location (address collision)
  - Aliasing more common in CISC ISAs
- **Memory disambiguation**
  - Determining whether two memory references will collide (alias) or not
  - Requires address calculation
- **Performing a memory operation**
  - A memory operation is performed when it is done in the L1-D cache
  - Loads perform in the execute stage
  - Stores perform in the retirement stage

OOO Load/Store Execution: Problem

Counter	Op.	src/dst	Address
1	Load	r1	#16(r7)
	…		
2	Store	r2	#12(r11)
	…		
3	Load	r3	#4(r19)
	…		
4	Load	r4	#0(r8)
	…		
5	Store	r5	#1(r9)
	…		
6	Load	r6	#5(r11)

- Load # 3 executes before Store # 2
  - r19 is ready but r2 is not
- Load # 3 gets data from cache (**speculation**)
- Later on: Store # 2 writes to data cache
- Suppose [4 + r19] and [12 + r11] are aliases
  - [4 + r19] = [12 + r11] = 0xFF220000
- We have an "ordering violation"
  - Load # 3 should get the most recent value from Store # 2 for correct execution
- Load # 3 is a misspeculated load
  - Incorrect execution (set mispredict bit in ROB or AL)

Load/Store Execution: Approaches

- **In Order Execution**
  - Execute memory operations in program order but OOO w.r.t. other instructions
  - Pessimistic and slow (assumes all memory ops. conflict)
- **Load bypassing**
  - Check all uncompleted/pending stores before issuing the load
  - Wait if there are older uncompleted stores or ones with a matching address
- **Load bypassing + Forwarding**
  - Same as above, but forward the store value before it reaches the data cache
  - Still slow (Need more aggression from our processors!!!!)
- **Execute when ready**
  - Loads execute when their address is ready
  - Make a best effort to get value from store queue (many potential stores with matching addresses)
  - Otherwise, detect violation and initiate recovery (like branch mispredictions)
  - Once we have machinery for recovery, we can speculate on a # things (powerful idea!)
    - Branches and loads relevant to this course

Load/Store Queue (LSQ)

- **To execute loads and stores in program order**
  - We need a load/store queue (LSQ)
- **LSQ is a circular FIFO**
  - Stores memory operations in program order
  - Allocate entry at tail on dispatch
  - Remove entry from head at retirement
  - Keeps type (L/S), address, and values (for stores)
  - Addresses and values are generated in dataflow order and copied to LSQ
- **Think of LSQ as an issue queue for loads/stores**

Scheme # 1: In-Order Load/Store Exec

- **Perform 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 b/w all memory operations**
  - Hardware is simple
  - Too slow (giving up the OOO advantage)

Scheme # 1: In-Order Load/Store Ex

- **Only the instruction at the 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 Retire stage
- Loads are only held for stores
- **Easy to implement but killing most of OOO benefits**
  - Significant performance hit

In-Order Load/Store Pipeline

- **Perform instructions from the LSQ head**
  - Load can perform whenever ready
  - Store can perform if also at ROB/AL head (precise recovery)

Stores

- Dispatch (D)
  - Allocate entry at LSQ tail
- Execute
  - Calculate and write address and data into the LSQ slot
- Retire
  - Write to **D $**, free LSQ head

Loads

- Dispatch (D)
  - Allocate entry at LSQ tail
- Agen
  - Calculate and write address into the LSQ slot
- Execute
  - Send load to **D$** if at LSQ head
- Retire: 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 a 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 (will do something “clever” later)

Scheme # 2: Load Bypassing

- **Loads: for the oldest ready load in LQ, check the address of older stores in SQ**
  - If any older stores with an uncomputed or matching address, load cannot issue
  - Check SQ in parallel with D$
- Requires associative memory (CAM)
- Stores: can always execute when at ROB head
- **Advantage: No need to wait for stores to drain from the SQ**
  - Cache accesses take a few cycles
  - There maybe older instructions (relative to store) that are not retired yet

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
- **Advantage: Avoids waiting until the store is sent to the cache**
  - On the other hand, bypassing needs to wait for store to reach D $ if there is an address match
- Stores: can always execute when at ROB head

Pipeline: Load Forwarding + Bypassing

Stores

- Dispatch (D)
  - Allocate entry at SQ tail and record age
- Execute
  - Calculate and write address and data into the SQ slot
- Retire
  - Write address/data from SQ head to D $, free SQ head

Loads

- Dispatch (D)
  - Allocate entry at LQ tail and record age
- Agen
  - Calculate and write address into the corresponding LQ slot
- Execute
  - Send load to D$ when D$ is available and check the SQ for aliasing stores
- Retire: 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-speculation)

Recording Age

Op #	Op.	src/dst	Address
1	Load	r1	#16(r7)
2	Store	r2	#12(r11)
3	Load	r3	#4(r19)
4	Load	r4	#0(r8)
5	Store	r5	#1(r9)
6	Load	r6	#5(r11)

Operation #	Op/ Queue	LQ_IDX (dispatch)	SQ_IDX (dispatch)	LQ_tail	SQ_tail
1	Load/LQ	1	n/a	1→2	1
2	Store/SQ	2	1	2	1→2
3	Load/LQ	2	1	2→3	2
4	Load/LQ	3	1	3→4	2
5	Store/SQ	4	2	4	2→3
6	Load/LQ	4	2	4→5	3

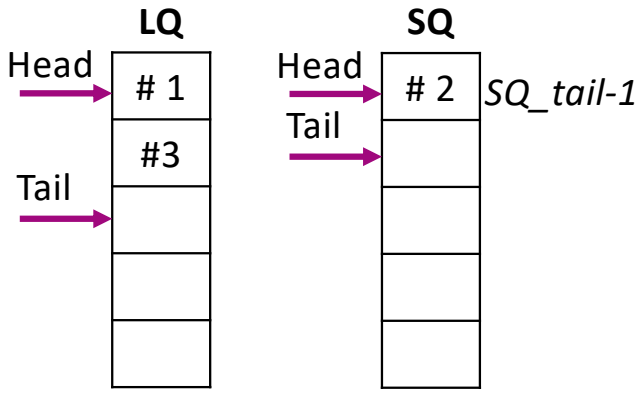
- **Loads and stores have an LQ Index (LQ_IDX) and SQ Index (SQ_IDX)**
  - To help them remember their age relative to all other operations

Recording Age

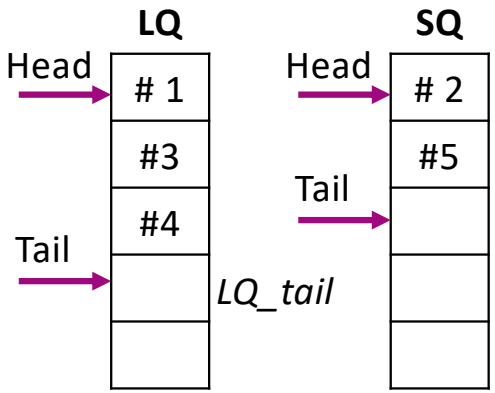
Op #	Op.	src/dst	Address
1	Load	r1	#16(r7)
2	Store	r2	#12(r11)
3	Load	r3	#4(r19)
4	Load	r4	#0(r8)
5	Store	r5	#1(r9)
6	Load	r6	#5(r11)

Operation #	Op/ Queue	LQ_IDX (dispatch)	SQ_IDX (dispatch)	LQ_tail	SQ_tail
1	Load/LQ	1	n/a	1→2	1
2	Store/SQ	2	1	2	1→2
3	Load/LQ	2	1	2→3	2
4	Load/LQ	3	1	3→4	2
5	Store/SQ	4	2	4	2→3
6	Load/LQ	4	2	4→5	3

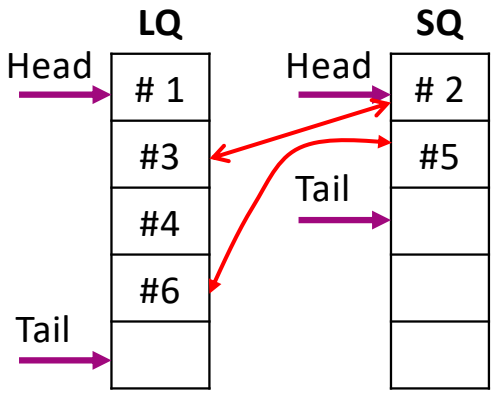
Load # 3 is dispatched
LQ_Index = 2, SQ_Index = 1



Store # 5 is dispatched
SQ_Index = 2, LQ_Index = 4



Load # 6 is dispatched
LQ_Index = 4, SQ_Index = 2



Split LQ/SQ

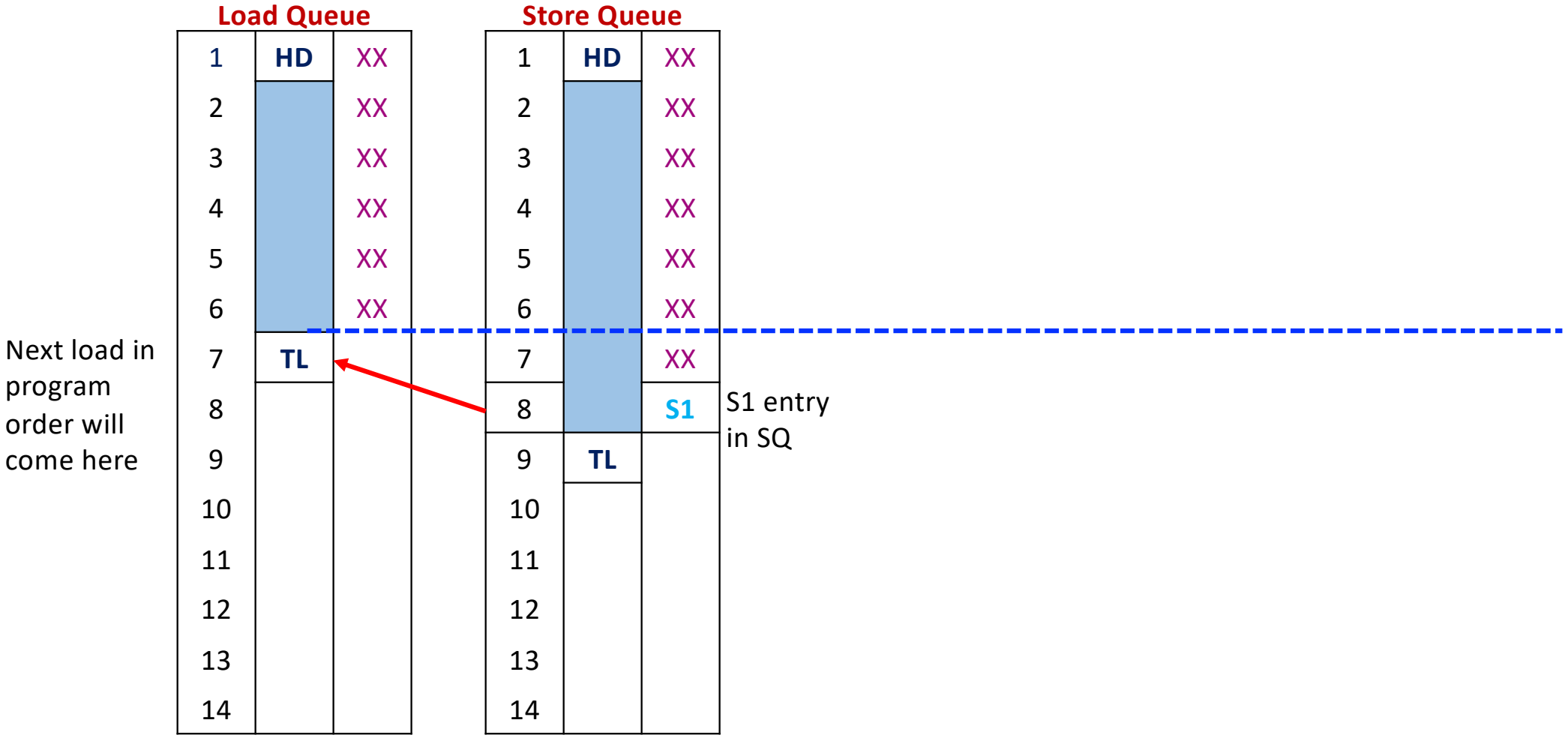
- **Last three schemes need a split LSQ**
  - We will only discuss the most aggressive scheme in detail
- **To execute loads whenever their address is ready**
  - Load needs to remember SQ_tail – 1 (SQ_index)
    - Stores younger than the load in program order (i.e., those after SQ_index) are irrelevant for this load
  - Store needs to remember LQ_tail
    - Loads older than the store are irrelevant for this store

Split LQ/SQ

- **Store Queue: All active stores in program order**
  - Stores are speculative until they reach the head of AL
  - SQ commits stores to **D$** non-speculatively and in program order
  - *Loads search SQ for store values on which they depend*
    - *Intent is to get a value*
- **Load Queue: All active loads in program order**
  - Loads execute OOO w.r.t. prior stores
  - Executed loads get wrong value in case of aliasing
    - If an older store has not been executed yet
  - *Stores search LQ for mispredicted loads*
    - *Intent is to detect ordering violations and cancel a load*

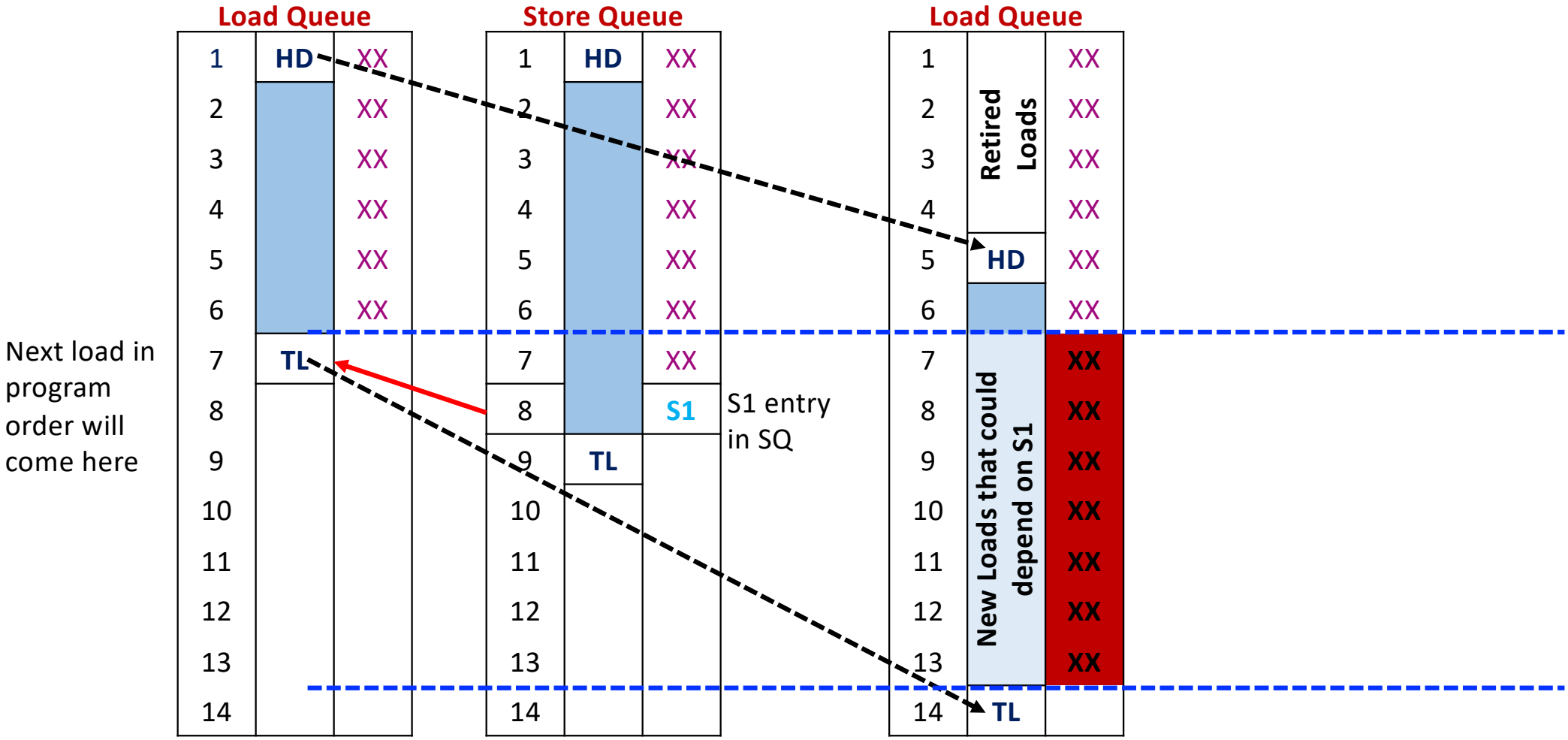
Store searches LQ for mispredicted loads

For S1 on dispatch:
SQ_Index = 8 (Allocate then Tail++)
LQ_Index = 7



Store searches LQ for mispredicted loads

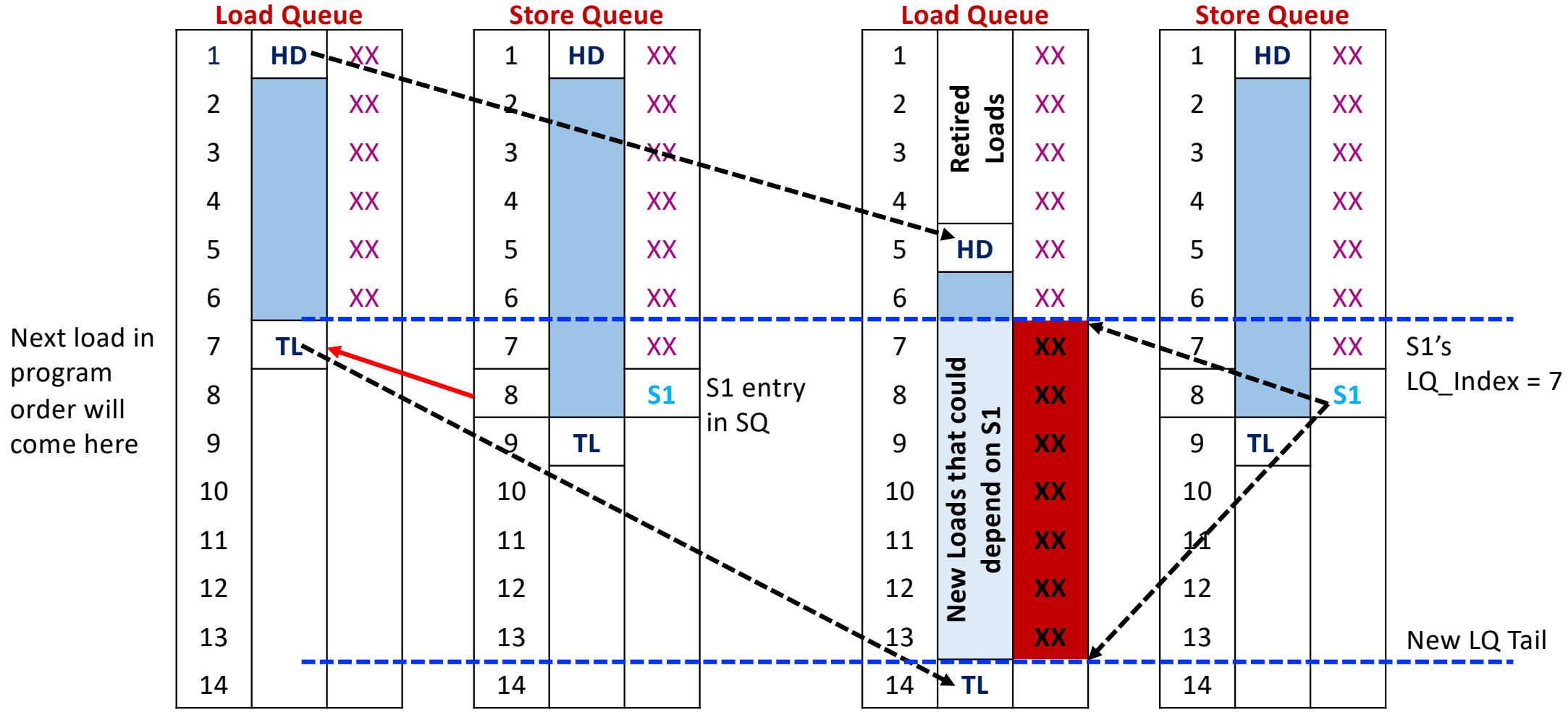
For S1 on dispatch:
SQ_Index = 8 (Allocate then Tail++)
LQ_Index = 7



Store searches LQ for mispredicted loads

For S1 on dispatch:
SQ_Index = 8 (Allocate then Tail++)
LQ_Index = 7

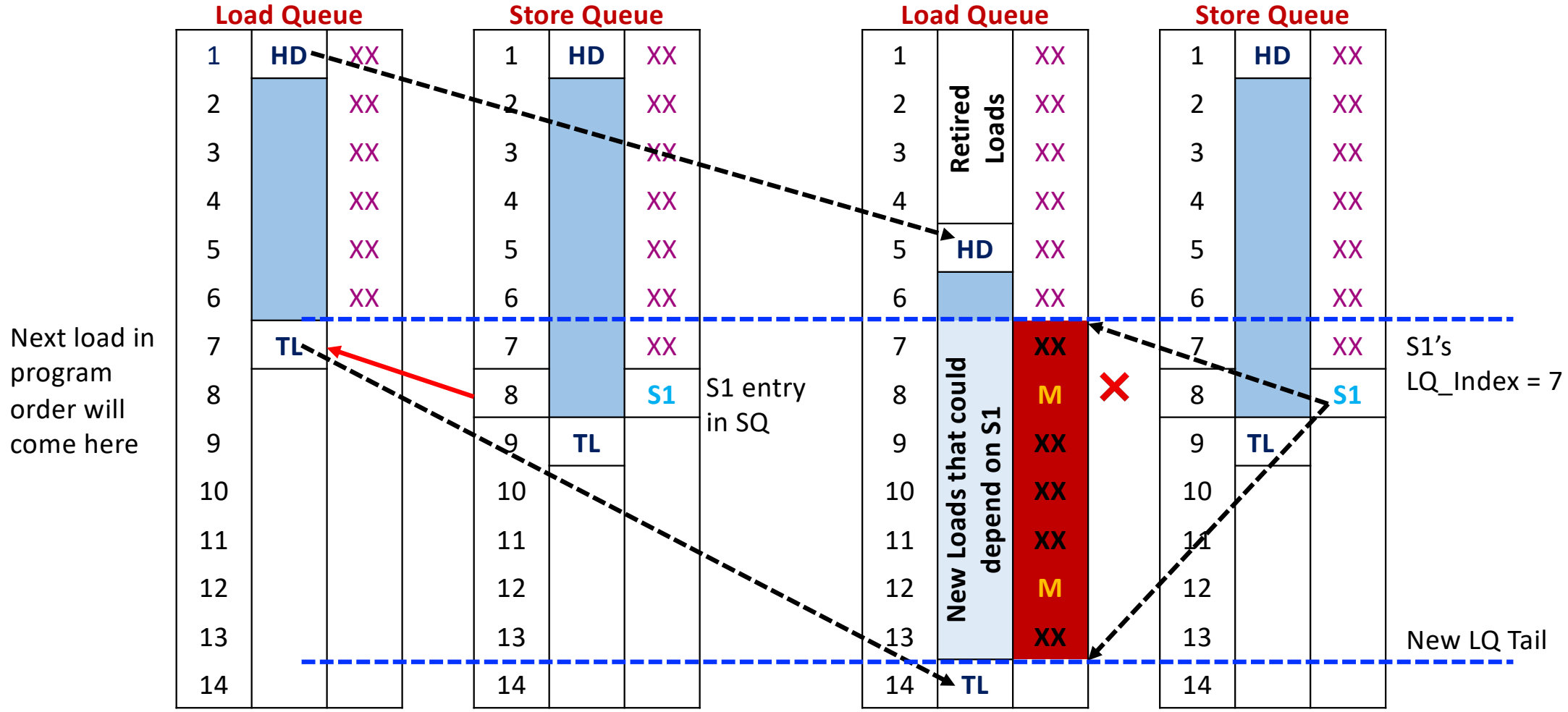
When S1 executes:
Compare its address to all addresses in LQ b/w 7 & LQ_tail



Store searches LQ for mispredicted loads

For S1 on dispatch:
SQ_Index = 8 (Allocate then Tail++)
LQ_Index = 7

When S1 executes:
Compare its address to all addresses in LQ b/w 7 & LQ_tail

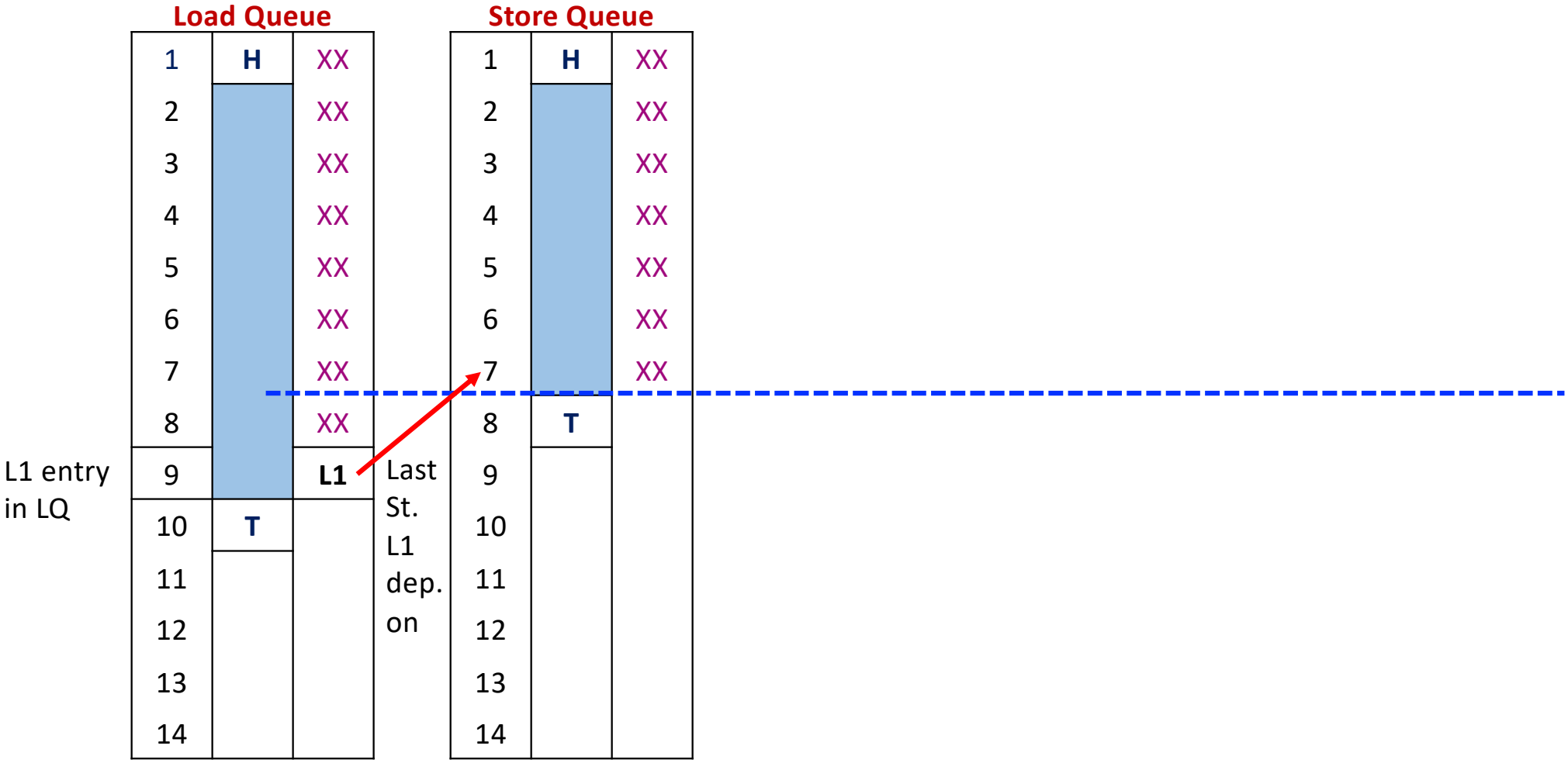


Race Condition

- **Store # 8 and Load # 8 execute in the same cycle. The hardware must not allow the following scenario**
  - Store (#8) searches LQ and does not find a matching load (#8)
  - Store executes
  - In the same cycle, Load # 8 gets (stale) value from data cache
  - Race condition! ***Disaster*** An entire chain/program is fed wrong values!
- **Solutions:**
  - Address calculation in first half of clock cycle and LQ/SQ search in the second half
  - The issue (select logic) can prevent certain combinations from executing in the same cycle (ILP limiter, not used in high-end processors)
  - Each load sets a special bit in LQ just before executing. If the load is in the store's "special interest group," store delays for one cycle

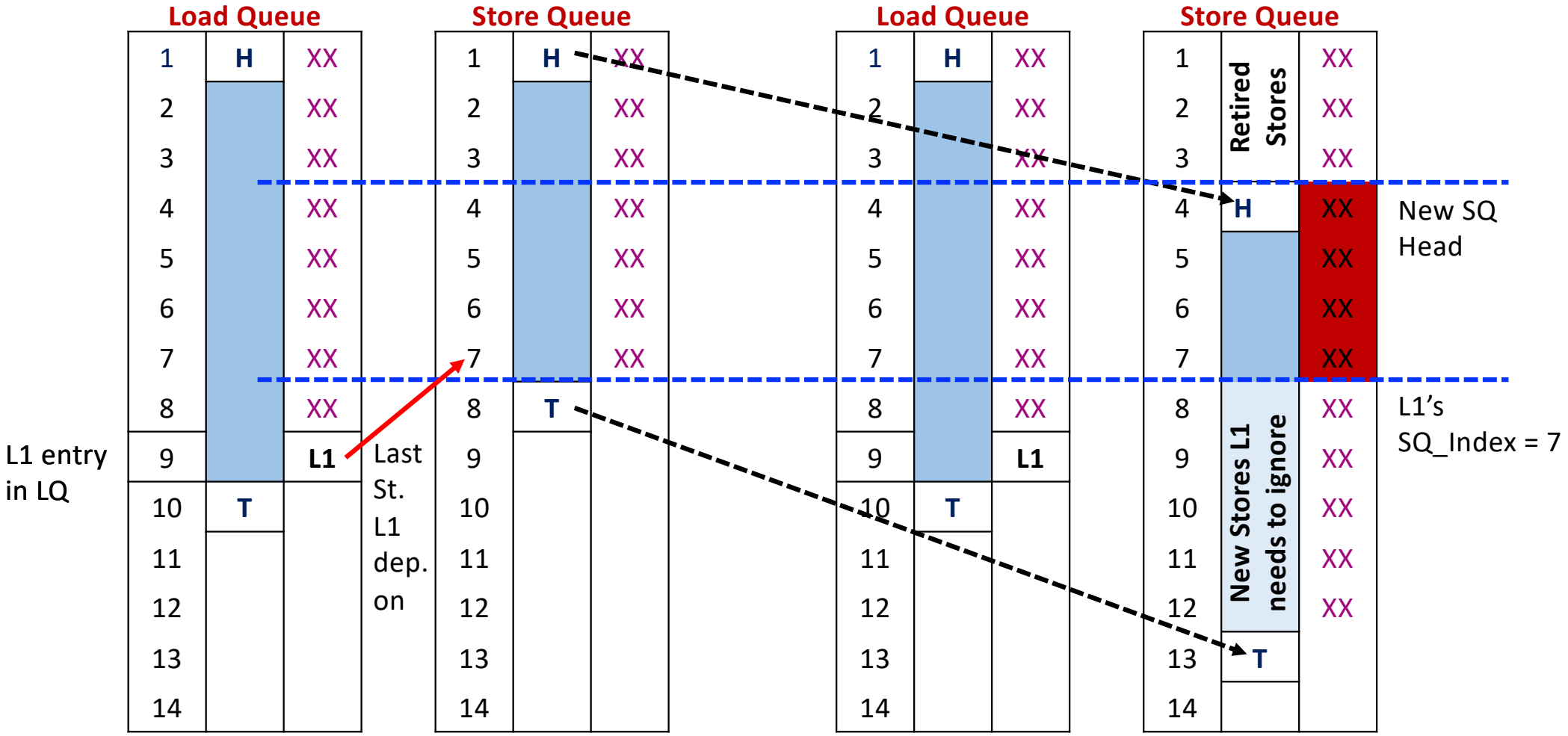
Load searches SQ for values

For L1 on dispatch:
LQ_Index = 9 (Allocate then Tail++)
SQ_Index = 7 (SQ_tail-1)



Load searches SQ for values

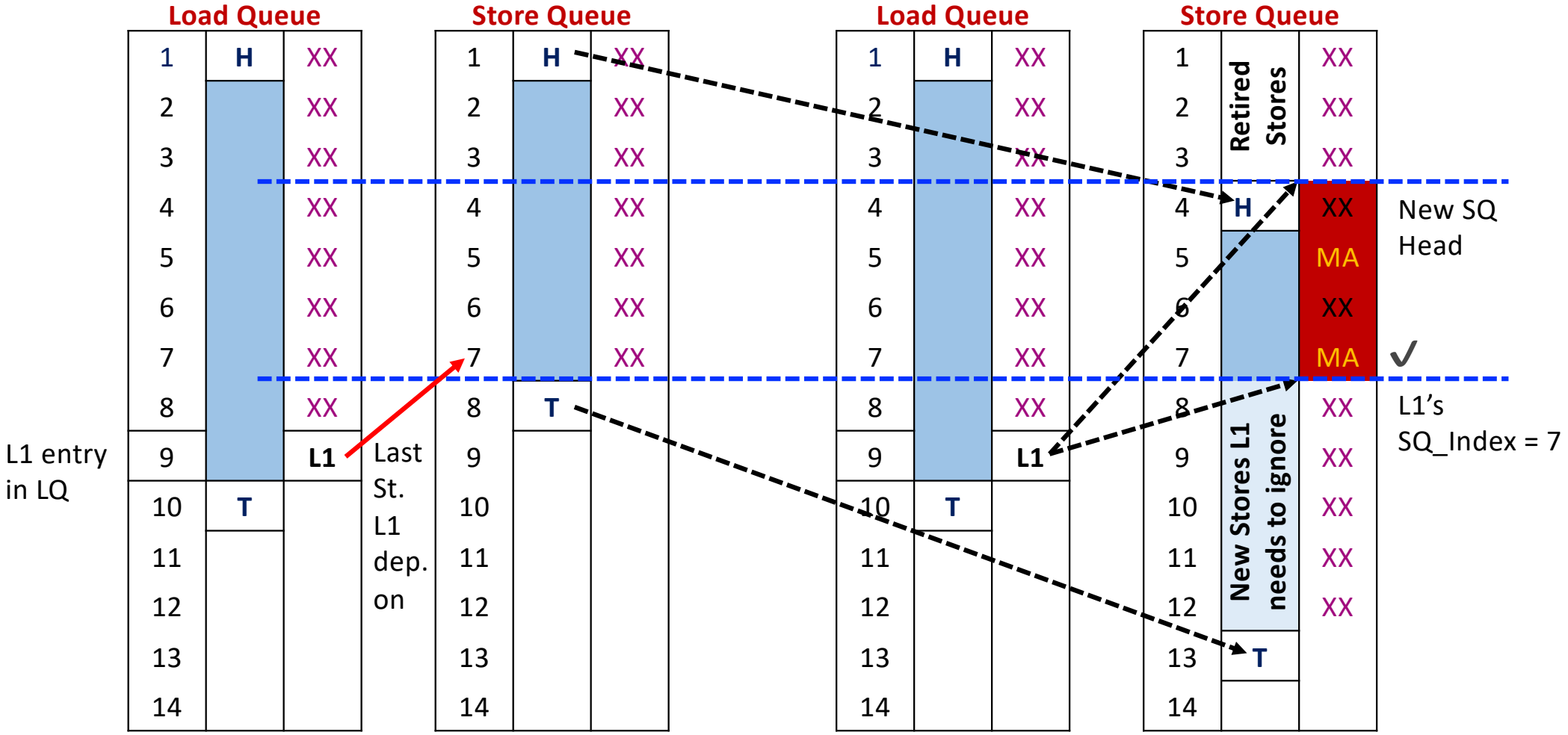
For L1 on dispatch:
LQ_Index = 9 (Allocate then Tail++)
SQ_Index = 7 (SQ_tail-1)



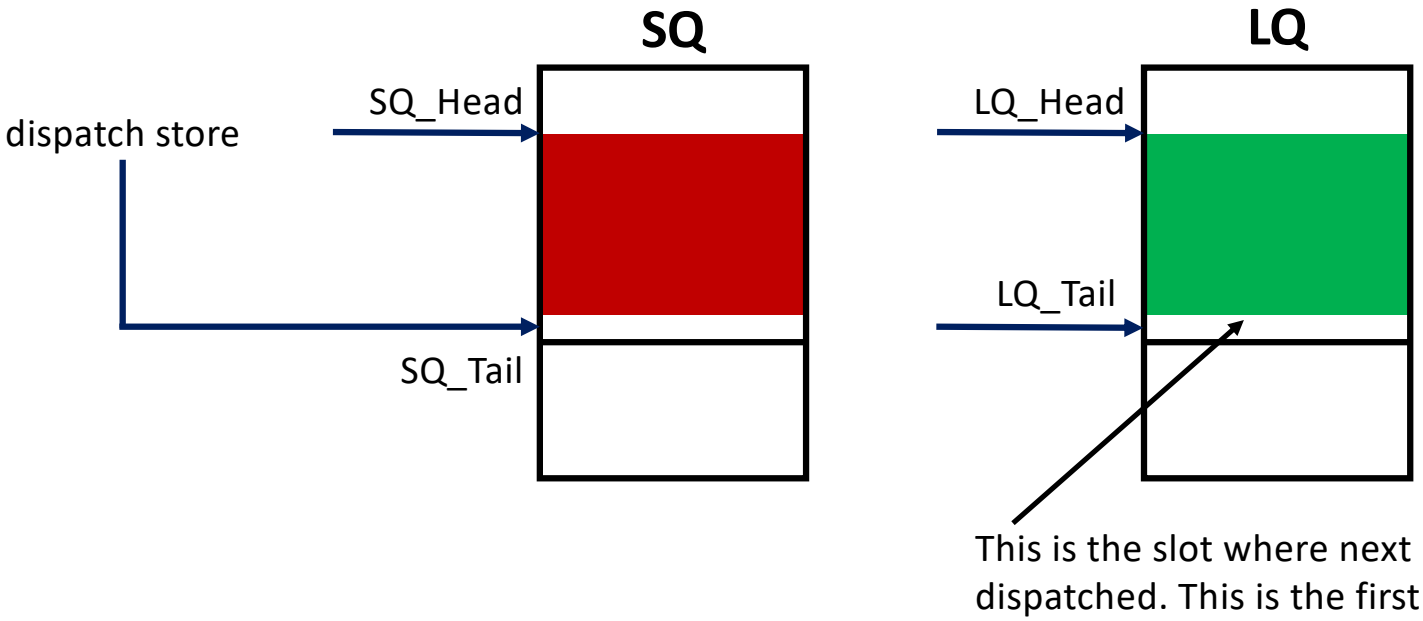
Load searches SQ for values

For L1 on dispatch:
LQ_Index = 9 (Allocate then Tail++)
SQ_Index = 7 (SQ_tail-1)

When L1 executes:
Compare its address to all addresses in SQ b/w current SQ head & 7 (L1's SQ_index)



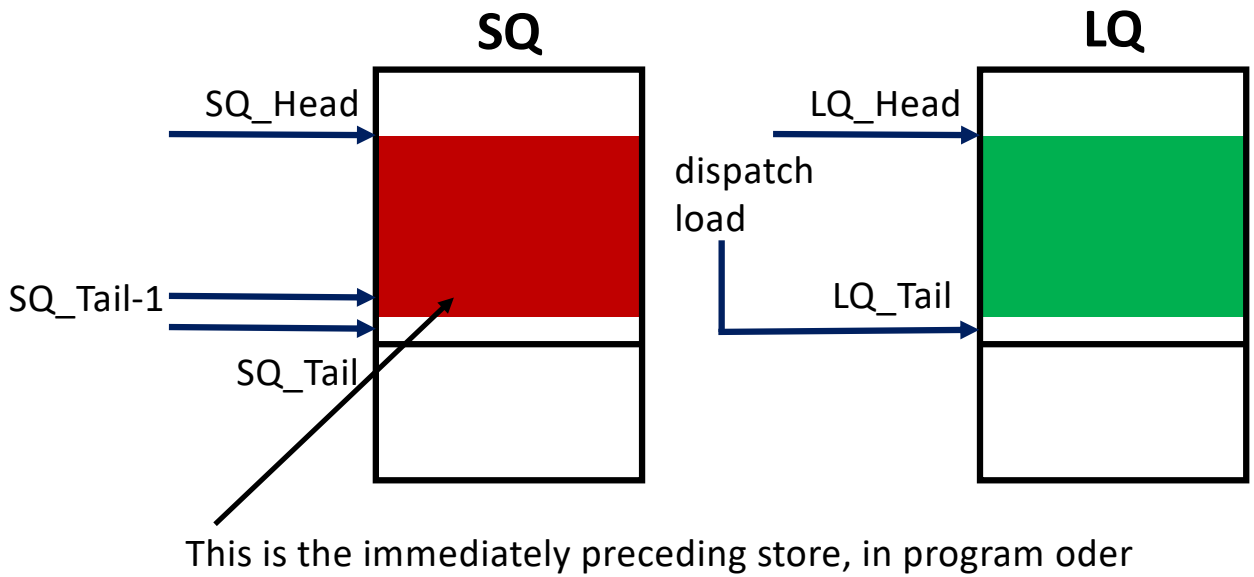
Split LQ/SQ: Store Index Mgmt



At dispatch time, the store instruction inherits the following indices:

1. SQ_index = SQ_tail: The store's entry in the SQ
   When the store executes later, it uses SQ_index to place its address and value in the SQ. These are needed for store-load forwarding and committing stores.
2. LQ_index = LQ_tail: Index of first load after the store, in program order
   When the store executes later, it searches the LQ for mispredicted loads: loads after the store, in program order, that depend on the store but executed before the store. Loads between LQ_index and LQ_tail are after the store in program order

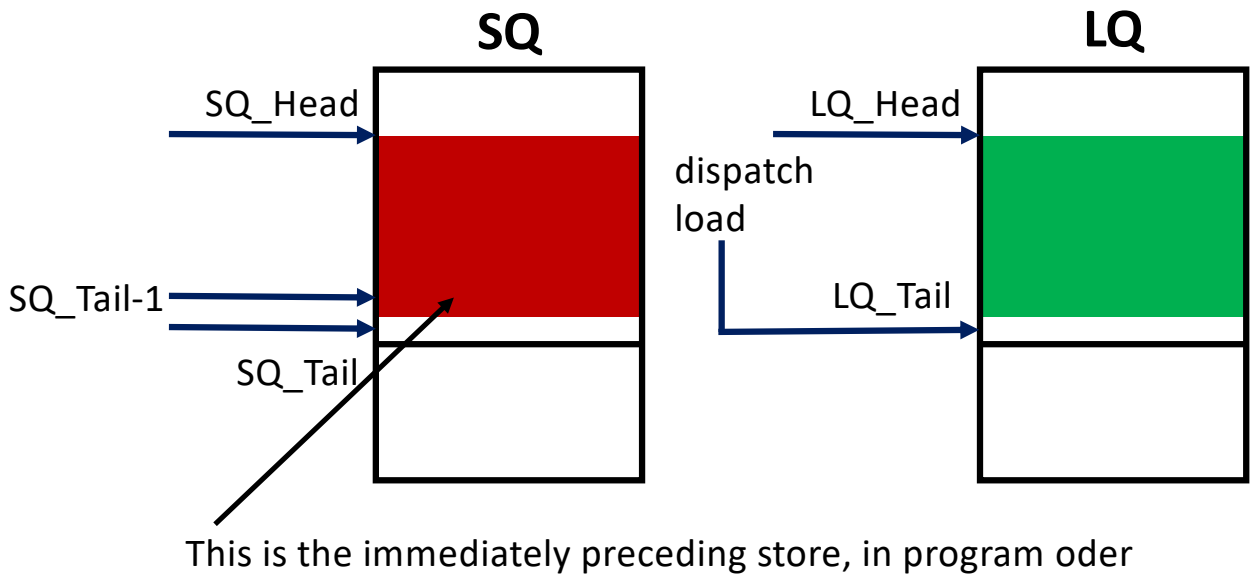
Split LQ/SQ: Load Index Mgmt



At dispatch time, the load instruction inherits the following indices:

1. LQ_index = LQ_tail: The load's entry in the LQ
   When the load executes later, it uses LQ_index to place the address in LQ. The address is needed to detect mispredicted loads
2. SQ_index = SQ_tail-1: Index of immediately preceding store, in program order
   When the laod executes later, it searches the SQ for a dependence on a prior store. It only considers stores between SQ_head and SQ_index: these are the stores before the load, in program order

Split LQ/SQ: Load Index Mgmt

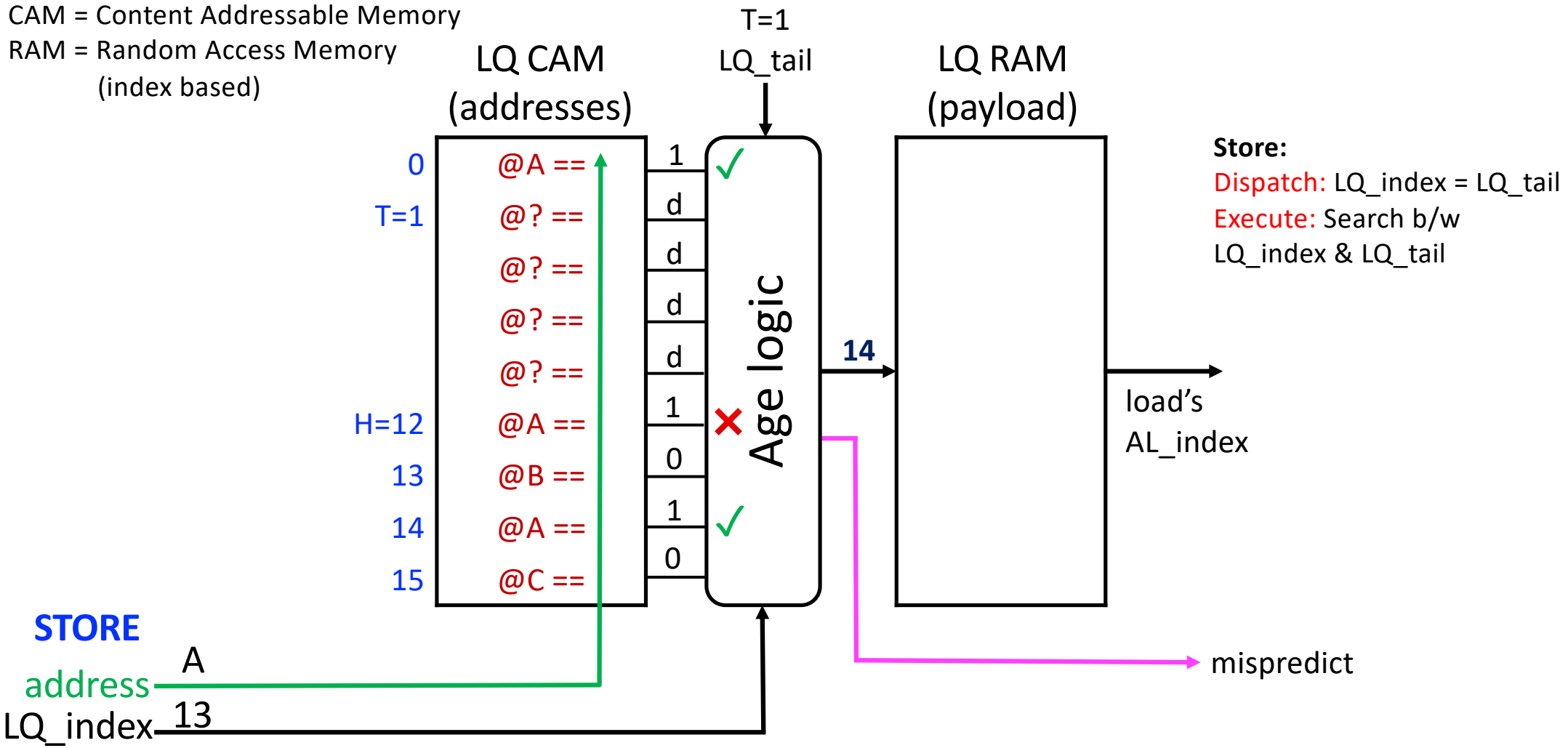


At dispatch time, the load instruction inherits the following indices:

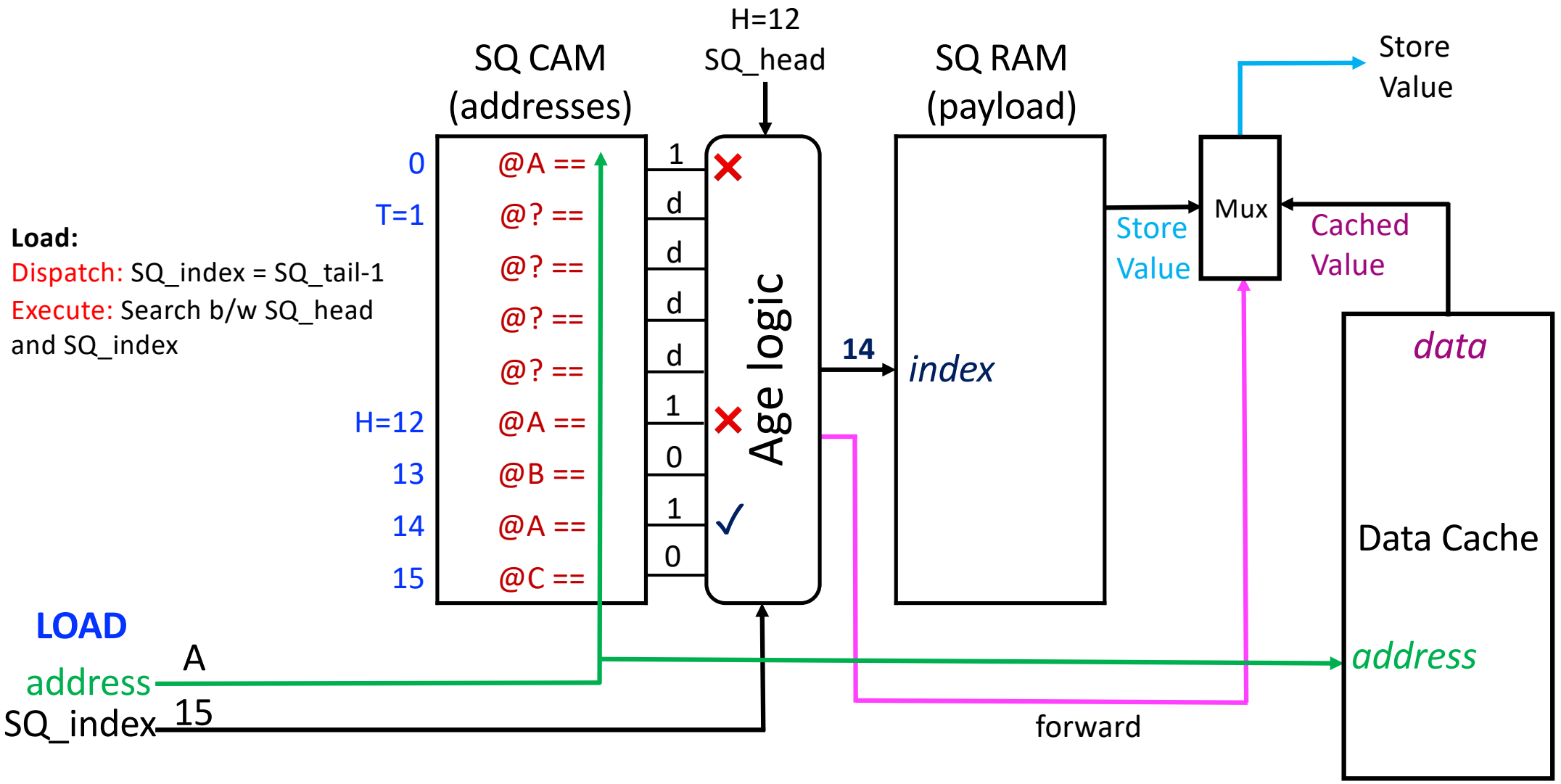
1. LQ_index = LQ_tail: The load's entry in the LQ
   When the load executes later, it uses LQ_index to place the address in LQ. The address is needed to detect mispredicted loads
2. SQ_index = SQ_tail-1: Index of immediately preceding store, in program order
   When the laod executes later, it searches the SQ for a dependence on a prior store. It only considers stores between SQ_head and SQ_index: these are the stores before the load, in program order

Store Execution Datapath

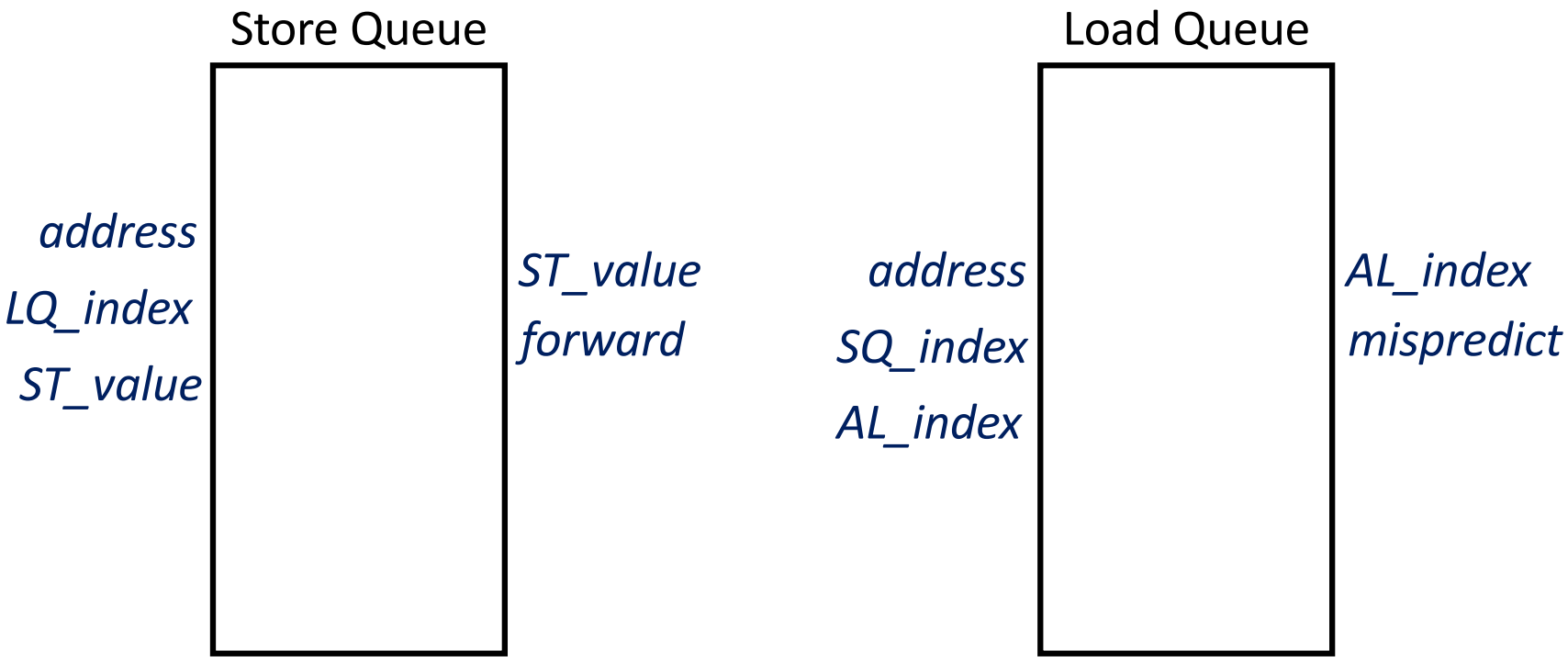
CAM = Content Addressable Memory
RAM = Random Access Memory
(index based)



Load Execution Datapath



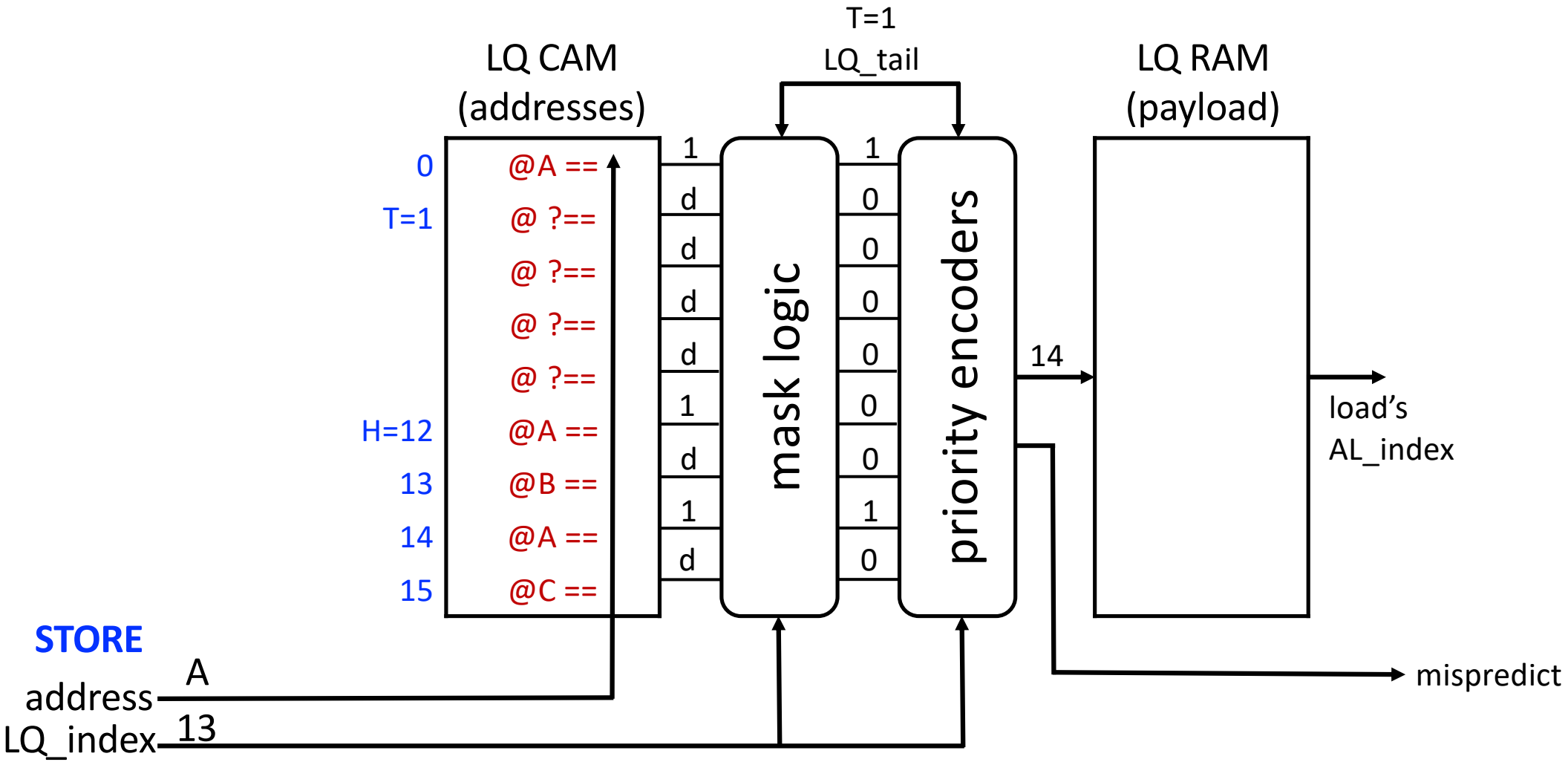
Backup: LQ/SQ Interfaces



Inside each queue exists

1. One content addressable memory (CAM) structure for searching matching addresses
2. One RAM structure for values (SQ) and AL_entry of mispredicted load (LQ)

Backup: Store Execution Datapath



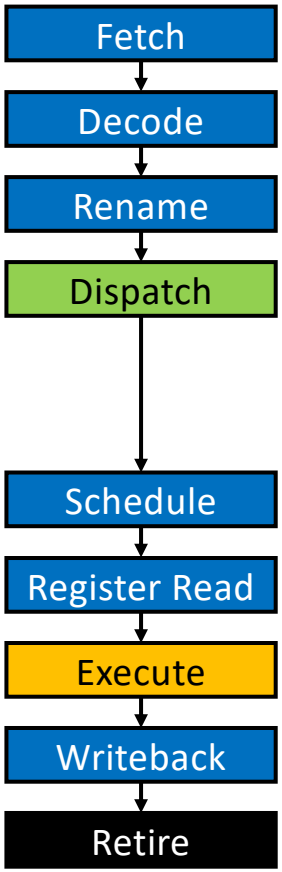
Store Execution: Execute when Ready

- **Dispatch Stage (in-order)**
  - Store is allocated the SQ entry at tail of SQ (its SQ_index)
  - Store also notes the current LQ tail, so it knows which loads are after it in program order (its LQ_index)
- **Execute Stage (out-of-order)**
  - AGEN: Generate store's address
  - Write SQ: Write store's address and value into its SQ entry (at its SQ_index)
  - Read LQ: Use store's address and LQ_index to search LQ for mispredicted loads: loads after the store in program order (between its LQ_index and LQ_tail), with the same address as the store, which already executed
- **Retire Stage (in-order)**
  - When a store reaches the head of the Active List, Active List signals SQ to commit its oldest store (SQ head) to the D$

Load Execution: Execute when Ready

- **Dispatch Stage (in-order)**
  - Load is allocated the LQ entry at tail of LQ (its LQ_index)
  - Load also notes the current SQ tail, so it knows which stores are before it in program order (its SQ_index)
- **Execute Stage (out-of-order)**
  - AGEN: Generate load's address
  - Write LQ: Write load's address into its LQ entry (at its LQ_index)
  - Read SQ and D$: Use load's address and SQ_index to search SQ for best estimate (some stores' addresses still unknown) of producer store: nearest store before the load in program order (between SQ_head and its SQ_index), with the same address as the load. If SQ hit, use store value, else use D$ value.
- **Retire Stage (in-order)**
  - When a load reaches the head of the Active List:
    - Signal LQ to remove its oldest load (LQ head)
    - If load's misprediction bit is set in Active List, initiate misprediction recovery (approach #1 or #2). Fetch unit is redirected to PC of load so that the load re-executes, this time correctly since it is oldest instruction in pipeline (all prior stores have committed to D$)

Speculative Load/Store Execution



Same content as previous two slides, but different presentation

(1) Allocate the load or store at the tail of the Active List
(2) Place the load or store in Issue Queue (IQ)
(3) Place the load or store in Load Queue (LQ) or Store Queue (SQ), respectively, at the tail
A load gets LQ tail (LQ_index: where it resides in LQ) and SQ tail minus 1 (SQ_index: index of immediately preceding store in SQ)
A store gets SQ tail (SQ_index: where it resides in SQ) and LQ tail (LQ_index: index of to-be-dispatched, immediately succeeding load in LQ)

(1) Calculate address (AGEN). (Remember that here is a separate address generation unit.)
(2) Load: Use address to access D$ and search SQ for matching addresses (D$ and SQ accessed in parallel). Based on the result of SQ search, a load gets value from SQ (closest matching store) or D$ (no matching store in SQ). Also record load's address in LQ.
Store: Use address to search LQ for matching addresses; if there is a future load that already executed, and its address matches, mark that load in the Active List as "mispredicted". Also record store's address and value in the SQ.

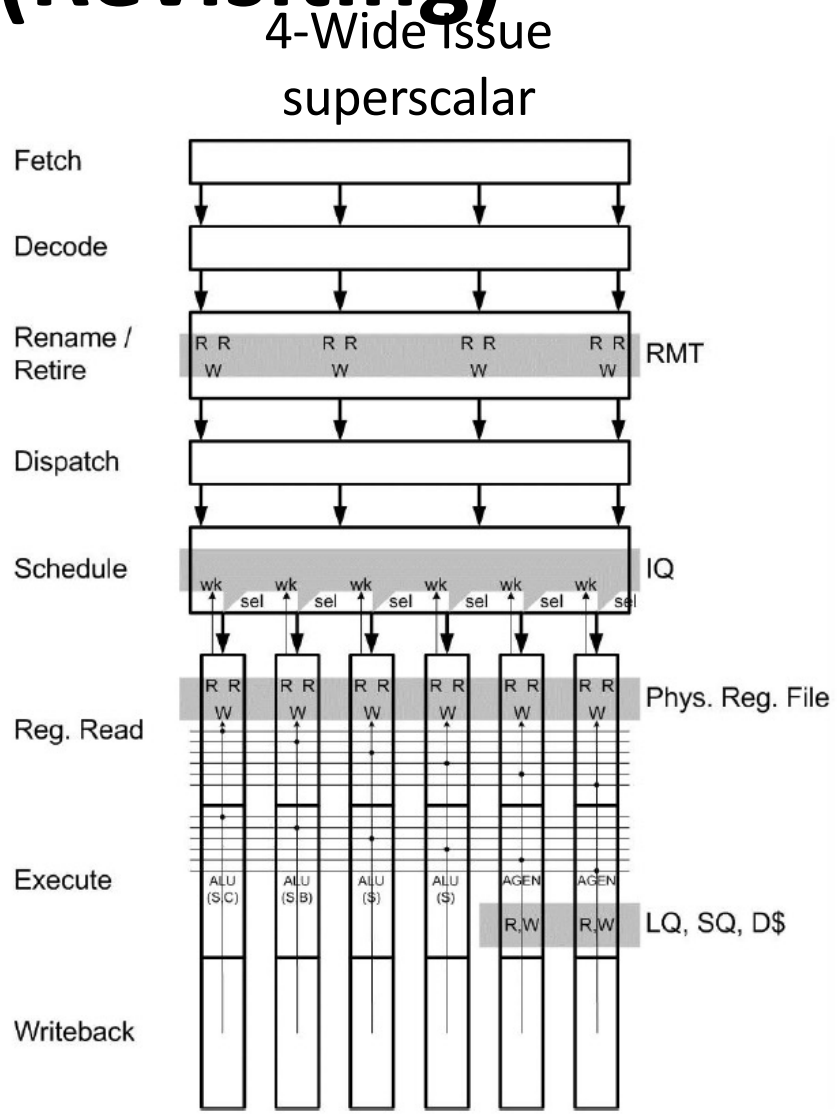
Load: If marked as "mispredicted", initiate recovery actions (e.g., use "Approach #1" or "Approach #2"); otherwise commit load the same way as other register-producing instructions.
(Note: Re-executing a mispredicted load after recovery will succeed because all prior stores have committed.)
Store: Signal the store at the head of the SQ to write its value to the D$ at its address.
(Note: The store at the head of the Active List is the same as the store at the head of the SQ.)

After load or store successfully commits, pop from LQ or SQ, respectively.

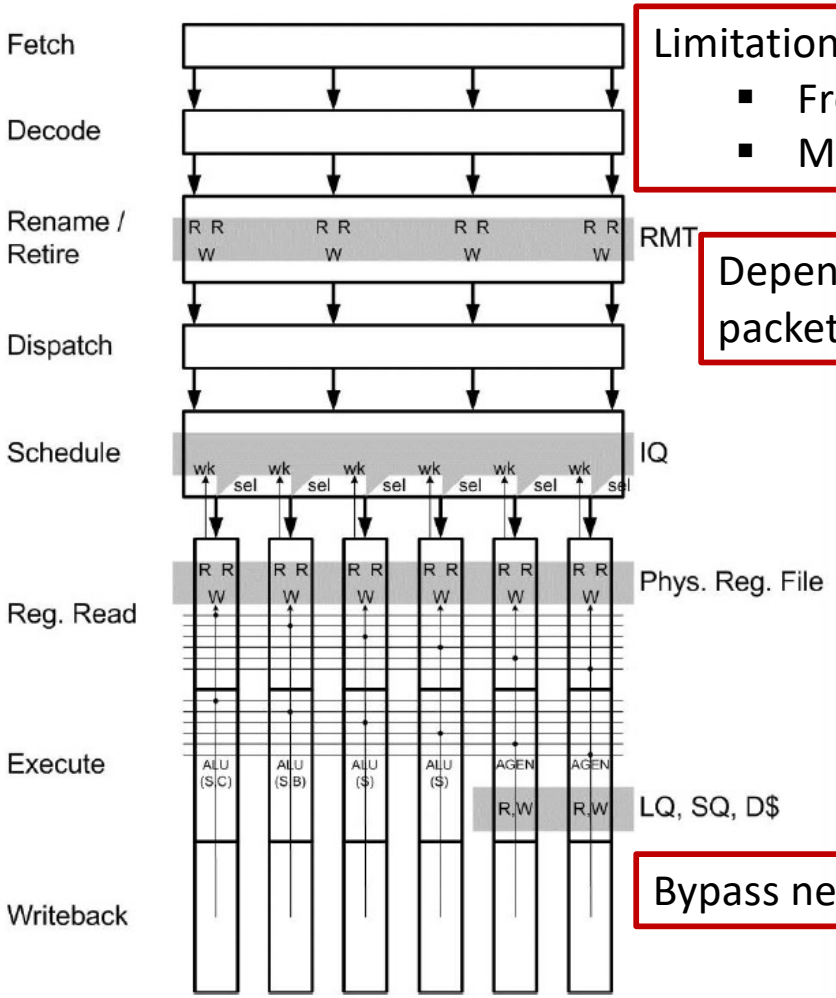
Superscalar Complexity (Revisiting)

Superscalar and complexity

- Fetch, rename, dispatch, issue, and commit multiple instructions per cycle
- Use dynamic scheduling, renaming, and hardware speculation
- Goal: IPC>1 (ideal = issue width)
- Complexity increases with (issue) width
- Beyond 6-8 issue, the industry moved to multicores (?)
  - Complexity of very wide issue superscalar is not worth the increase in IPC. Better to exploit thread-level parallelism (TLP)
  - *So, after many decades of sustained performance increases, multicores(2005→) shifted the burden of perf. on software (how is that going?)*



Superscalar Complexity (Revisiting)



Limitations & complexity of fetching

- Frequent taken branches
- Multiple branch prediction

Dependences within the rename packet/bundle

Large # ports (RMT)

- Ports scale linearly with width of superscalar
- Sizes scale super-linearly with width to exploit ILP
- IQ, LQ, and SQ are CAMs (associative structures)
- Specialized logic (wake-up/select)

Bypass network complexity

OOO Load Execution Summary

- **Important note: Store is complete when it is written in L1 D$**
- **In-order execution of loads and stores**
  - Issue loads and stores in program order from a monolithic LSQ
- **Load Bypassing**
  - Bypass older stores in the store queue if there is no aliasing
    - Need to wait until all older stores have their addresses computed
- **Load Bypassing + Forwarding**
  - Loads can be satisfied from SQ if there is an address match
    - Need to wait until all older stores have their addresses computed
- **Execute when Ready**
  - Loads execute when their addresses are ready
  - Loads speculate that older stores are to other addresses
    - Need the ability to undo bad loads (mis-speculations)

Speculative Load Handling

- A load is speculative if there are prior unknown store addresses
- **Four bleeding edge issues**
  - Memory Dependence Predictor (MDP)
  - Store-load synchronization strategy
  - Load misprediction recovery strategy
  - Impact of store execution (split stores vs. no split stores)

Memory Dependence Predictor

- Role of MDP is to predict whether to synchronize (stall) or speculatively execute a speculative load
- **Two static MDPs**
  - Always synchronize (most conservative)
  - Always speculatively execute (most aggressive)
- **Many possible dynamic MDPs**
  - Synchronize or speculatively execute, depending on the likelihood of a dependence gauged by MDP (most intelligent)
  - Simplest: classify loads as frequent or infrequent offenders
  - Most sophisticated (e.g., Store Sets): learn store-load relationships and account for which stores are currently in the pipeline

Store-Load Sync. Strategies

- **Synchronize through Issue Queue**
  - Store Sets: Store in a load's "store set" wakes up the load when the store issues. (Multiple stores in the "store set" issue in program order, with the youngest one issuing last and waking up the load.)
  - Pros: Timely wakeup, without polling
  - Consts: Issue queue complexity increases
- **Replay from Load Queue**
  - Load issues like usual, generates an address, then is marked as "unexecuted" in LQ. LQ replays the load when all prior stores have committed (for example) or similar criterion
  - Pros: IQ unchanged; Cons: Too much delay or too much polling

Load Misprediction Recovery Strategies

- **Squash**
  - Pros: Machinery already exists
  - Cons: Highest penalty
- **Selective re-execution of load-dependent instructions**
  - Replay from IQ (requires holding entries until confirmed) or from secondary replay buffer
  - Pros: Lowest penalty
  - Cons: Very complex hardware

Impact of Store Execution

- **What are split stores?**
  - Split a store instruction into two micro-ops: store address and store value
  - The two micro-ops issue separately from IQ and “meet up” in SQ
  - Helps in the case where store value is significantly delayed compared to store address
  - Getting the store address into the SQ asap results in more informed loads
- **Pros:** More informed loads, resulting in less unnecessary synchronization and fewer mispredictions
- **Cons:** complex implementation. Stores double their IQ width consumption.

Intertwined Factors

- **Choices for one impacts the others**
  - With an excellent MDP, can probably do with simplest synchronization (replay from LQ) and recovery strategies (squash)
  - With split stores, perhaps a sophisticated MDP is an overkill
  - With selective re-execution, perhaps always speculatively executing makes sense (static aggressive MDP)