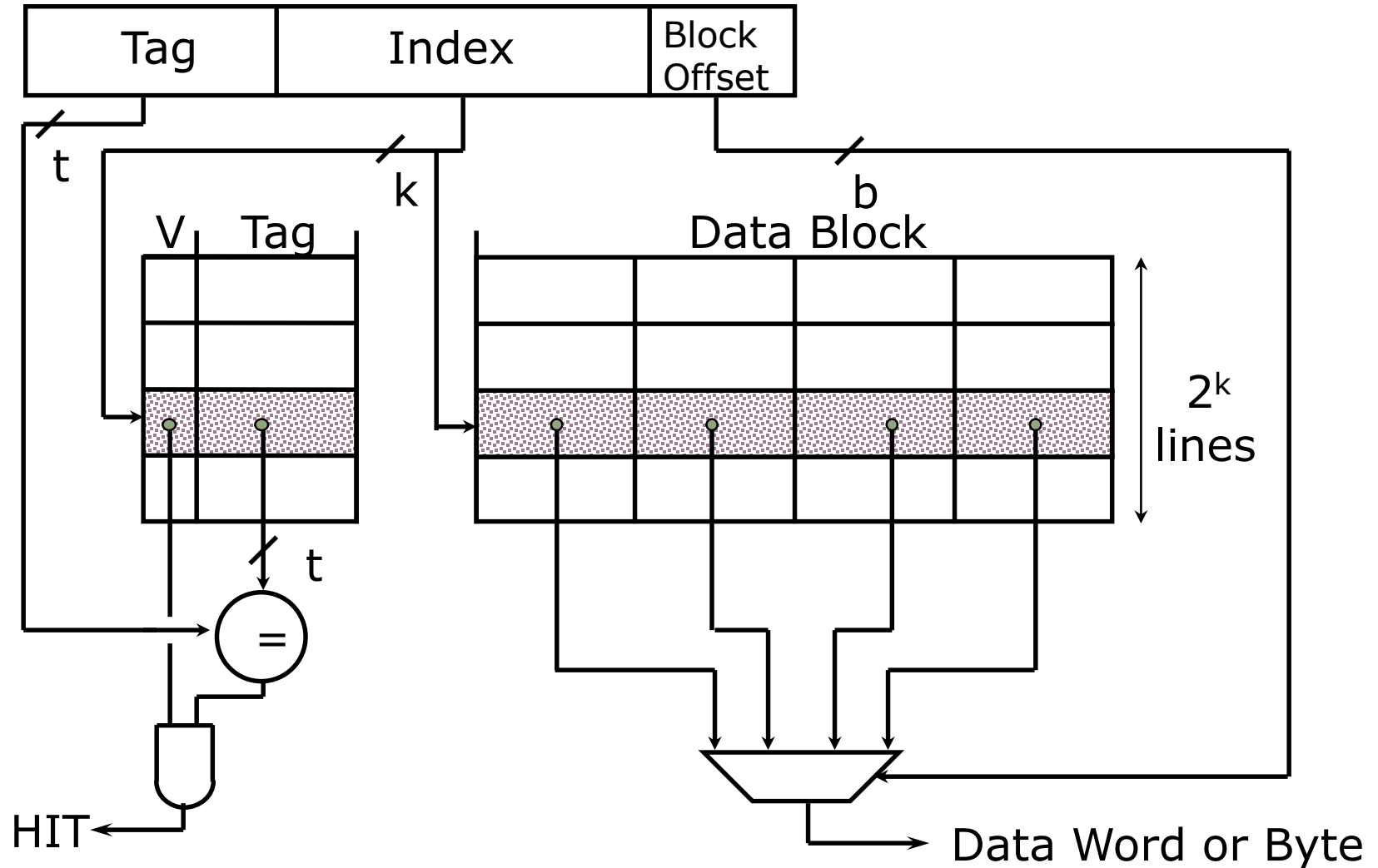


# Advanced Memory Operations

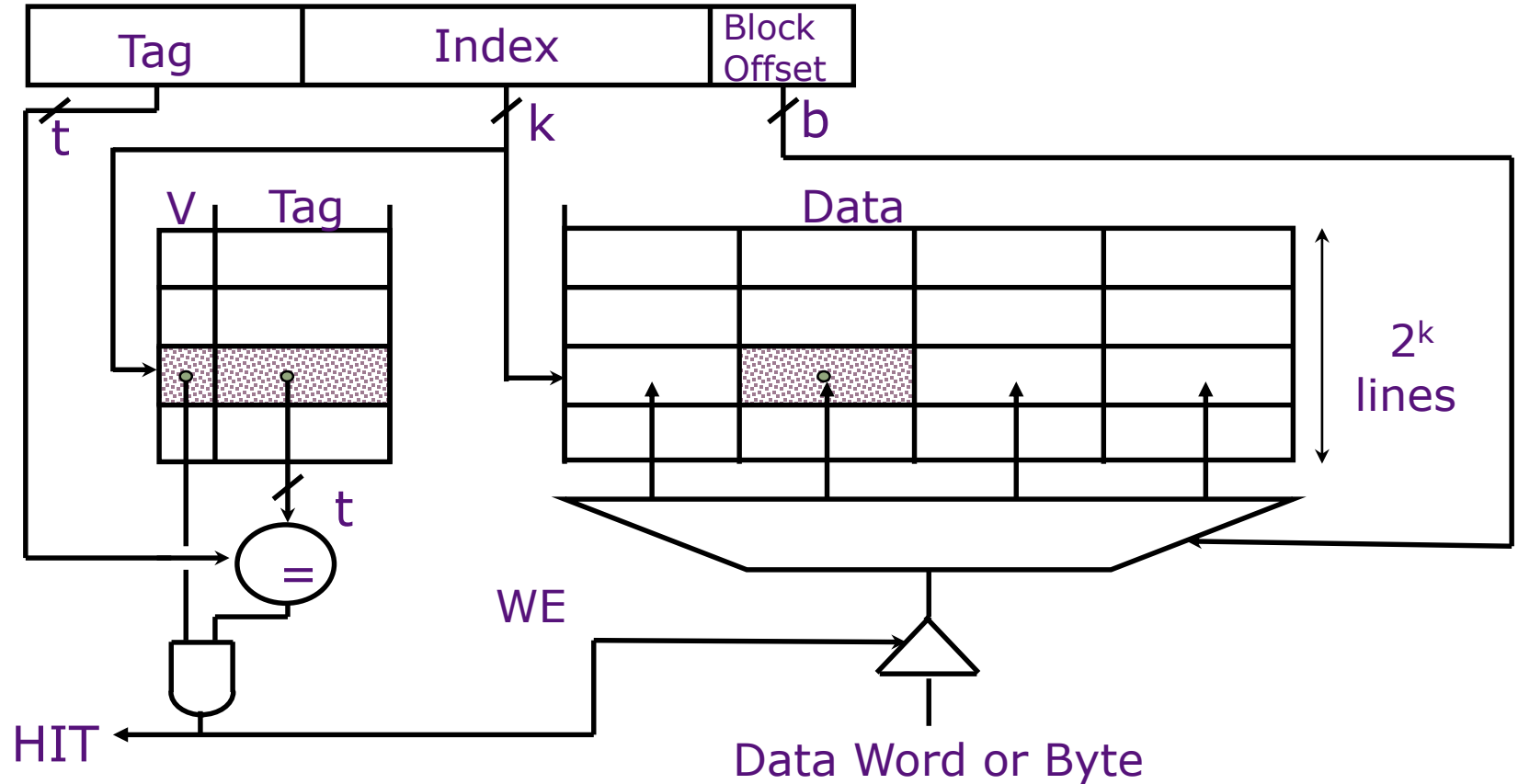
*Daniel Sanchez*

Computer Science and Artificial Intelligence Laboratory  
M.I.T.

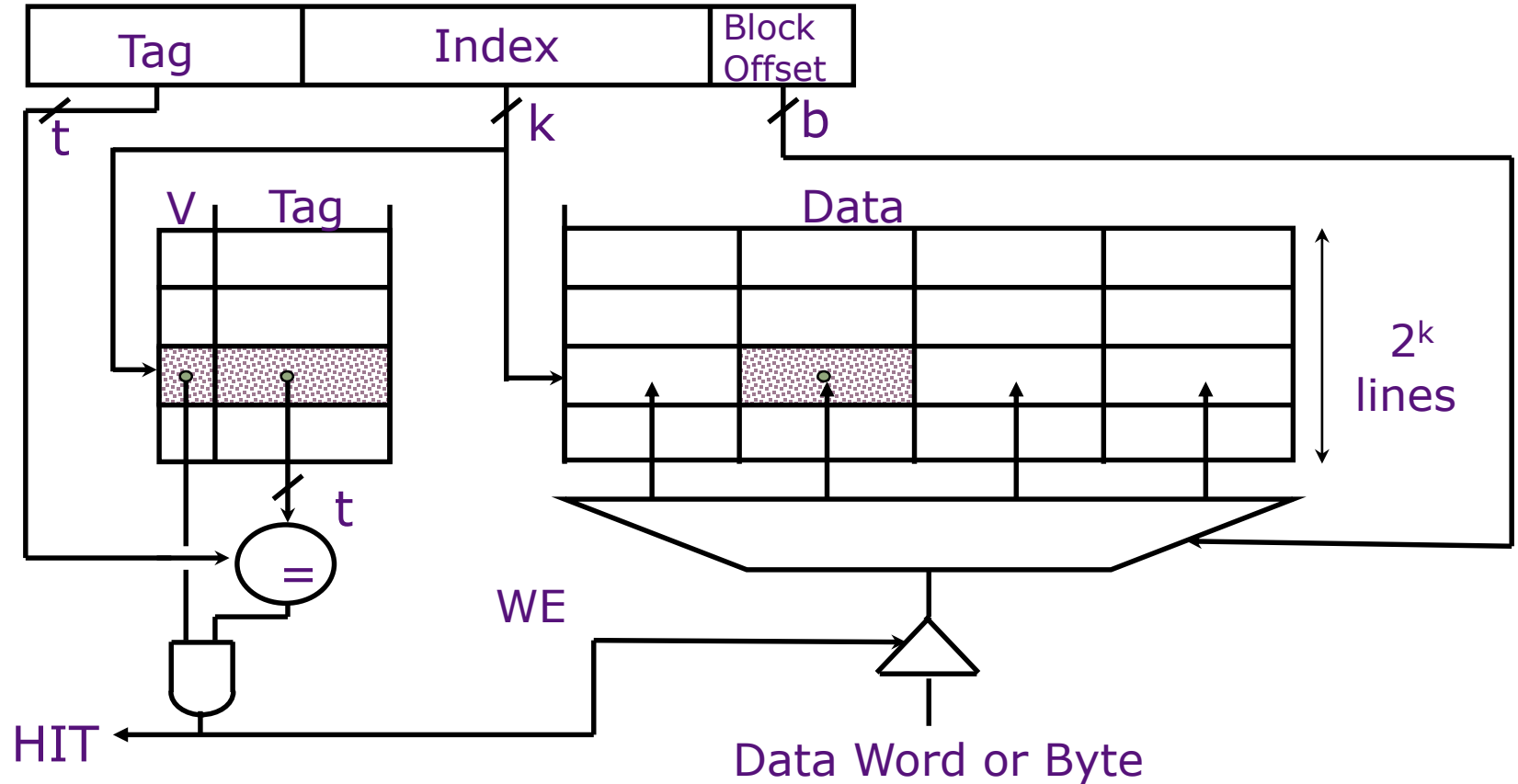
# Reminder: Direct-Mapped Cache



# Write Performance

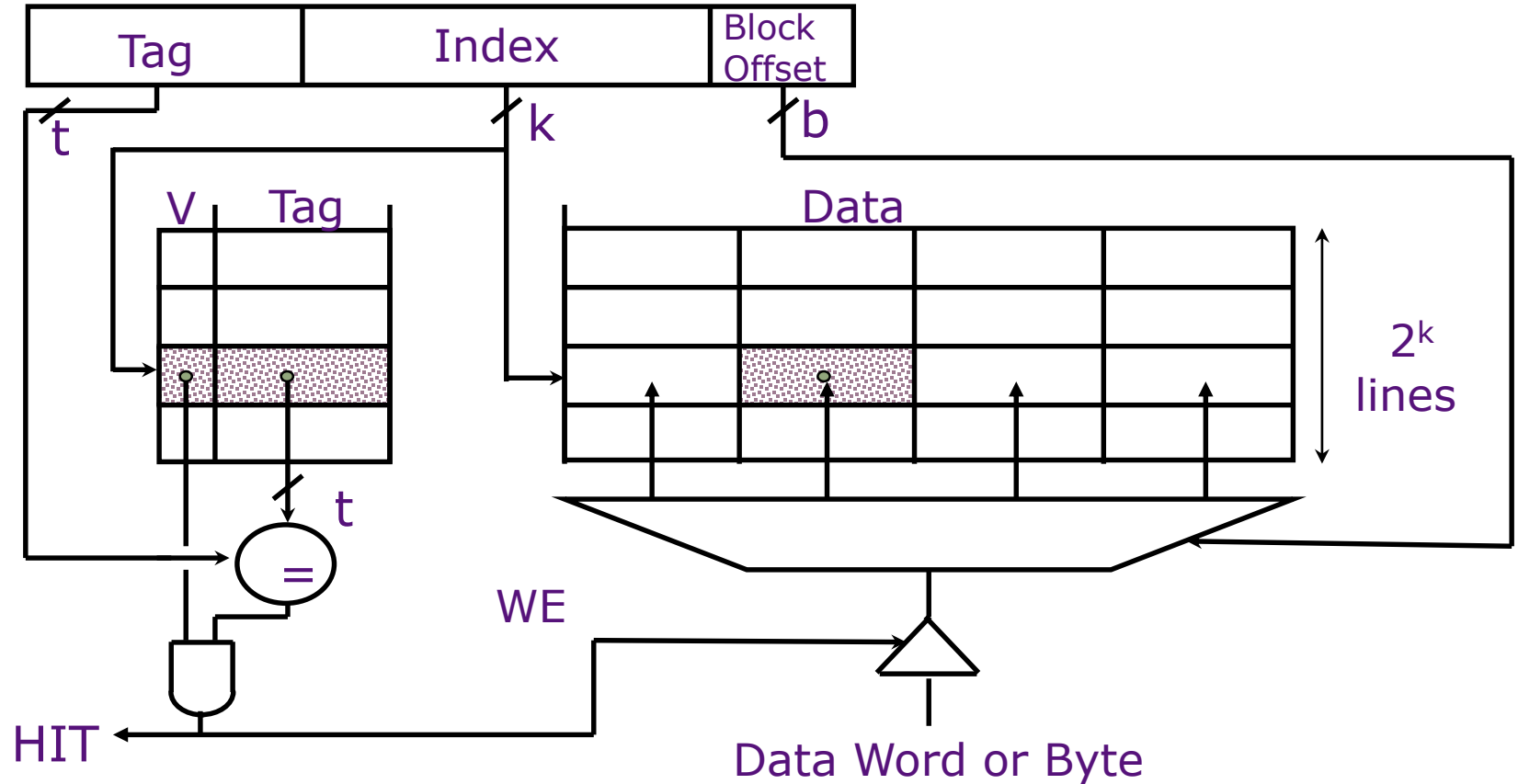


# Write Performance



*How does write timing compare to read timing?*

# Write Performance



*How does write timing compare to read timing?*

**Completely serial!**

# Reducing Write Hit Time

---

Problem: Writes take two cycles in memory stage, one cycle for tag check plus one cycle for data write if hit

# Reducing Write Hit Time

---

Problem: Writes take two cycles in memory stage, one cycle for tag check plus one cycle for data write if hit

View: Treat as data dependence on micro-architectural value 'hit/miss'

# Reducing Write Hit Time

---

Problem: Writes take two cycles in memory stage, one cycle for tag check plus one cycle for data write if hit

View: Treat as data dependence on micro-architectural value 'hit/miss'

Solutions:



# Reducing Write Hit Time

---

Problem: Writes take two cycles in memory stage, one cycle for tag check plus one cycle for data write if hit

View: Treat as data dependence on micro-architectural value 'hit/miss'

Solutions:

- Wait – delivering data as fast as possible:
  - Fully associative (CAM Tag) caches: Word line only enabled if hit

# Reducing Write Hit Time

---

Problem: Writes take two cycles in memory stage, one cycle for tag check plus one cycle for data write if hit

View: Treat as data dependence on micro-architectural value 'hit/miss'

Solutions:

- Wait – delivering data as fast as possible:
  - Fully associative (CAM Tag) caches: Word line only enabled if hit
- Speculate predicting hit with greedy data update:
  - Design data RAM that can perform read and write in one cycle
  - Restore old value after tag miss (abort)

# Reducing Write Hit Time

---

Problem: Writes take two cycles in memory stage, one cycle for tag check plus one cycle for data write if hit

View: Treat as data dependence on micro-architectural value 'hit/miss'

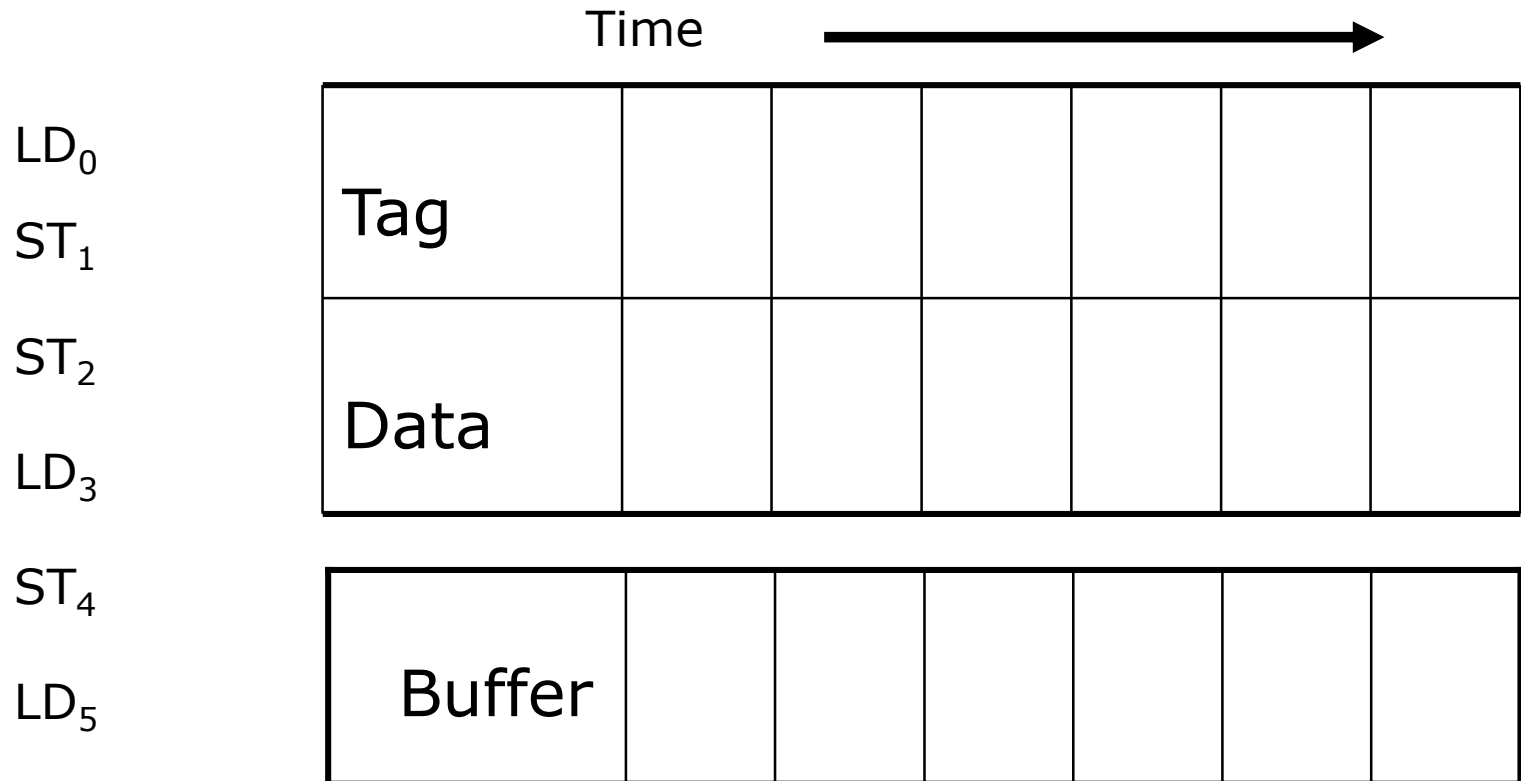
Solutions:

- Wait – delivering data as fast as possible:
  - Fully associative (CAM Tag) caches: Word line only enabled if hit
- Speculate predicting hit with greedy data update:
  - Design data RAM that can perform read and write in one cycle
  - Restore old value after tag miss (abort)
- Speculate predicting miss with lazy data update:
  - Hold write data for store in single buffer ahead of cache
  - Write cache data during next idle data access cycle (commit)

# Pipelined/Delayed Write Timing

---

Problem: Need to commit lazily saved write data

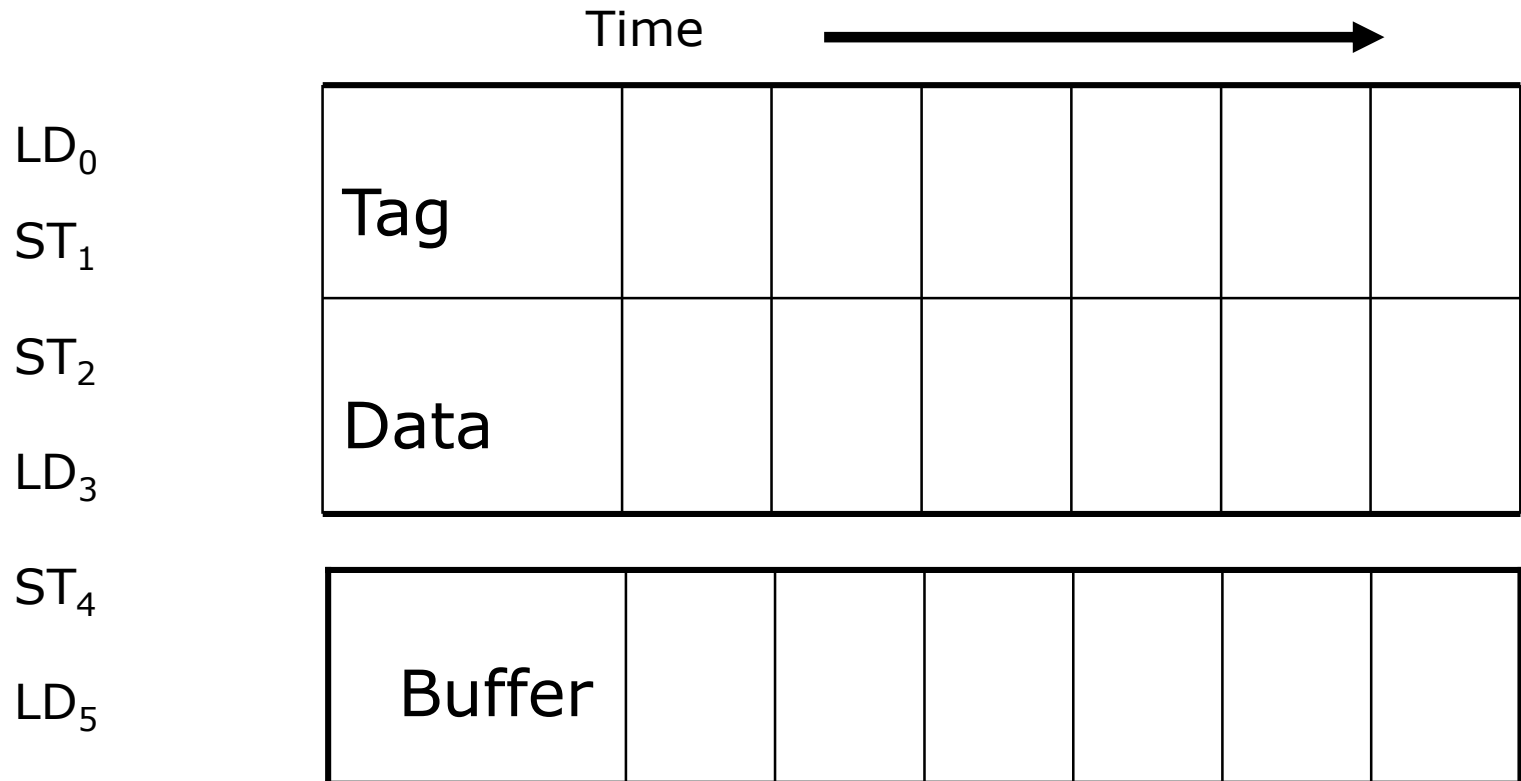


# Pipelined/Delayed Write Timing

---

Problem: Need to commit lazily saved write data

Solution: Write data during idle data cycle of next store's tag check

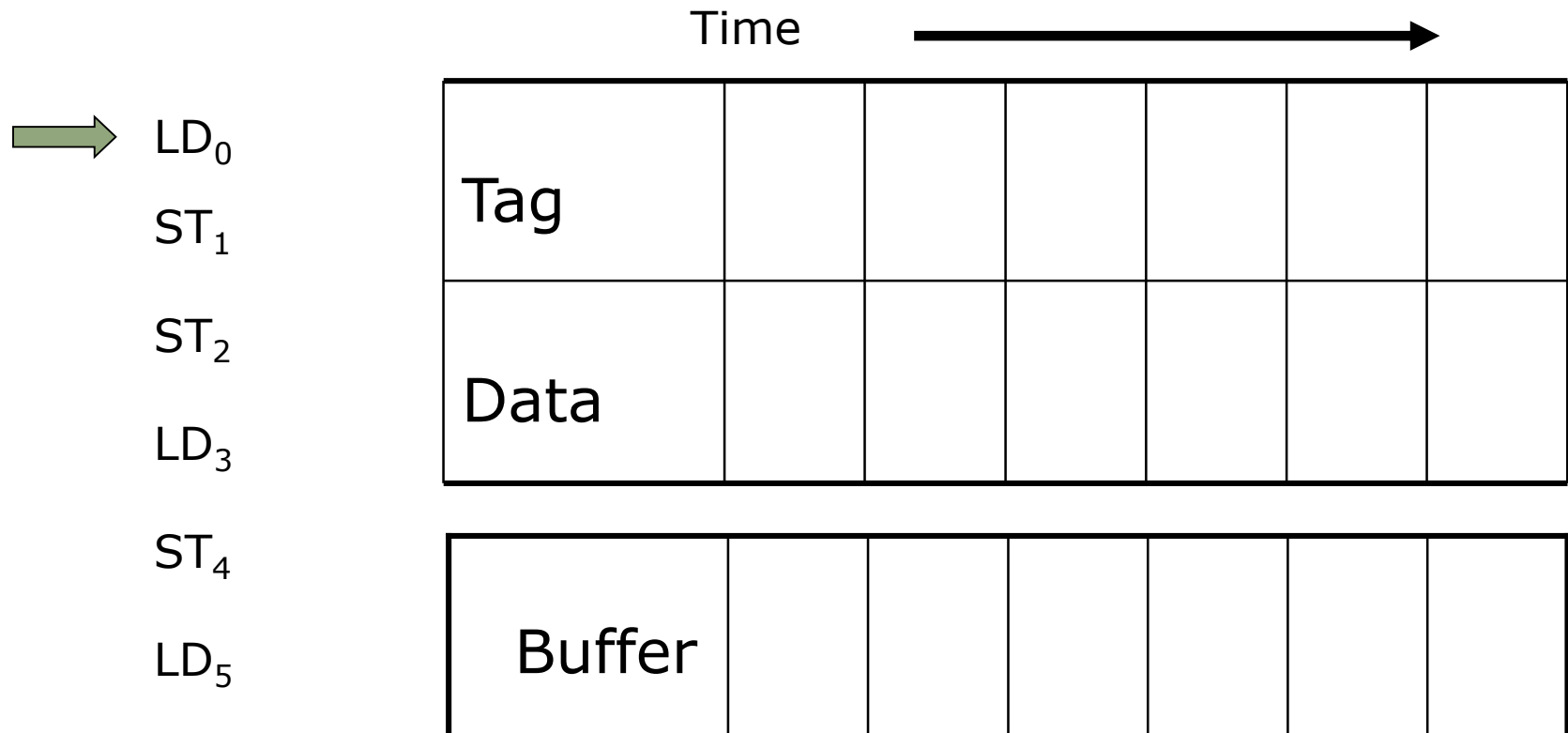


# Pipelined/Delayed Write Timing

---

Problem: Need to commit lazily saved write data

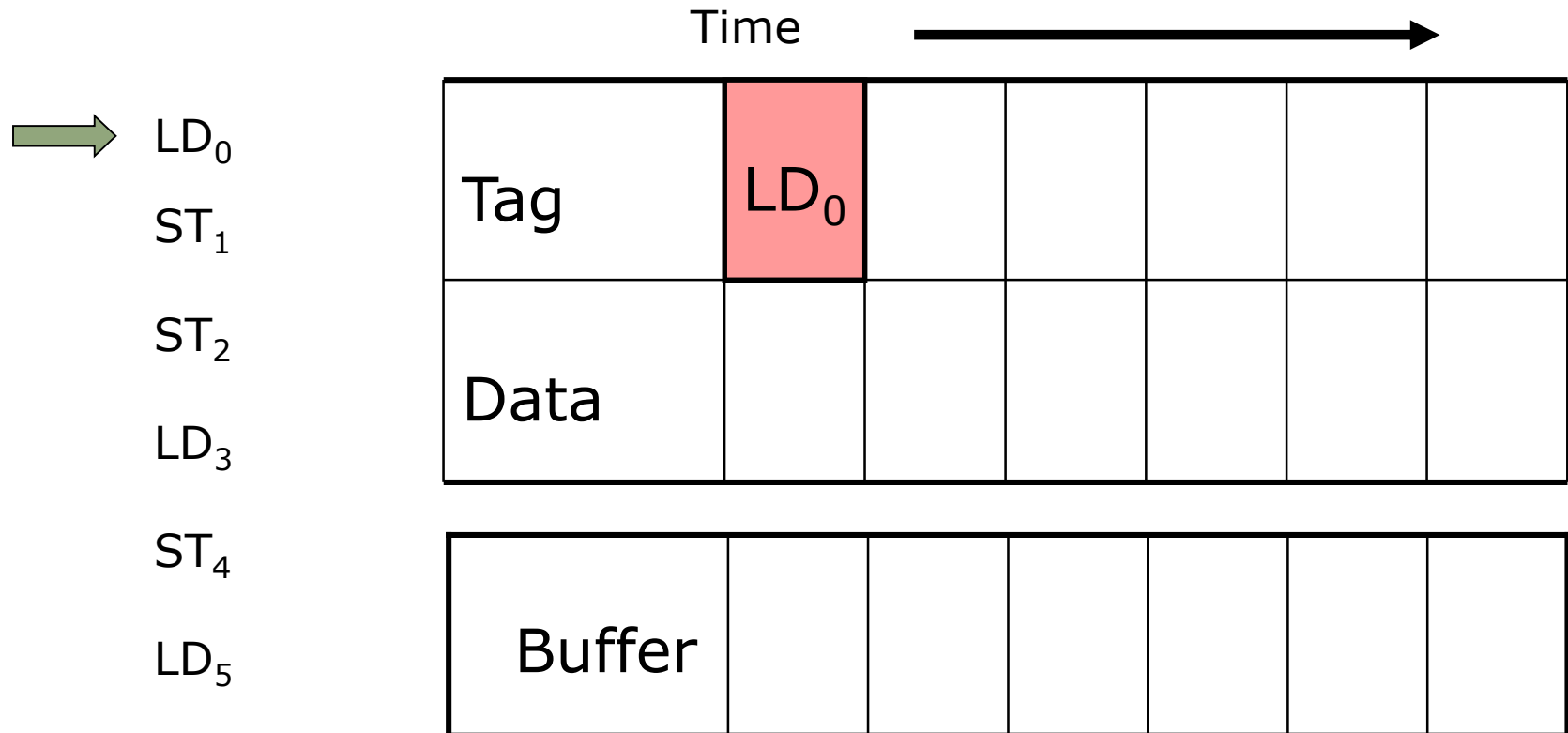
Solution: Write data during idle data cycle of next store's tag check



# Pipelined/Delayed Write Timing

Problem: Need to commit lazily saved write data

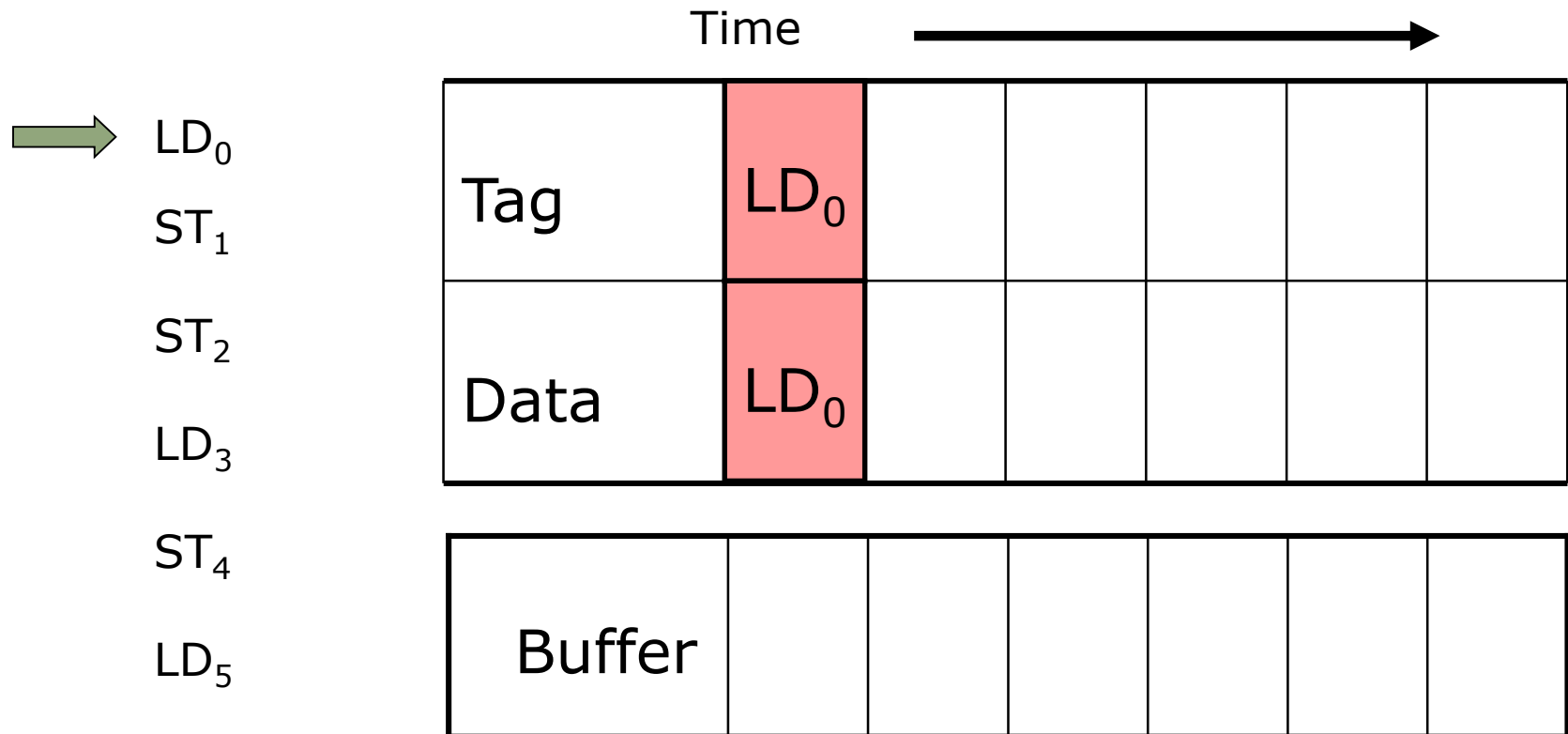
Solution: Write data during idle data cycle of next store's tag check



# Pipelined/Delayed Write Timing

Problem: Need to commit lazily saved write data

Solution: Write data during idle data cycle of next store's tag check

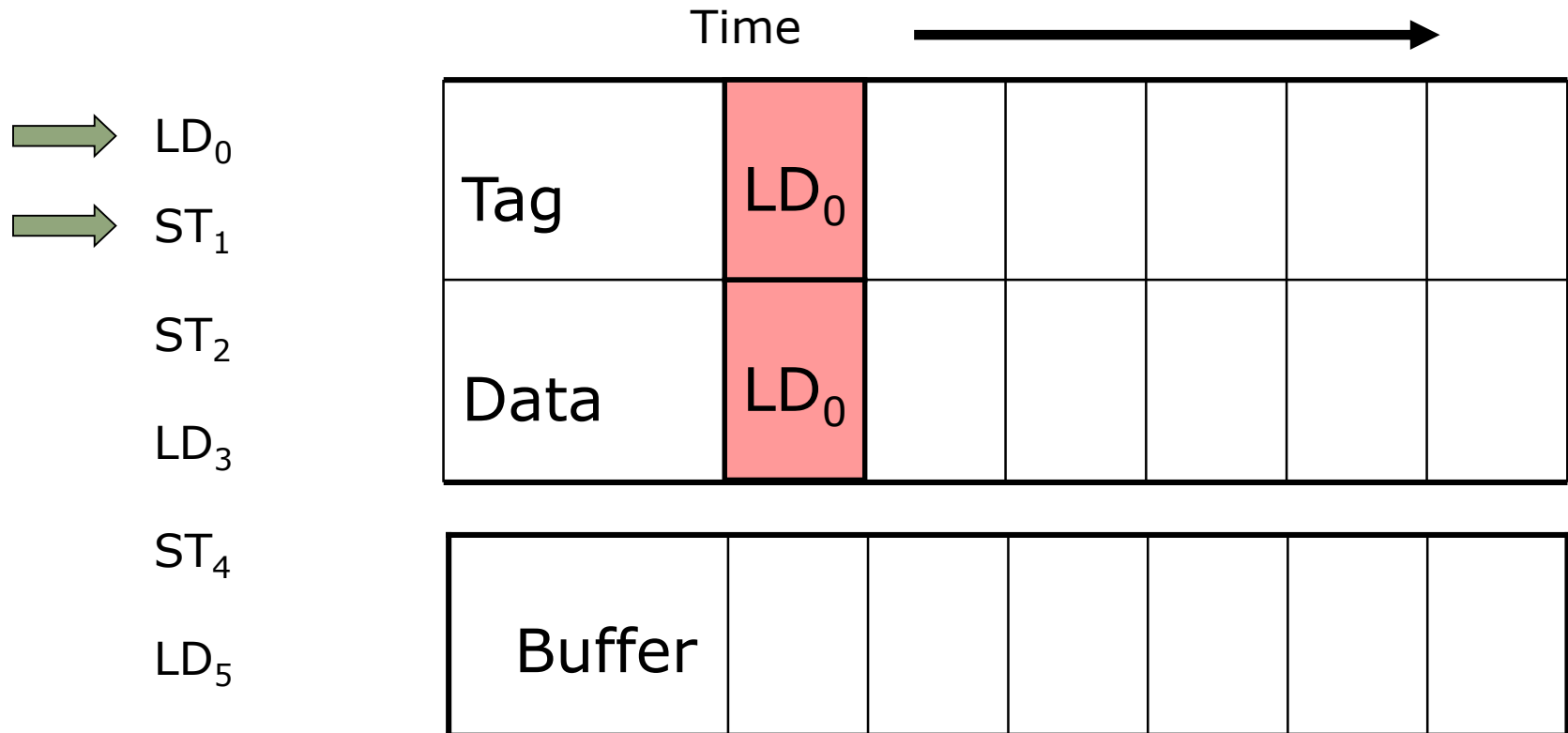




# Pipelined/Delayed Write Timing

Problem: Need to commit lazily saved write data

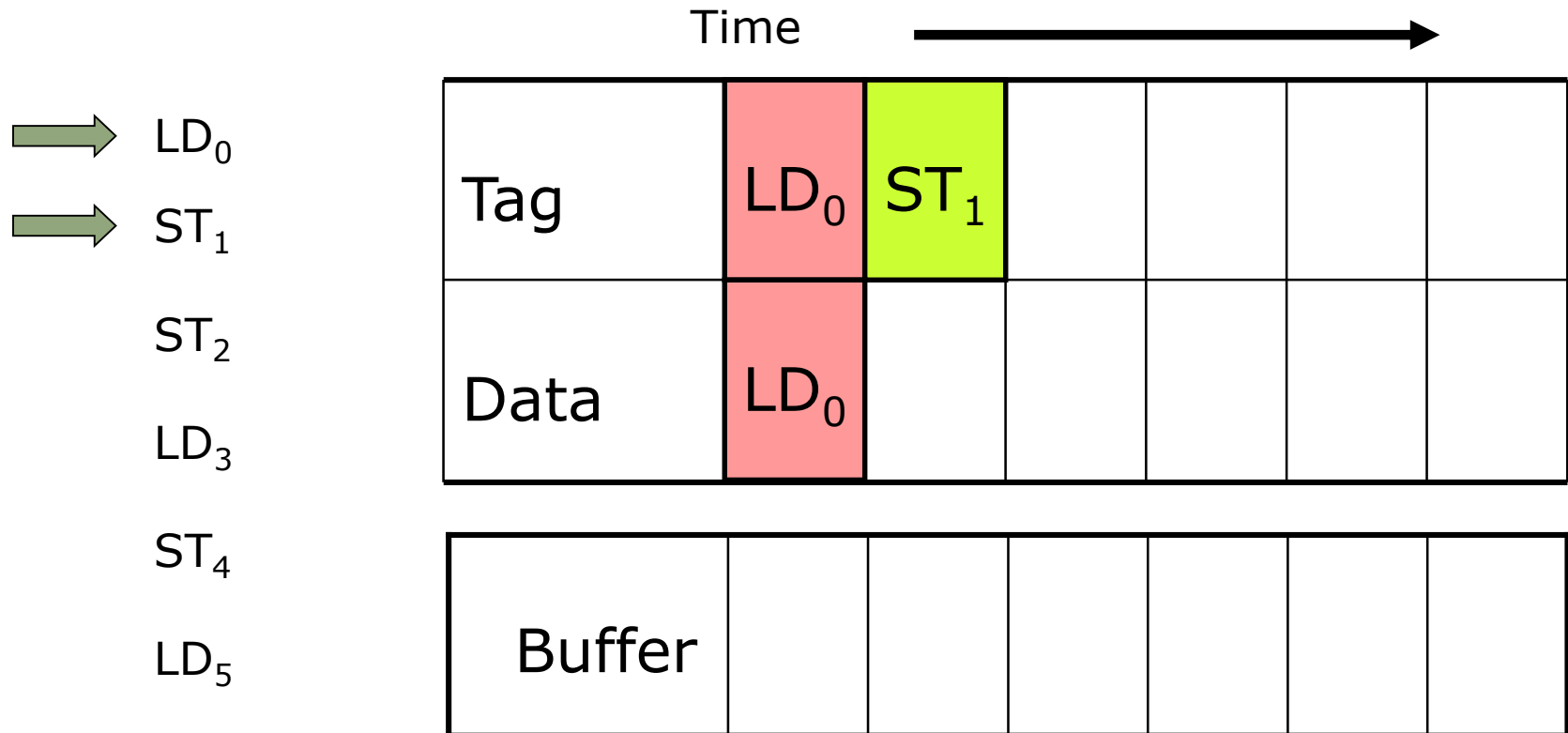
Solution: Write data during idle data cycle of next store's tag check



# Pipelined/Delayed Write Timing

Problem: Need to commit lazily saved write data

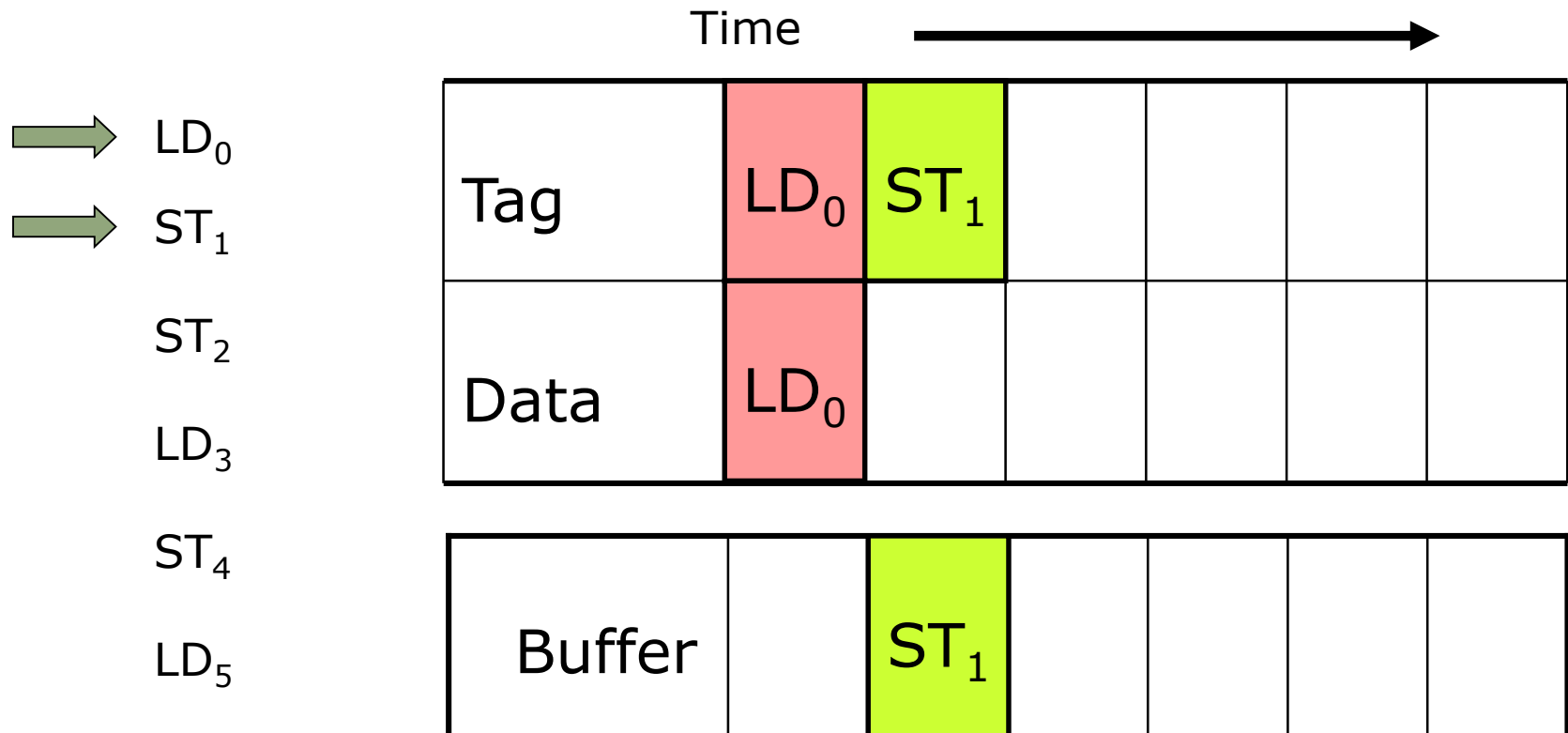
Solution: Write data during idle data cycle of next store's tag check



# Pipelined/Delayed Write Timing

Problem: Need to commit lazily saved write data

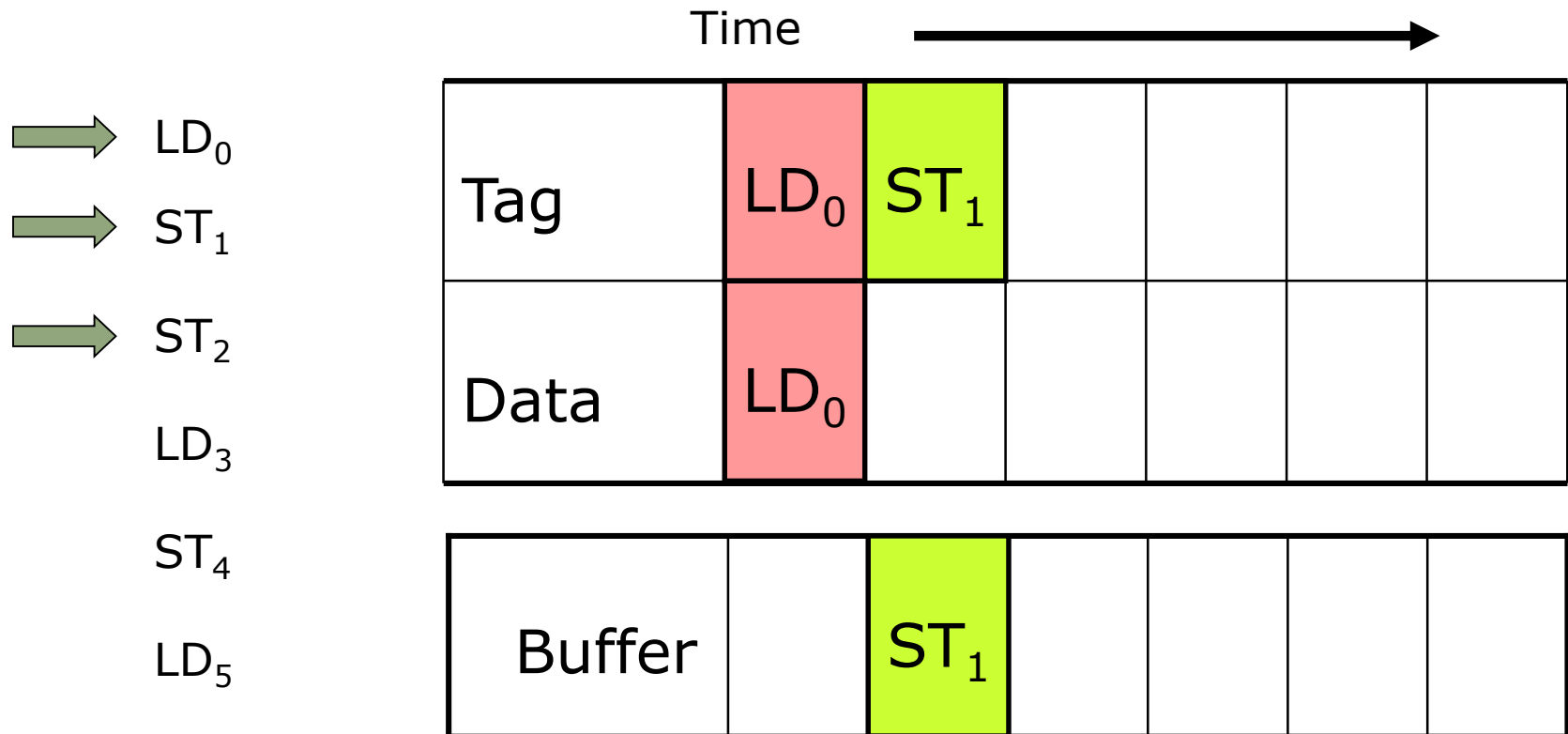
Solution: Write data during idle data cycle of next store's tag check



# Pipelined/Delayed Write Timing

Problem: Need to commit lazily saved write data

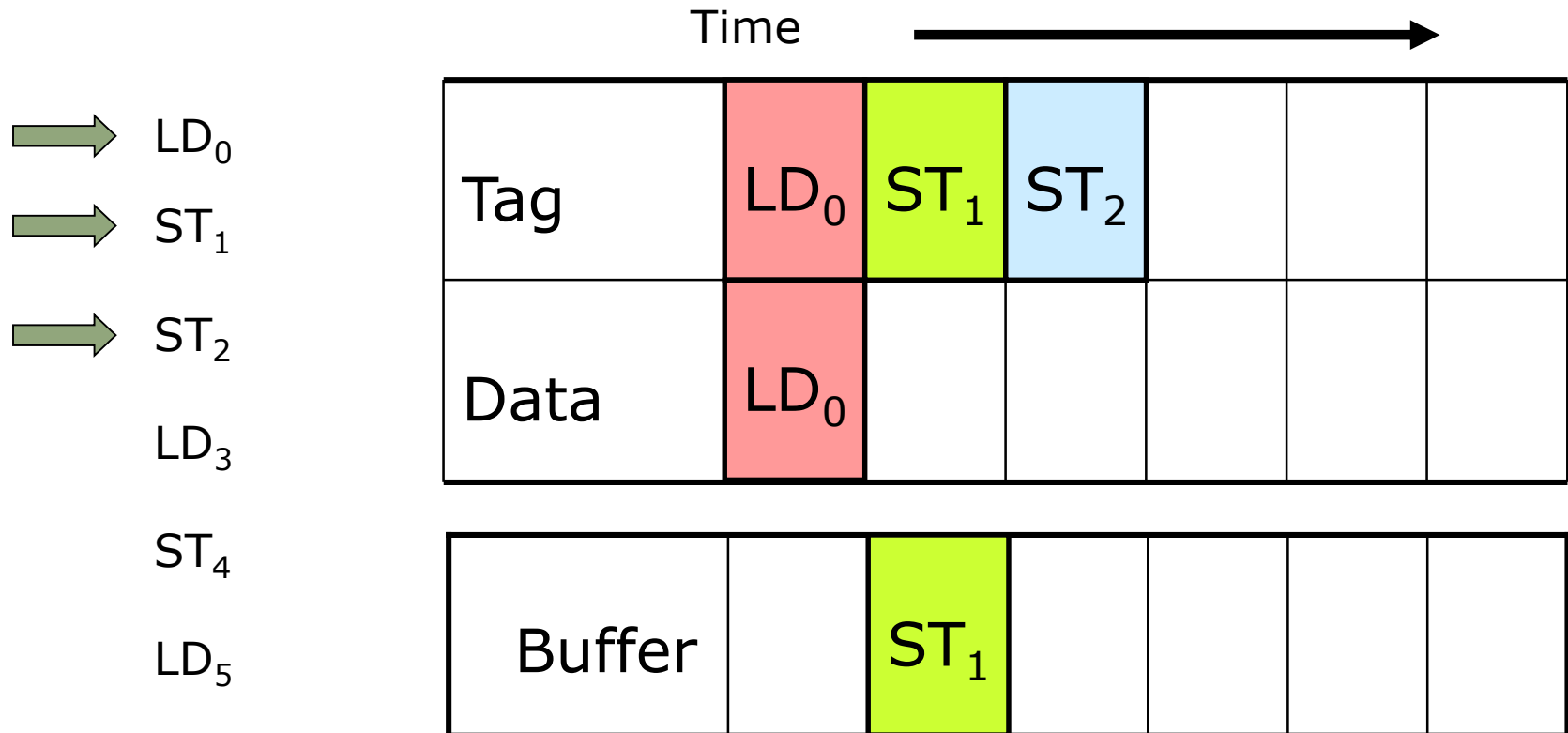
Solution: Write data during idle data cycle of next store's tag check



# Pipelined/Delayed Write Timing

Problem: Need to commit lazily saved write data

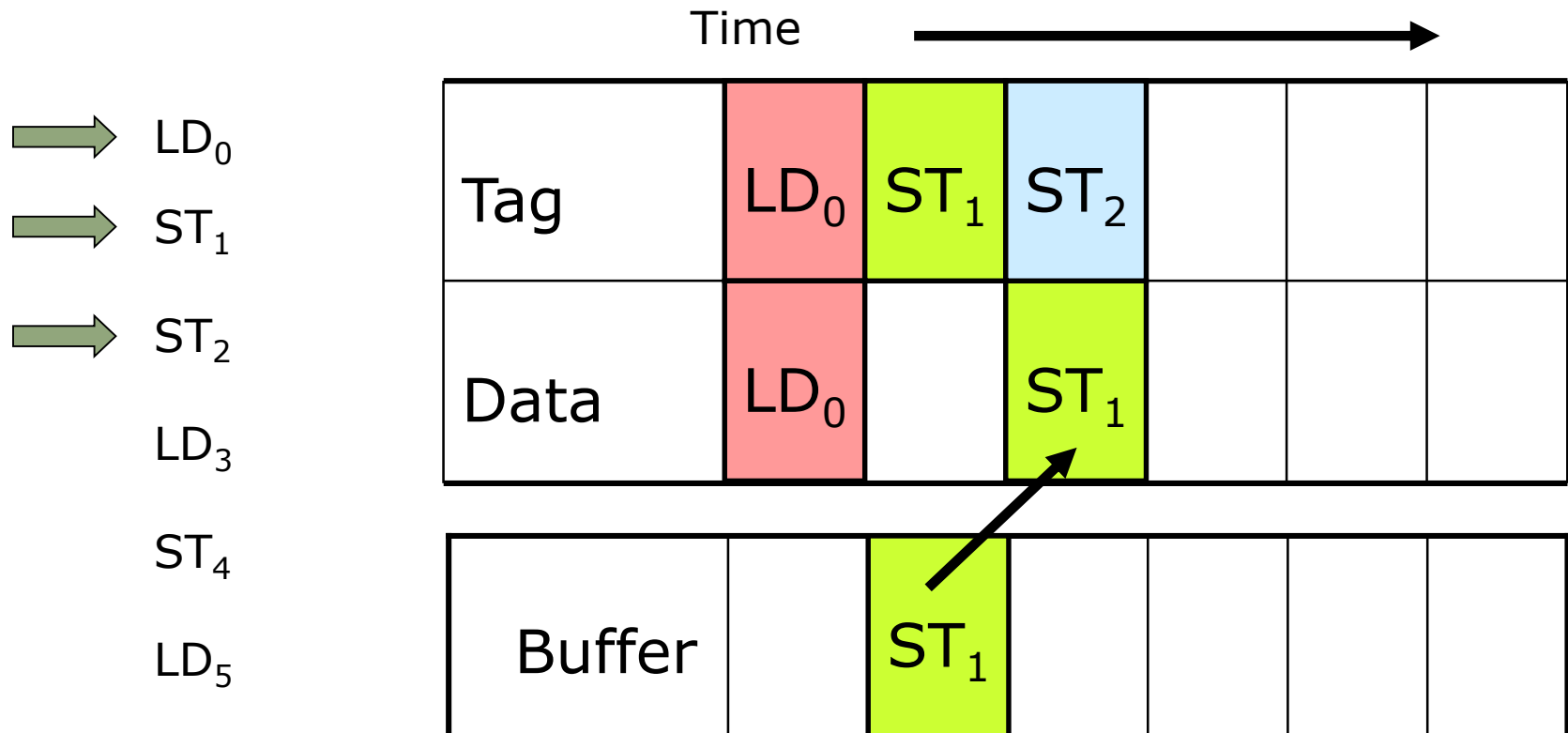
Solution: Write data during idle data cycle of next store's tag check



# Pipelined/Delayed Write Timing

Problem: Need to commit lazily saved write data

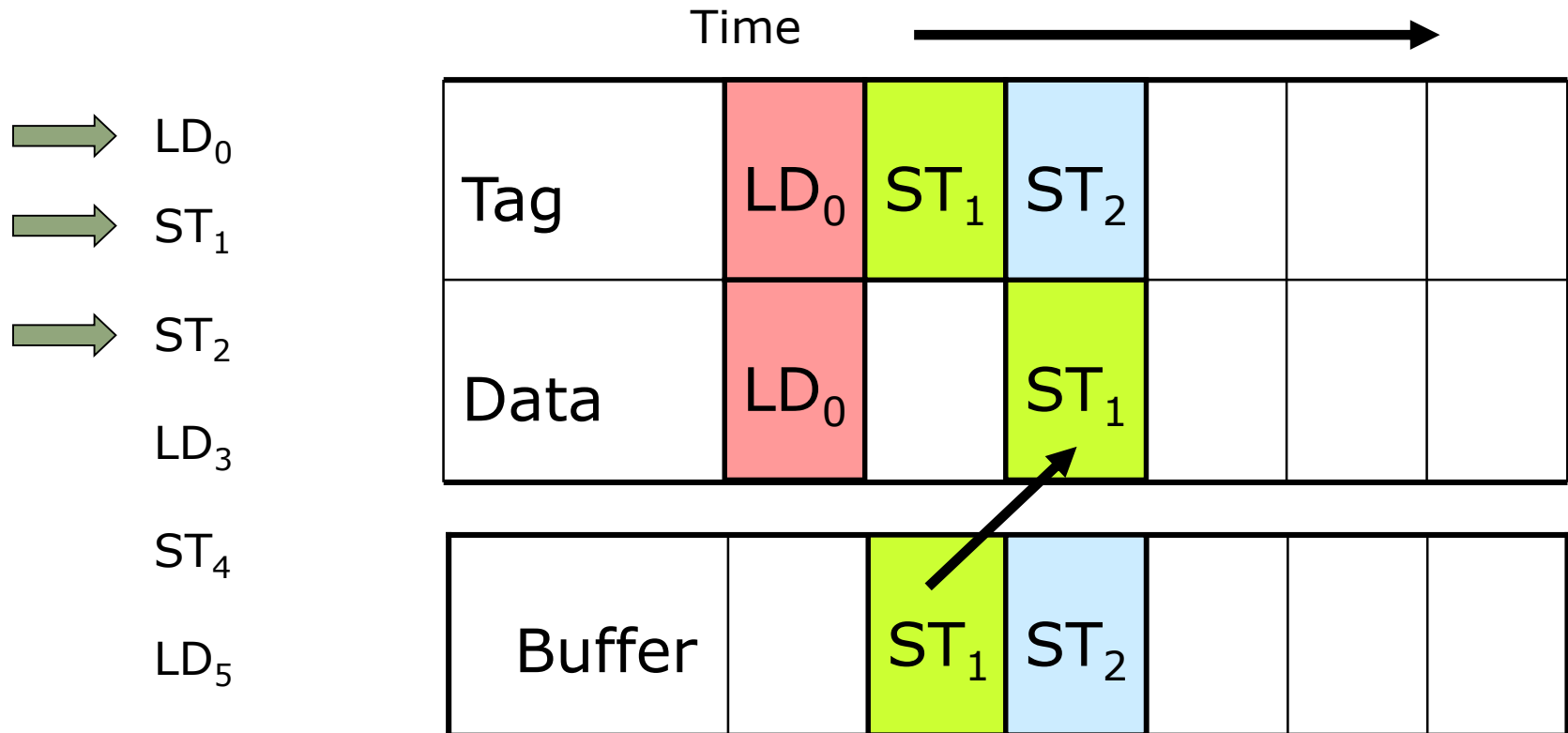
Solution: Write data during idle data cycle of next store's tag check



# Pipelined/Delayed Write Timing

Problem: Need to commit lazily saved write data

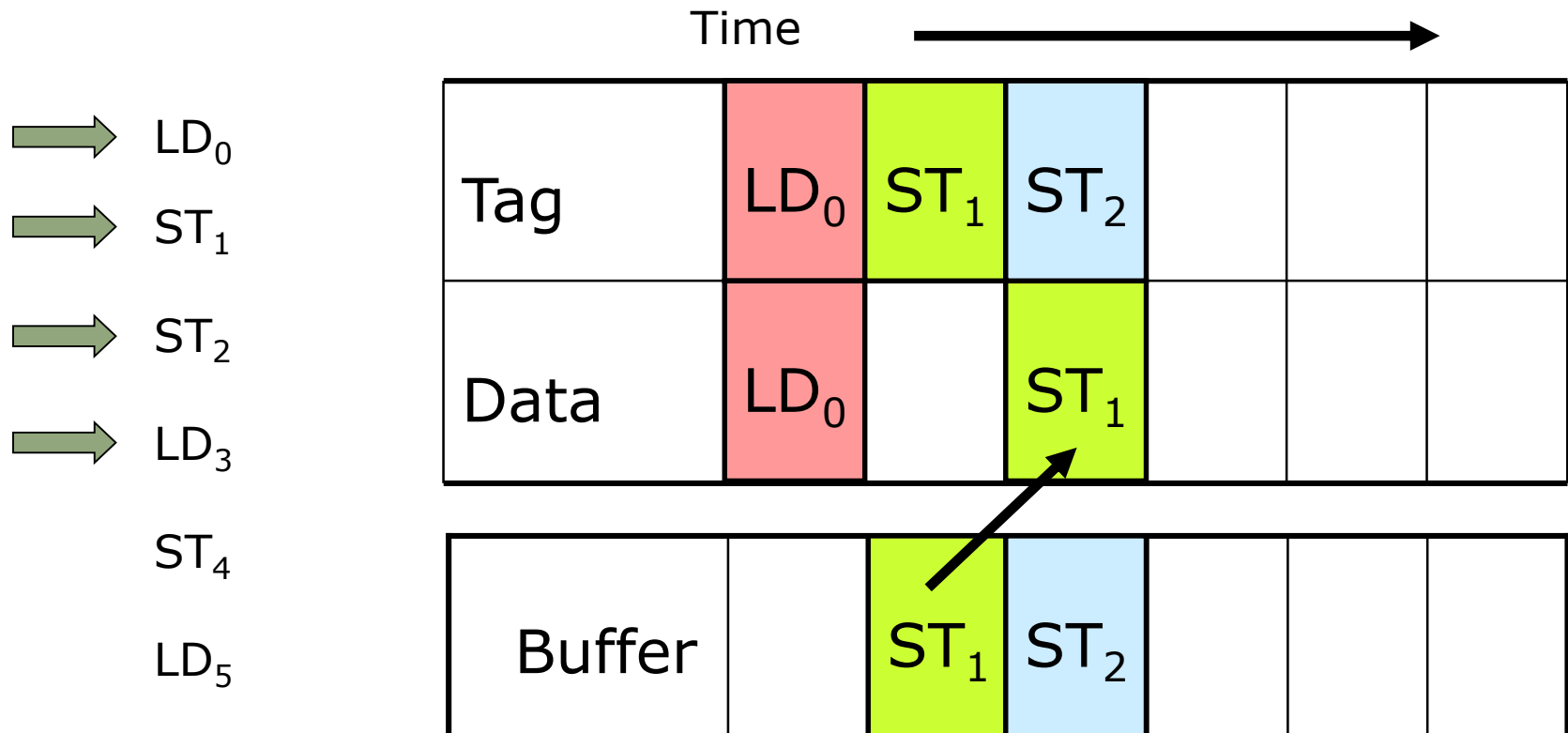
Solution: Write data during idle data cycle of next store's tag check



# Pipelined/Delayed Write Timing

Problem: Need to commit lazily saved write data

Solution: Write data during idle data cycle of next store's tag check

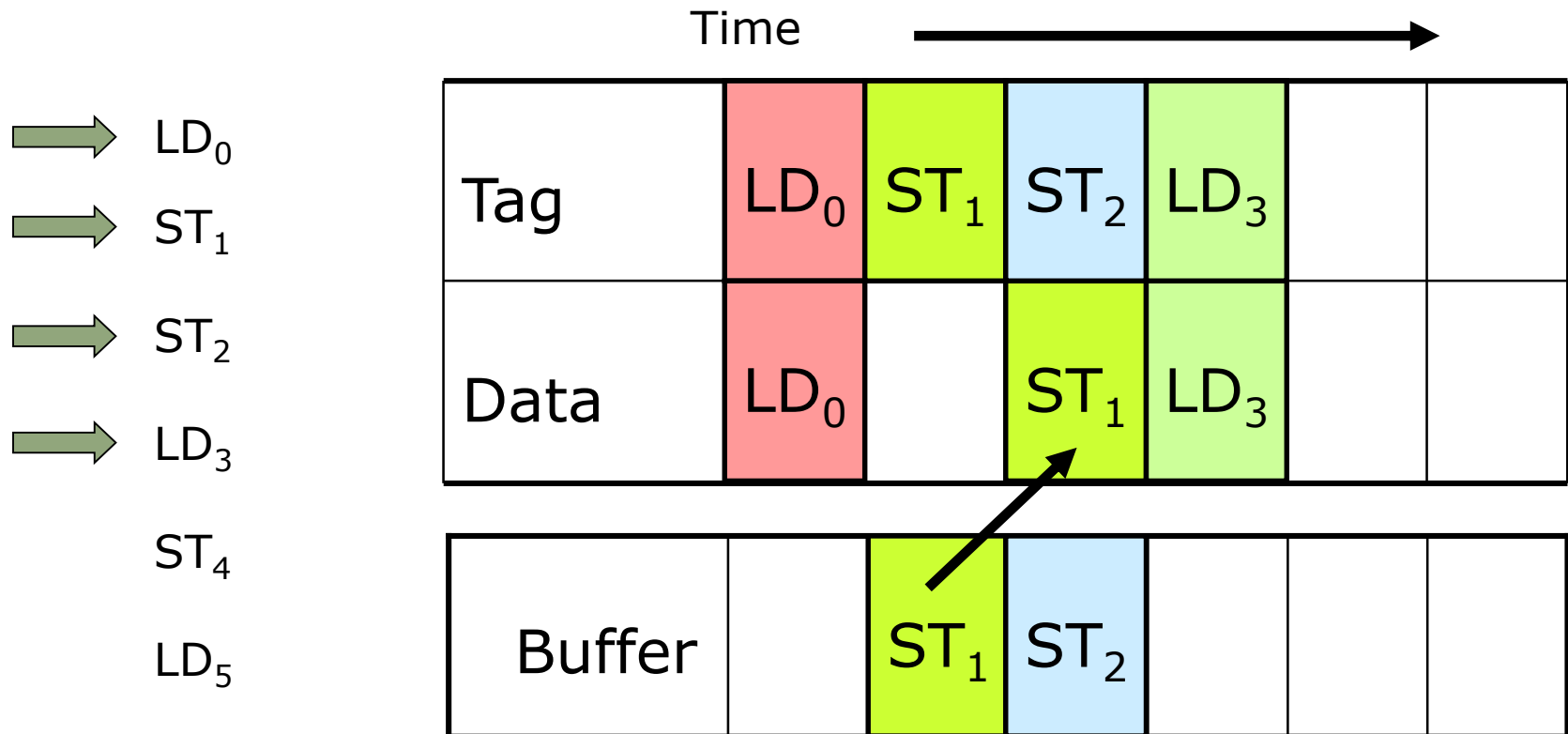




# Pipelined/Delayed Write Timing

Problem: Need to commit lazily saved write data

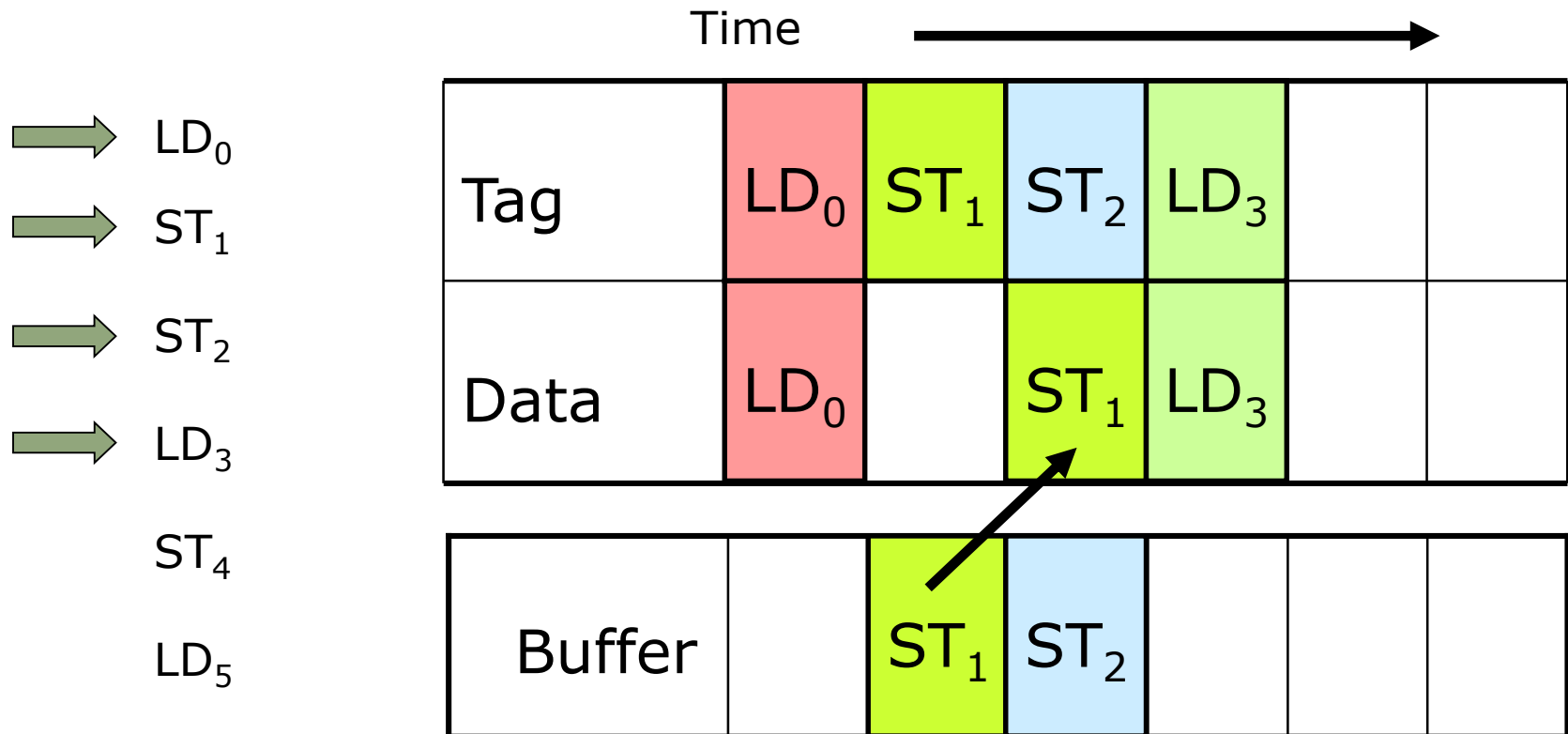
Solution: Write data during idle data cycle of next store's tag check



# Pipelined/Delayed Write Timing

Problem: Need to commit lazily saved write data

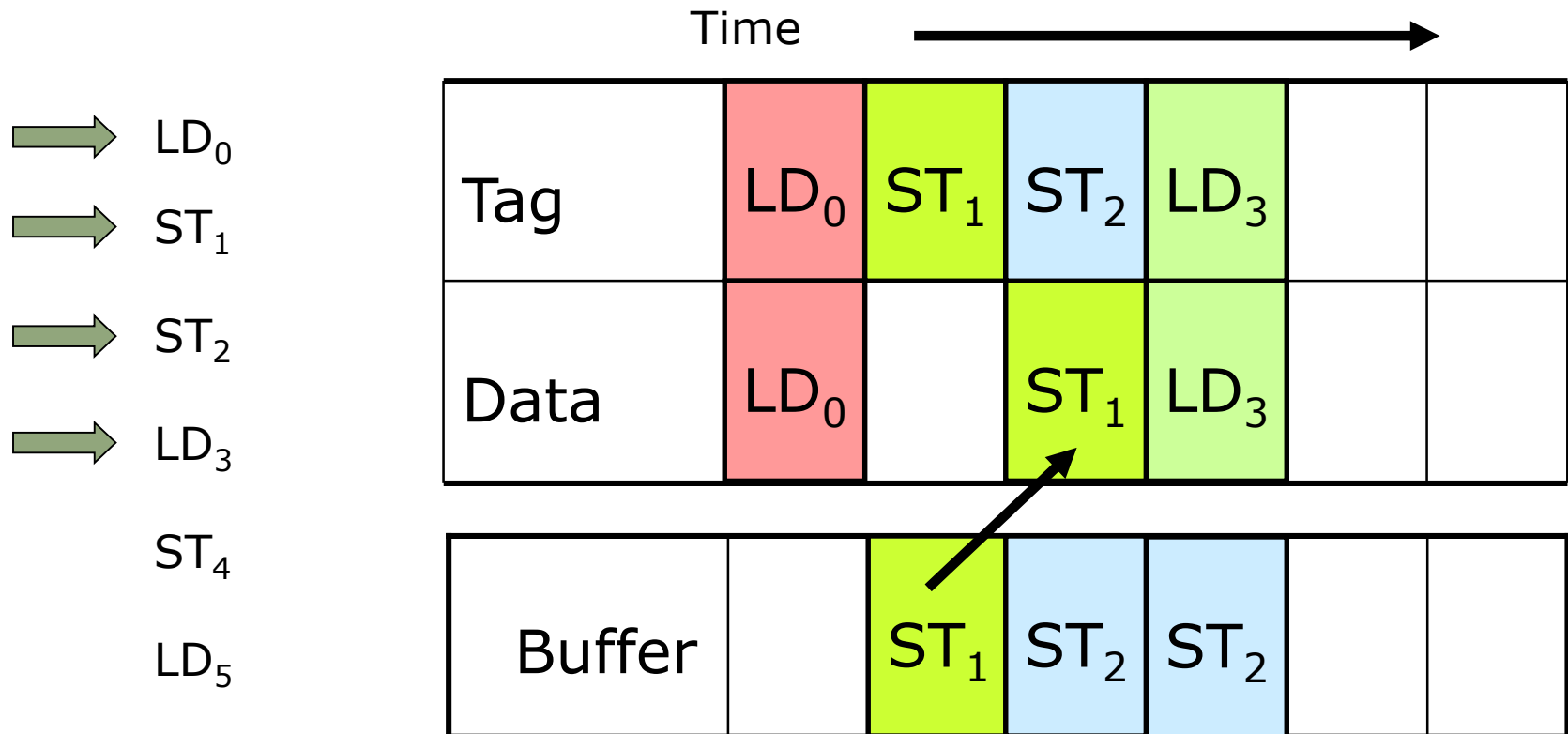
Solution: Write data during idle data cycle of next store's tag check



# Pipelined/Delayed Write Timing

Problem: Need to commit lazily saved write data

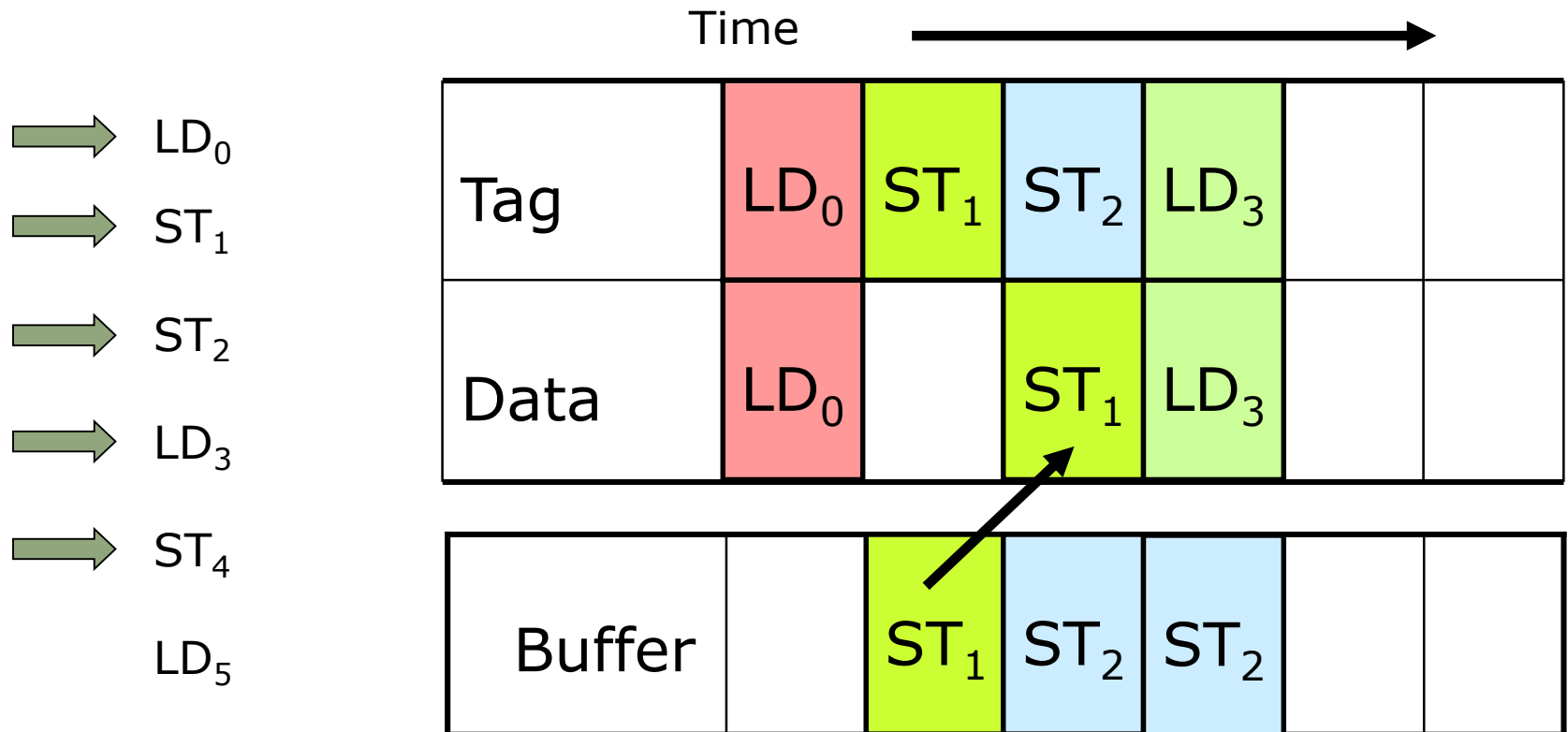
Solution: Write data during idle data cycle of next store's tag check



# Pipelined/Delayed Write Timing

Problem: Need to commit lazily saved write data

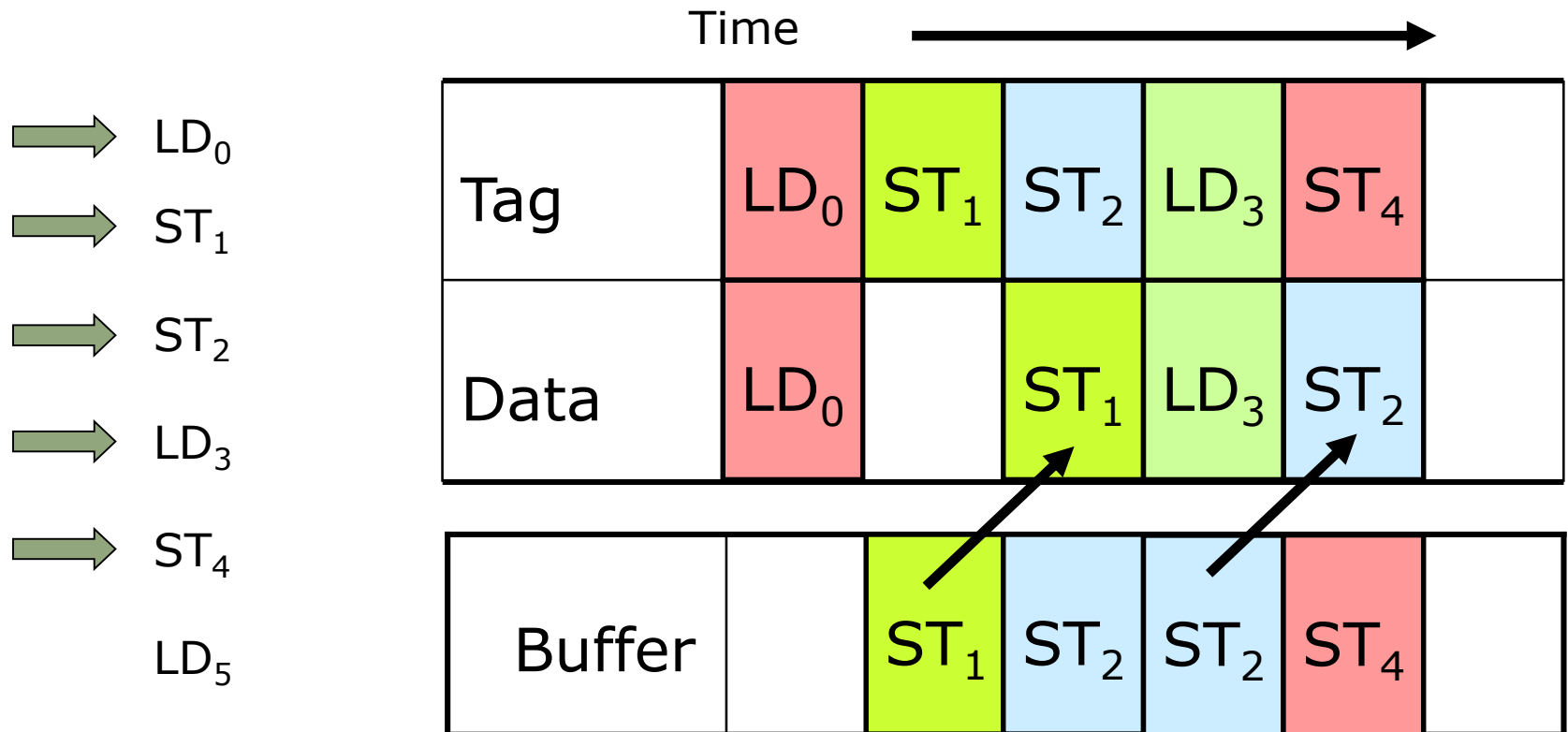
Solution: Write data during idle data cycle of next store's tag check



# Pipelined/Delayed Write Timing

Problem: Need to commit lazily saved write data

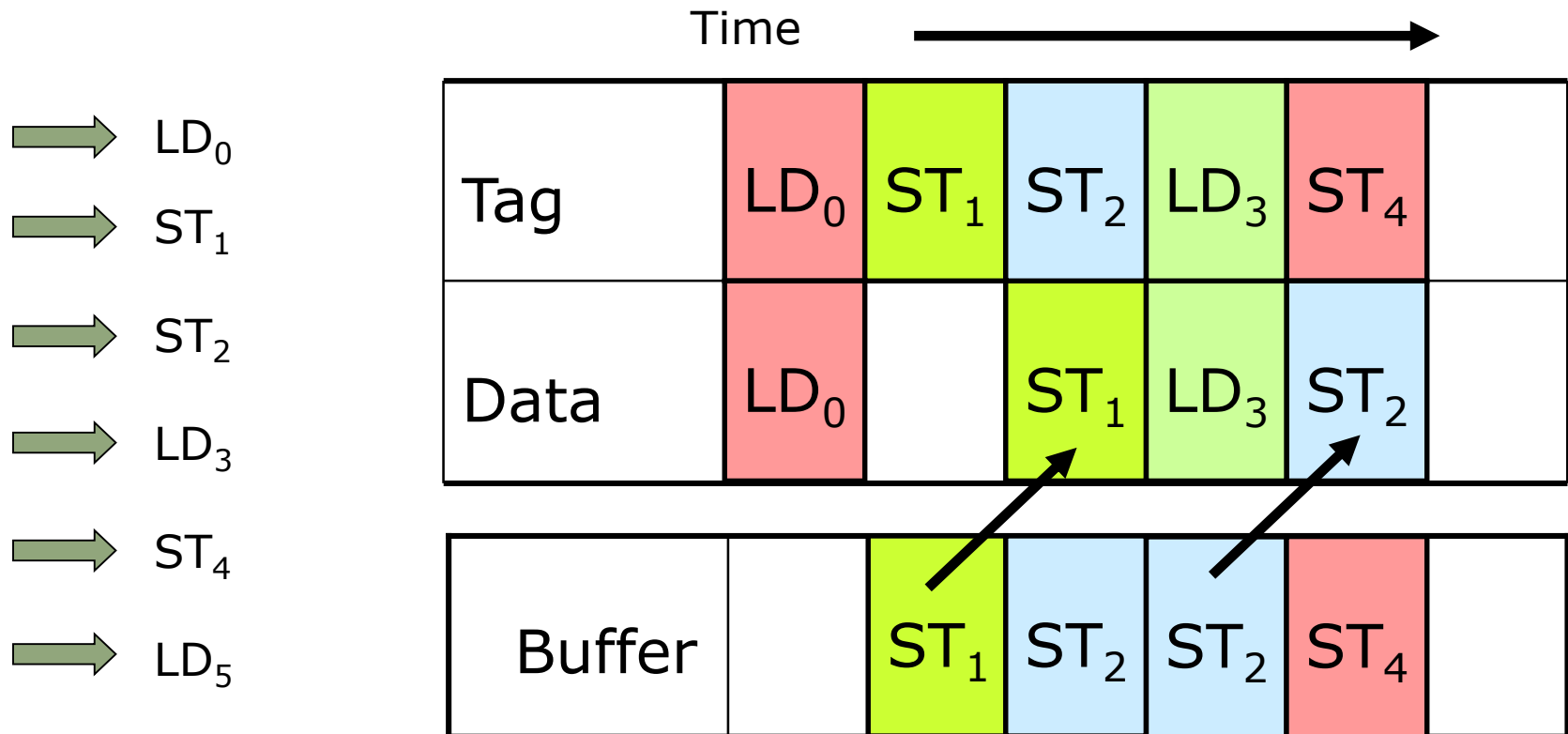
Solution: Write data during idle data cycle of next store's tag check



# Pipelined/Delayed Write Timing

Problem: Need to commit lazily saved write data

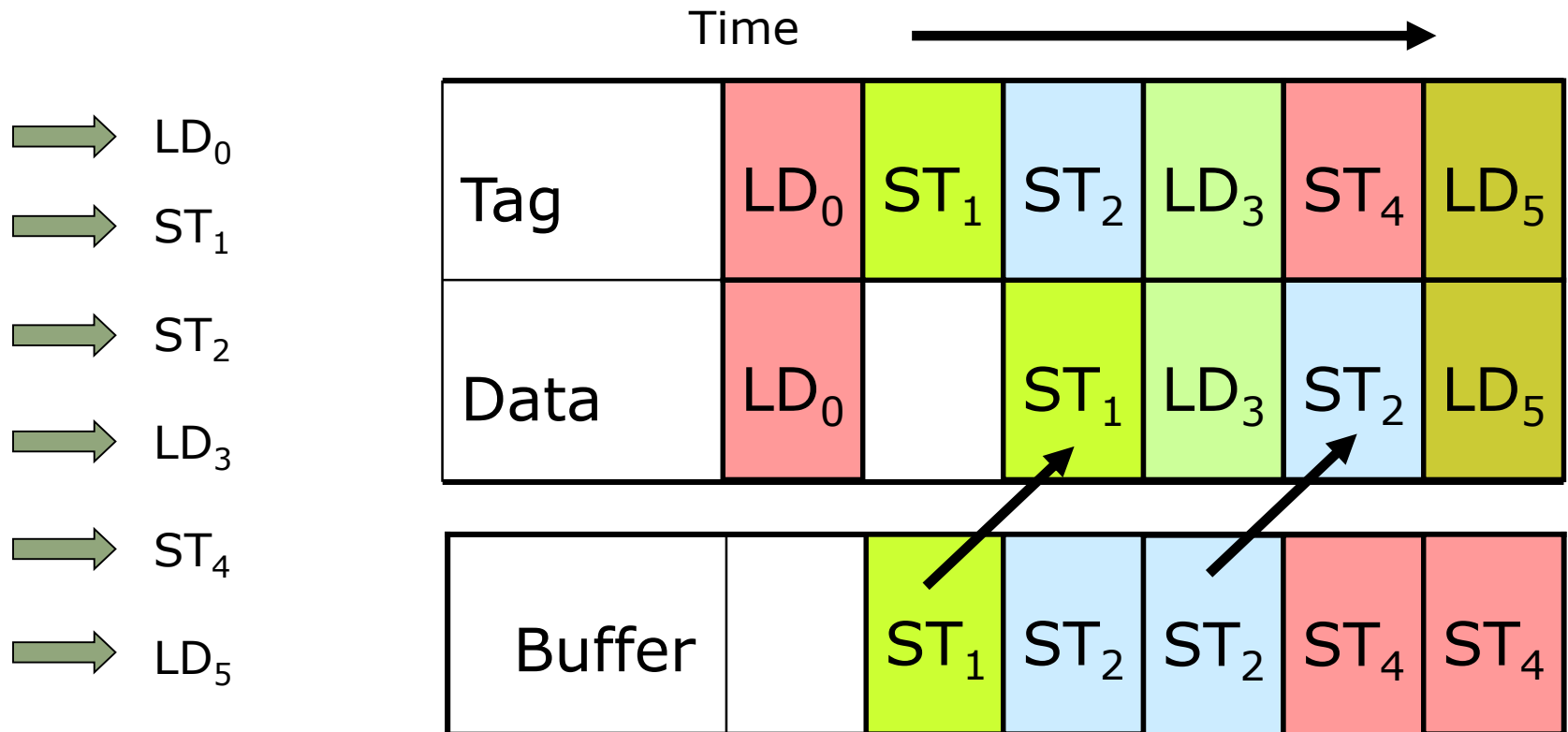
Solution: Write data during idle data cycle of next store's tag check



# Pipelined/Delayed Write Timing

Problem: Need to commit lazily saved write data

Solution: Write data during idle data cycle of next store's tag check



# Pipelining Cache Writes

---

*What if instruction needs data in delayed write buffer?*



# Pipelining Cache Writes

---

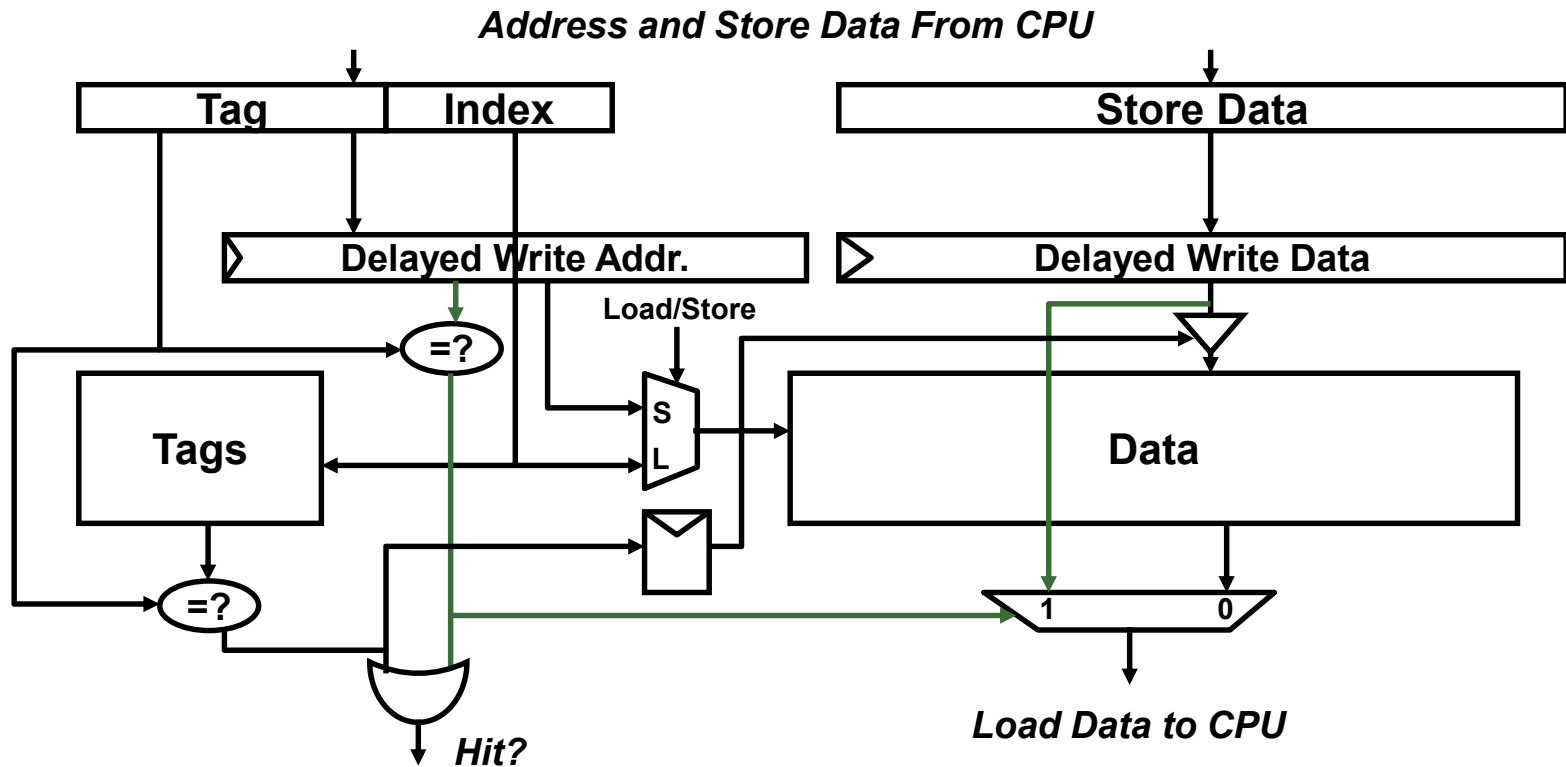
*What if instruction needs data in delayed write buffer?*

Bypass

# Pipelining Cache Writes

What if instruction needs data in delayed write buffer?

Bypass





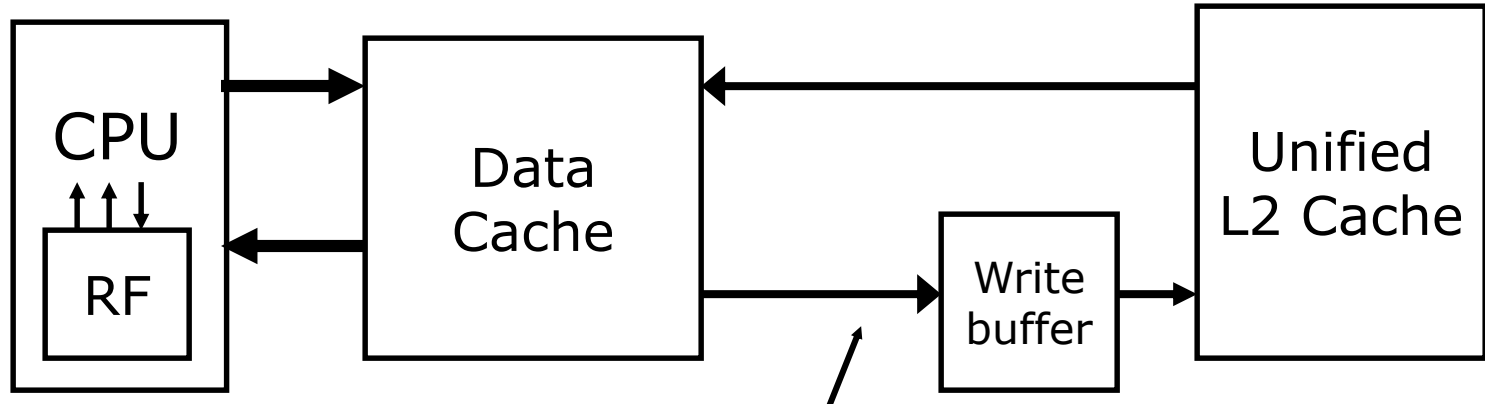
# Write Policy Choices

---

- Cache hit:
  - **Write-through:** write both cache & memory
    - generally higher traffic but simplifies multi-processor design
  - **Write-back:** write cache only  
(memory is written only when the entry is evicted)
    - a dirty bit per block can further reduce the traffic
- Cache miss:
  - **No-write-allocate:** only write to main memory
  - **Write-allocate** (*aka fetch on write*): fetch into cache
- Common combinations:
  - write-through and no-write-allocate
  - write-back with write-allocate

# Reducing Read Miss Penalty

---

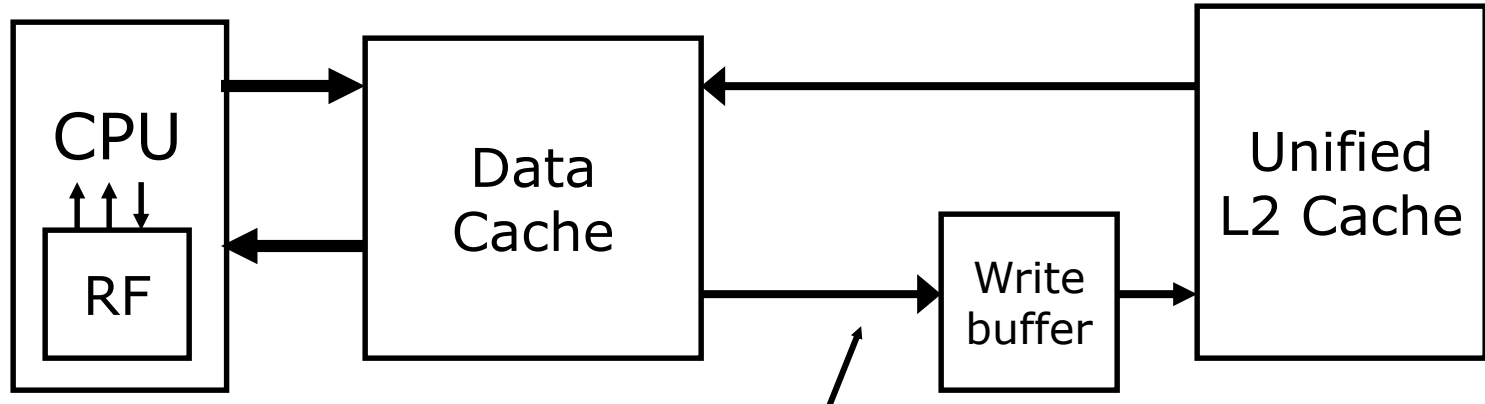


Evicted dirty lines for writeback cache  
OR  
All writes in writethrough cache

**Problem:** Write buffer may hold updated value of location needed by a read miss – RAW data hazard

# Reducing Read Miss Penalty

---

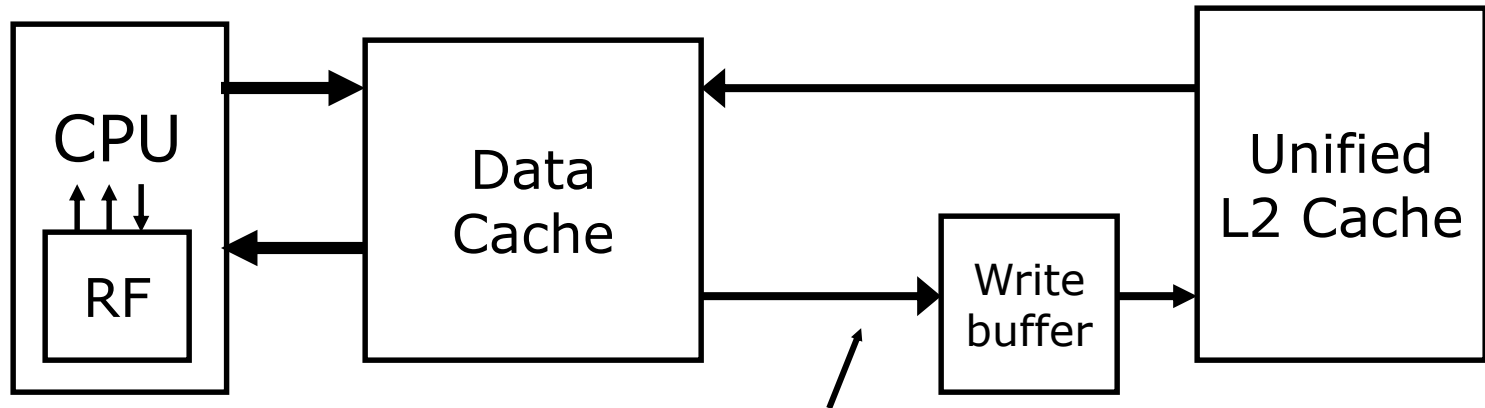


Evicted dirty lines for writeback cache  
OR  
All writes in writethrough cache

**Problem:** Write buffer may hold updated value of location needed by a read miss – RAW data hazard

**Stall:** On a read miss, wait for the write buffer to go empty

# Reducing Read Miss Penalty



Evicted dirty lines for writeback cache  
OR  
All writes in writethrough cache

**Problem:** Write buffer may hold updated value of location needed by a read miss – RAW data hazard

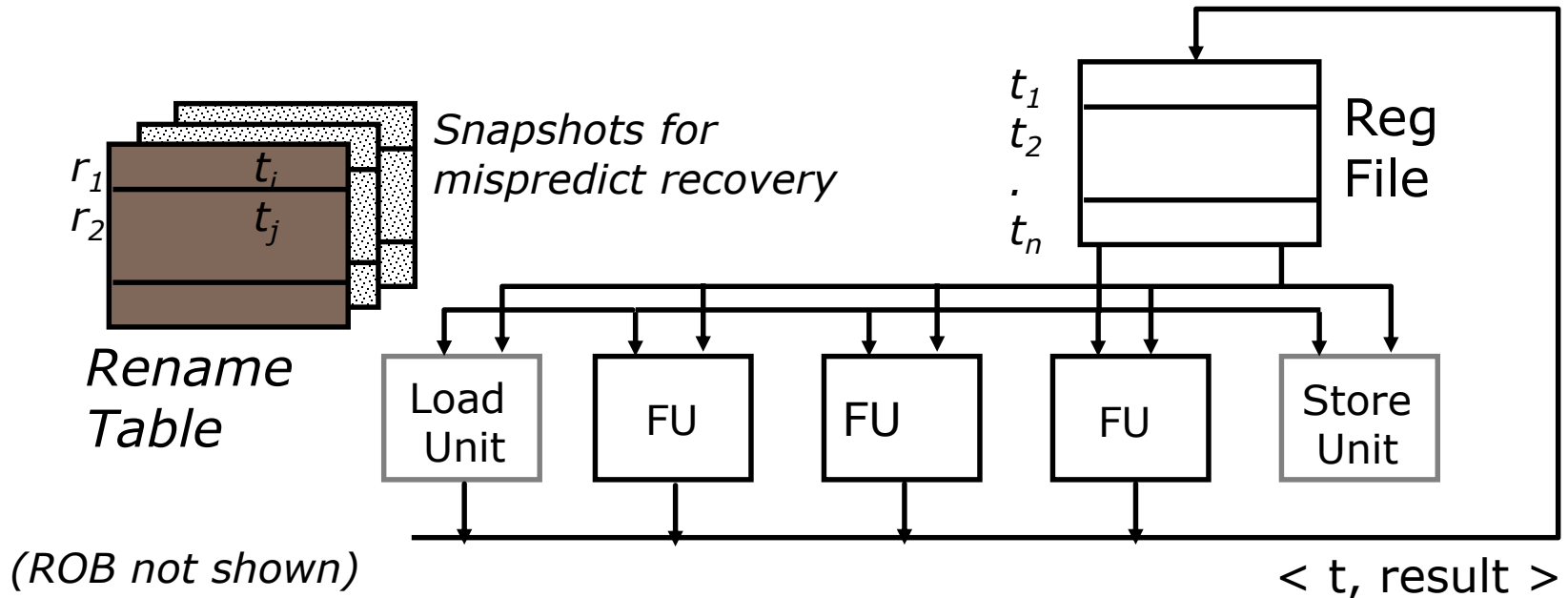
**Stall:** On a read miss, wait for the write buffer to go empty

**Bypass:** Check write buffer addresses against read miss addresses, if no match, allow read miss to go ahead of writes, else, return value in write buffer

# O-o-O With Physical Register File

(MIPS R10K, Alpha 21264, Pentium 4)

---



We've handled the register dependencies, but what about memory operations?



# Speculative Loads / Stores

---

- Problem: Just like register updates, stores should not permanently change the architectural memory state until after the instruction is committed
- Choice: Data update policy: greedy or lazy?

# Speculative Loads / Stores

---

- Problem: Just like register updates, stores should not permanently change the architectural memory state until after the instruction is committed
- Choice: Data update policy: greedy or lazy?
  - **Lazy:** Add a speculative store buffer, a structure to lazily hold speculative store data.

# Speculative Loads / Stores

---

- Problem: Just like register updates, stores should not permanently change the architectural memory state until after the instruction is committed
- Choice: Data update policy: greedy or lazy?  
**Lazy:** Add a speculative store buffer, a structure to lazily hold speculative store data.
- Choice: Handling of store-to-load data hazards: stall, bypass, speculate...?

# Speculative Loads / Stores

---

- Problem: Just like register updates, stores should not permanently change the architectural memory state until after the instruction is committed
- Choice: Data update policy: greedy or lazy?
  - **Lazy:** Add a speculative store buffer, a structure to lazily hold speculative store data.
- Choice: Handling of store-to-load data hazards: stall, bypass, speculate...?
  - **Bypass:** ...

# Store Buffer Responsibilities

---

# Store Buffer Responsibilities

---

- **Lazy store of data:** Buffer new data values for stores

# Store Buffer Responsibilities

---

- **Lazy store of data:** Buffer new data values for stores
- **Commit/abort:** The data from the oldest instructions must either be committed to memory or forgotten

# Store Buffer Responsibilities

---

- **Lazy store of data:** Buffer new data values for stores
- **Commit/abort:** The data from the oldest instructions must either be committed to memory or forgotten

*Commits are generally done in order – why?*



# Store Buffer Responsibilities

---

- **Lazy store of data:** Buffer new data values for stores
- **Commit/abort:** The data from the oldest instructions must either be committed to memory or forgotten

*Commits are generally done in order – why?*

**WAW Hazards**

# Store Buffer Responsibilities

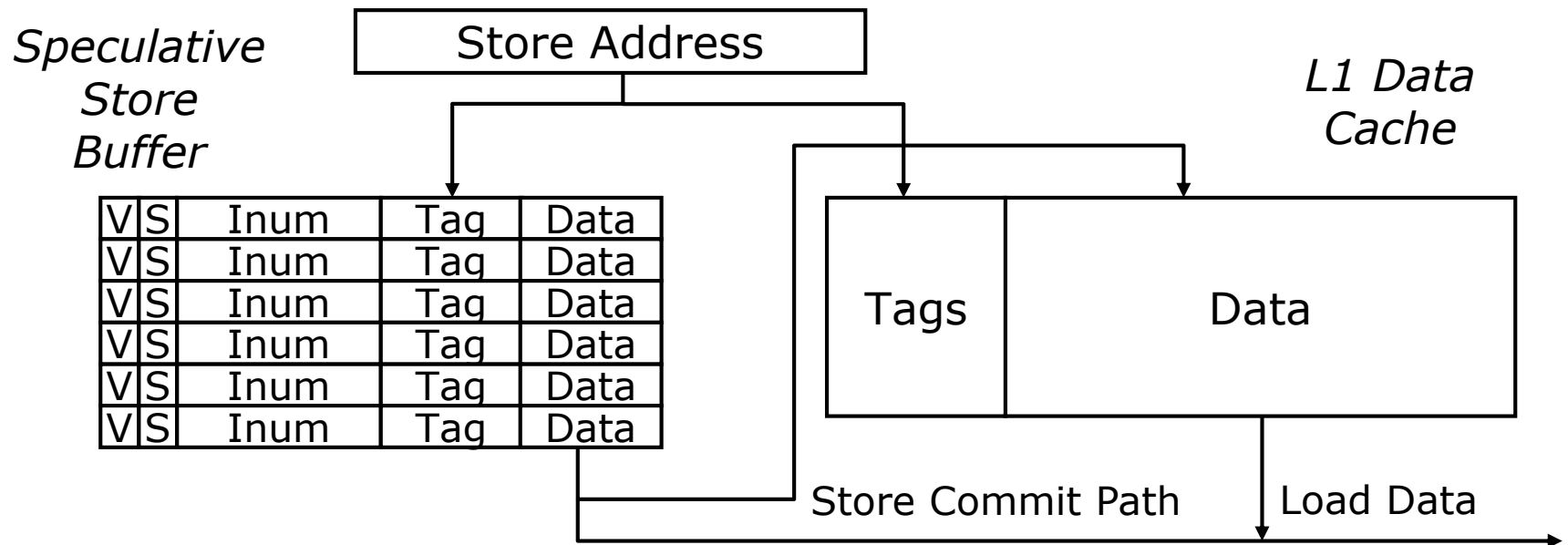
---

- **Lazy store of data:** Buffer new data values for stores
- **Commit/abort:** The data from the oldest instructions must either be committed to memory or forgotten
- **Bypass:** Data from older instructions must be provided (or forwarded) to younger instructions before the older instruction is committed

*Commits are generally done in order – why?*

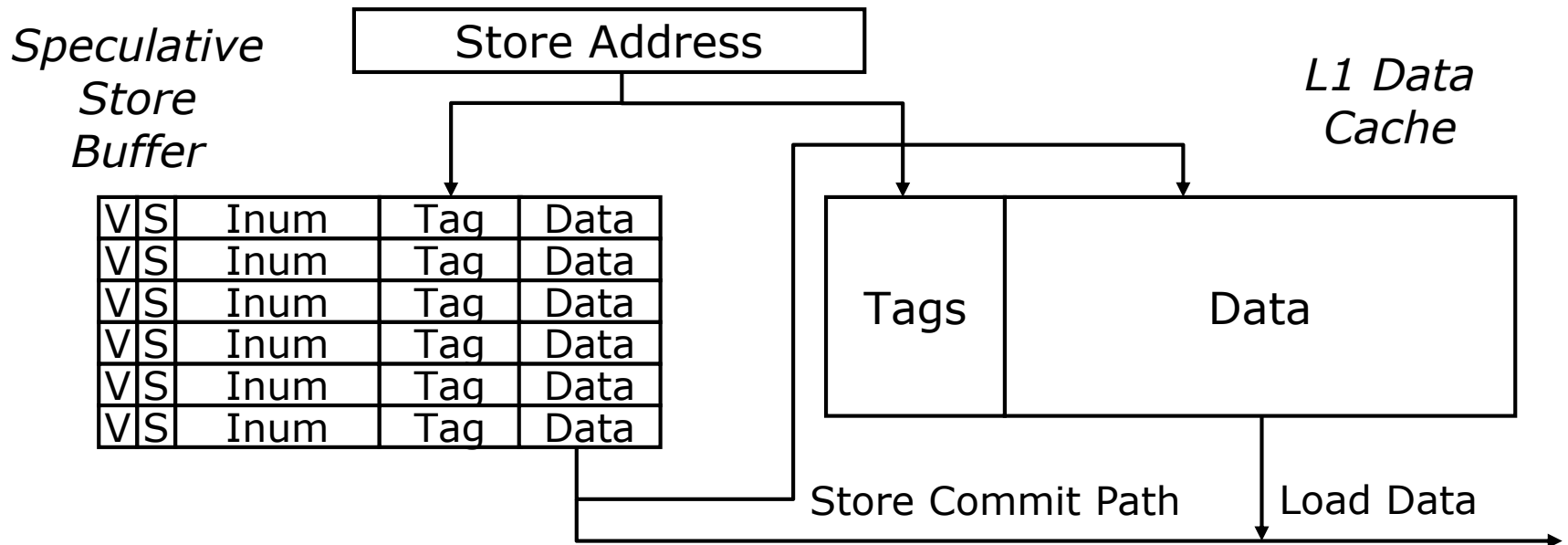
**WAW Hazards**

# Store Buffer – Lazy data management



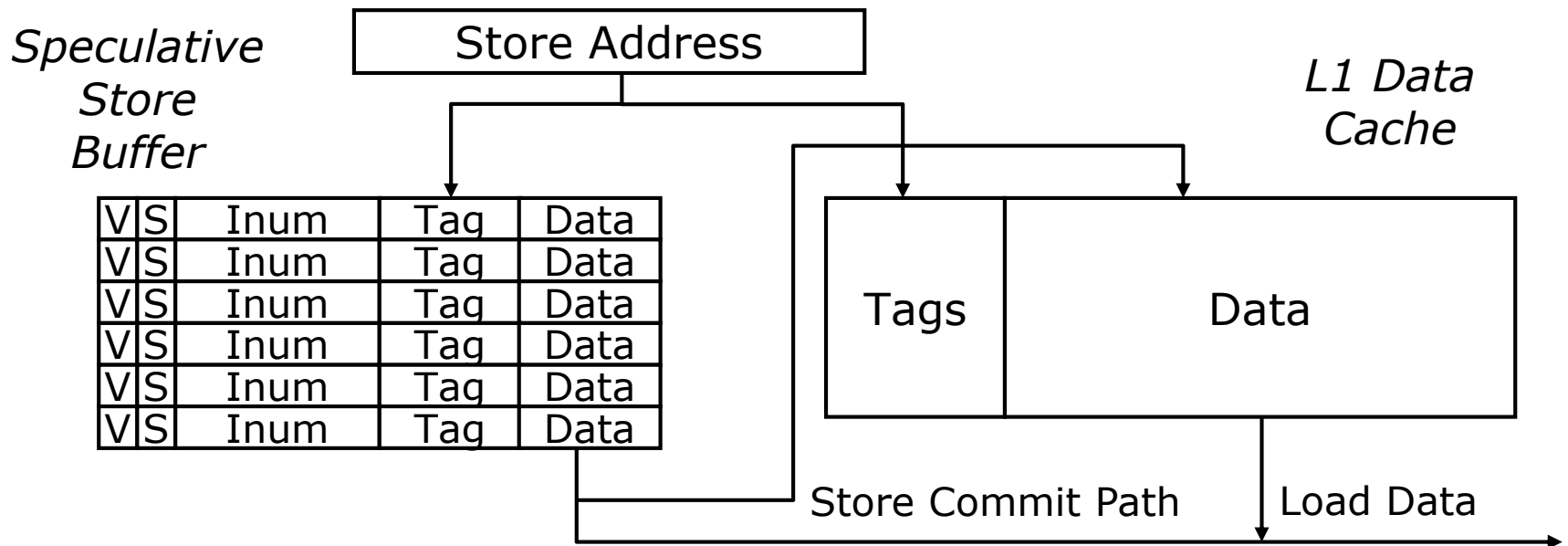
- On store execute:

# Store Buffer – Lazy data management



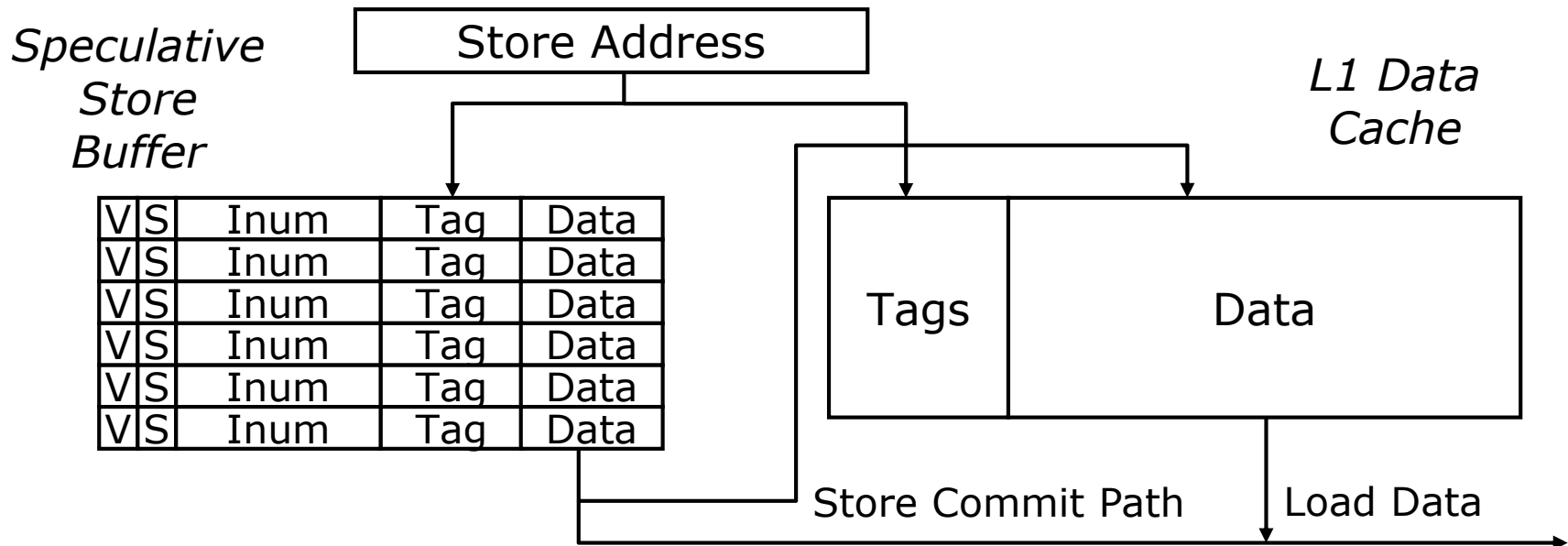
- On store execute:
  - mark valid and speculative; save tag, data, and instruction number

# Store Buffer – Lazy data management



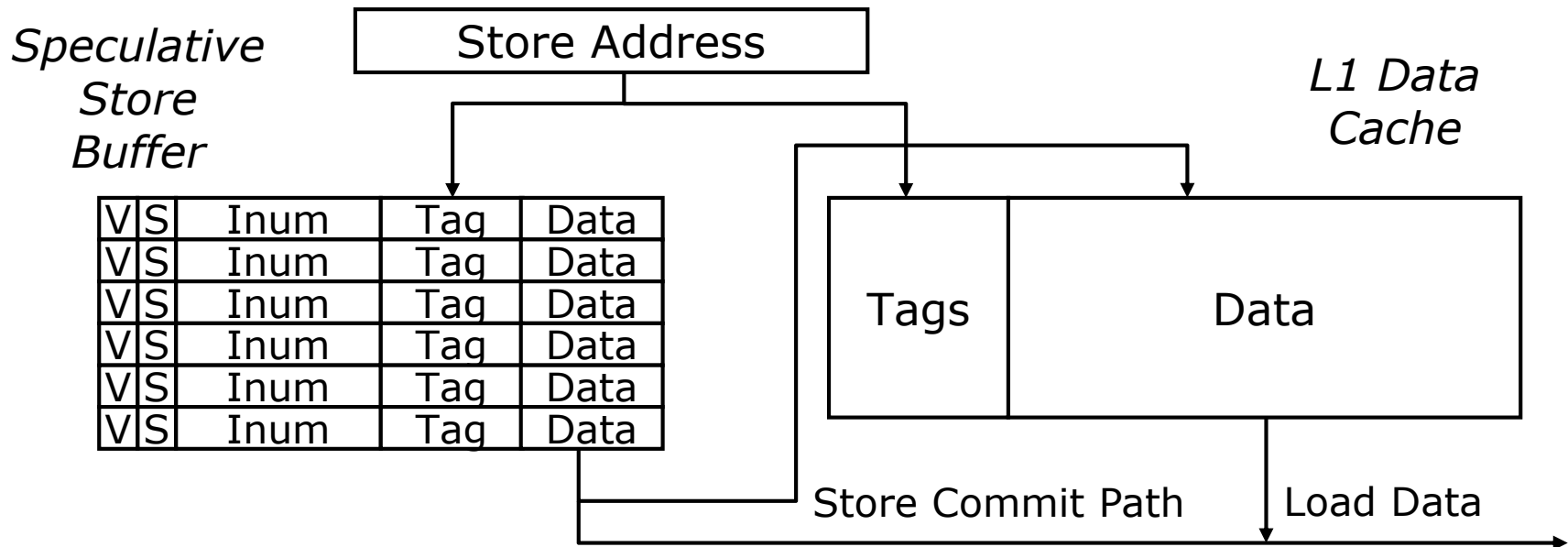
- On store execute:
  - mark valid and speculative; save tag, data, and instruction number
- On store commit:

# Store Buffer – Lazy data management



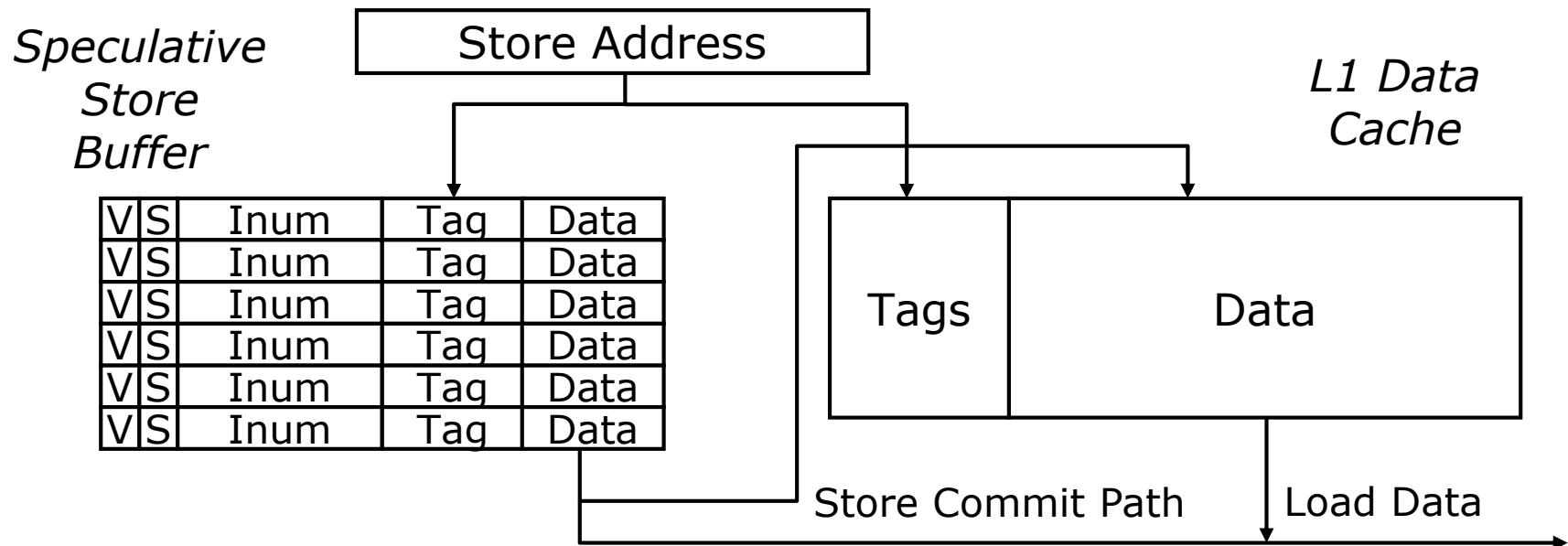
- On store execute:
  - mark valid and speculative; save tag, data, and instruction number
- On store commit:
  - clear speculative bit and eventually move data to cache

# Store Buffer – Lazy data management



- On store execute:
  - mark valid and speculative; save tag, data, and instruction number
- On store commit:
  - clear speculative bit and eventually move data to cache
- On store abort:

# Store Buffer – Lazy data management



- On store execute:
  - mark valid and speculative; save tag, data, and instruction number
- On store commit:
  - clear speculative bit and eventually move data to cache
- On store abort:
  - clear valid bit



# Store Buffer - Bypassing

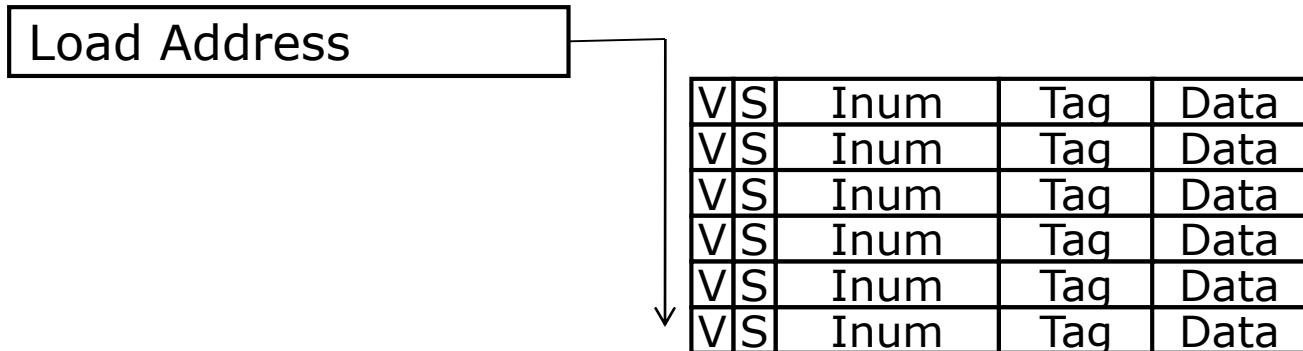
---

Load Address

V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data

# Store Buffer - Bypassing

---



# Store Buffer - Bypassing

---

Load Address

What fields must be examined for bypassing?

V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data

# Store Buffer - Bypassing

---

Load Address

What fields must be examined for bypassing?

Valid, Inum, and tag

V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data

# Store Buffer - Bypassing

---

Load Address

What fields must be examined for bypassing?

Valid, Inum, and tag

V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data

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

# Store Buffer - Bypassing

---

Load Address

What fields must be examined for bypassing?

Valid, Inum, and tag

V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data

- If data in both store buffer and cache, which should we use?  
Speculative store buffer - if store older than load

# Store Buffer - Bypassing

---

Load Address

What fields must be examined for bypassing?

Valid, Inum, and tag

V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data

- If data in both store buffer and cache, which should we use?  
Speculative store buffer - if store older than load
- If same address in store buffer twice, which should we use?

# Store Buffer - Bypassing

---

Load Address

What fields must be examined for bypassing?

Valid, Inum, and tag

V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data

- If data in both store buffer and cache, which should we use?  
Speculative store buffer - if store older than load
- If same address in store buffer twice, which should we use?  
Youngest store older than load



# Store Buffer - Bypassing

Load Address

What fields must be examined for bypassing?

Valid, Inum, and tag

V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data

- If data in both store buffer and cache, which should we use?  
Speculative store buffer - if store older than load
- If same address in store buffer twice, which should we use?  
Youngest store older than load
- Calculating entry needed in the store buffer can be considered a dependence on the index needed to access the store buffer. So store buffer bypassing can be managed speculatively by building a simple predictor that guesses that the specific entry in the store buffer the load needs. So what happens if we guessed the wrong entry?

# Store Buffer - Bypassing

Load Address

What fields must be examined for bypassing?

Valid, Inum, and tag

V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data

- If data in both store buffer and cache, which should we use?  
Speculative store buffer - if store older than load
- If same address in store buffer twice, which should we use?  
Youngest store older than load
- Calculating entry needed in the store buffer can be considered a dependence on the index needed to access the store buffer. So store buffer bypassing can be managed speculatively by building a simple predictor that guesses that the specific entry in the store buffer the load needs. So what happens if we guessed the wrong entry?  
Declare a mis-speculation and abort.

# Memory Dependencies

---

For registers, we used tags or physical register numbers to determine dependencies. What about memory operations?

**st r1, (r2)**

**ld r3, (r4)**

*When is the load dependent on the store?*

# Memory Dependencies

---

For registers, we used tags or physical register numbers to determine dependencies. What about memory operations?

**st r1, (r2)**

**ld r3, (r4)**

*When is the load dependent on the store?*

When  $r2 == r4$

# Memory Dependencies

---

For registers, we used tags or physical register numbers to determine dependencies. What about memory operations?

**st r1, (r2)**

**ld r3, (r4)**

*When is the load dependent on the store?*

**When  $r2 == r4$**

*Does our ROB know this at issue time?*

# Memory Dependencies

---

For registers, we used tags or physical register numbers to determine dependencies. What about memory operations?

**st r1, (r2)**

**ld r3, (r4)**

*When is the load dependent on the store?*

**When  $r2 == r4$**

*Does our ROB know this at issue time?* **No**

# In-Order Memory Queue

---

**st r1, (r2)**  
**ld r3, (r4)**

Stall naively:

- Execute all loads and stores in program order

=> Load and store cannot start 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

# Conservative O-o-O Load Execution

---

**st r1, (r2)**  
**ld r3, (r4)**

Stall intelligently:

- Split execution of store instruction into two phases:  
address calculation and data write
- Can execute load before store, if addresses known and  $r4 \neq r2$
- Each load address compared with addresses of all previous uncommitted stores (*can use partial conservative check, e.g., bottom 12 bits of address*)
- Don't execute load if any previous store address not known

*(MIPS R10K, 16 entry address queue)*



# Address Speculation

---

**st r1, (r2)**  
**ld r3, (r4)**

# Address Speculation

---

```
st r1, (r2)  
ld r3, (r4)
```

1. Guess that  $r4 \neq r2$ , and execute load before store address known

# Address Speculation

---

```
st r1, (r2)  
ld r3, (r4)
```

1. Guess that  $r4 \neq r2$ , and execute load before store address known
2. If  $r4 \neq r2$  commit...

# Address Speculation

---

**st r1, (r2)**  
**ld r3, (r4)**

1. Guess that  $r4 \neq r2$ , and execute load before store address known
2. If  $r4 \neq r2$  commit...
3. But if  $r4 = r2$ , squash load and *all* following instructions
  - To support squash we need to hold all completed but uncommitted load/store addresses/data in program order

# Address Speculation

---

**st r1, (r2)**  
**ld r3, (r4)**

1. Guess that  $r4 \neq r2$ , and execute load before store address known
2. If  $r4 \neq r2$  commit...
3. But if  $r4 == r2$ , squash load and *all* following instructions
  - To support squash we need to hold all completed but uncommitted load/store addresses/data in program order

*How do we resolve the speculation, i.e., detect when we need to squash?*

# Address Speculation

---

**st r1, (r2)**  
**ld r3, (r4)**

1. Guess that  $r4 \neq r2$ , and execute load before store address known
2. If  $r4 \neq r2$  commit...
3. But if  $r4 = r2$ , squash load and *all* following instructions
  - To support squash we need to hold all completed but uncommitted load/store addresses/data in program order

*How do we resolve the speculation, i.e., detect when we need to squash?*

Watch for stores that arrive after load that needed its data

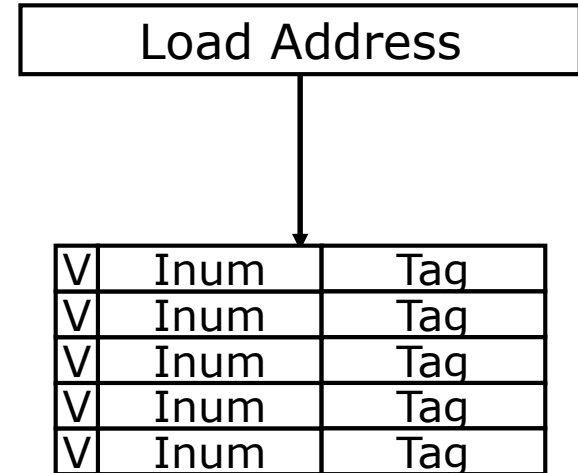
# Speculative Load Buffer

---

## **Speculation check:**

Detect if a load has executed before an earlier store to the same address – missed RAW hazard

*Speculative  
Load Buffer*



- On load execute:

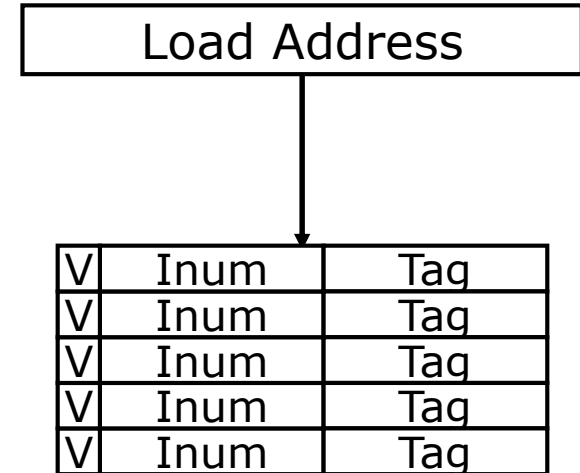
# Speculative Load Buffer

---

## **Speculation check:**

Detect if a load has executed before an earlier store to the same address – missed RAW hazard

*Speculative  
Load Buffer*



- On load execute:
  - mark entry valid, and instruction number and tag of data.



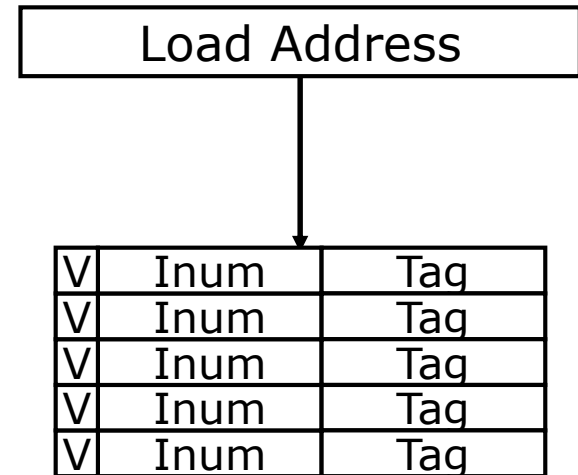
# Speculative Load Buffer

---

## Speculation check:

Detect if a load has executed before an earlier store to the same address – missed RAW hazard

*Speculative Load Buffer*



- On load execute:
  - mark entry valid, and instruction number and tag of data.
- On load commit:

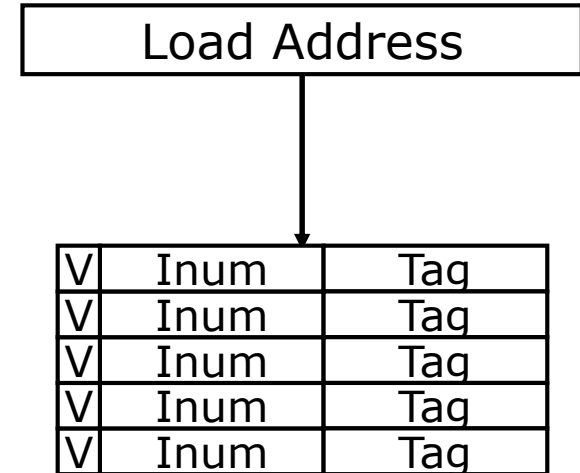
# Speculative Load Buffer

---

## Speculation check:

Detect if a load has executed before an earlier store to the same address – missed RAW hazard

*Speculative Load Buffer*



- On load execute:
  - mark entry valid, and instruction number and tag of data.
- On load commit:
  - clear valid bit

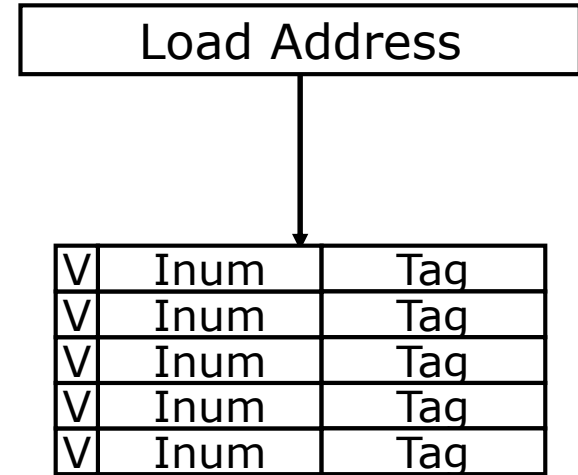
# Speculative Load Buffer

---

## Speculation check:

Detect if a load has executed before an earlier store to the same address – missed RAW hazard

*Speculative Load Buffer*



- On load execute:
  - mark entry valid, and instruction number and tag of data.
- On load commit:
  - clear valid bit
- On load abort:

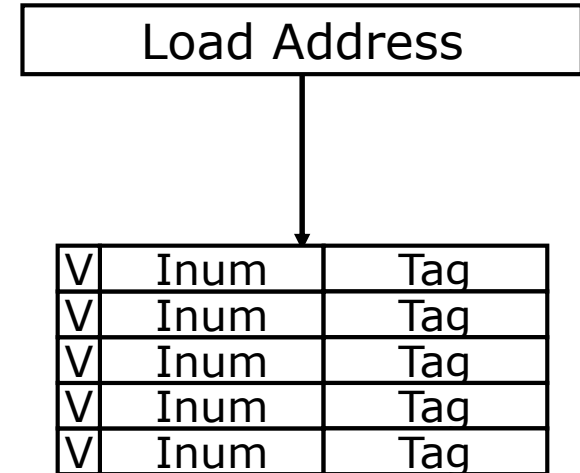
# Speculative Load Buffer

---

## Speculation check:

Detect if a load has executed before an earlier store to the same address – missed RAW hazard

*Speculative Load Buffer*

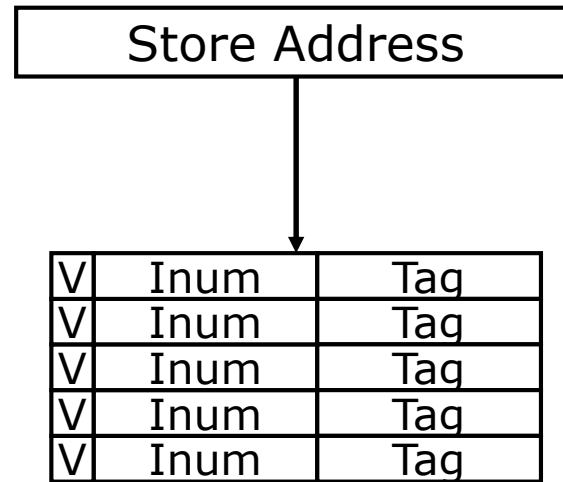


- On load execute:
  - mark entry valid, and instruction number and tag of data.
- On load commit:
  - clear valid bit
- On load abort:
  - clear valid bit

# Speculative Load Buffer

---

*Speculative  
Load Buffer*

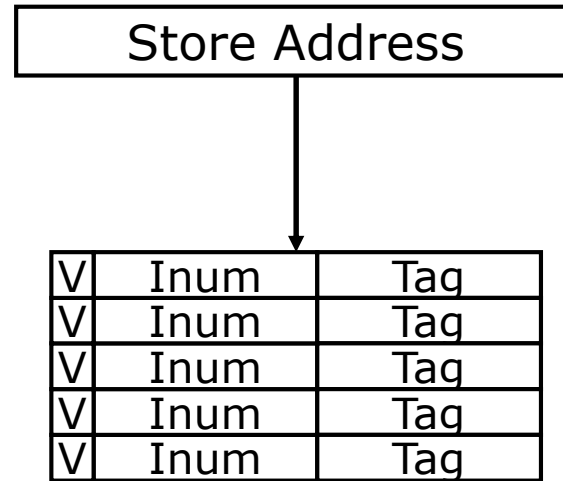


- If data in load buffer with instruction younger than store:

# Speculative Load Buffer

---

*Speculative  
Load Buffer*

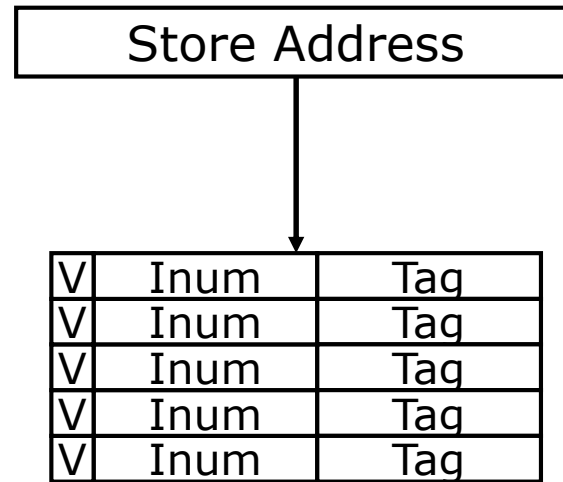


- If data in load buffer with instruction younger than store:
  - Speculative violation – abort!

# Speculative Load Buffer

---

*Speculative  
Load Buffer*

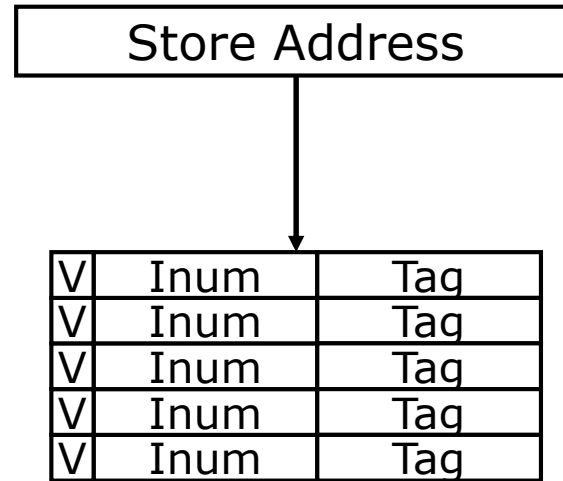


- If data in load buffer with instruction younger than store:
    - Speculative violation – abort!
- => Large penalty for inaccurate address speculation

# Speculative Load Buffer

---

*Speculative  
Load Buffer*



- If data in load buffer with instruction younger than store:
    - *Speculative violation – abort!*
- => Large penalty for inaccurate address speculation

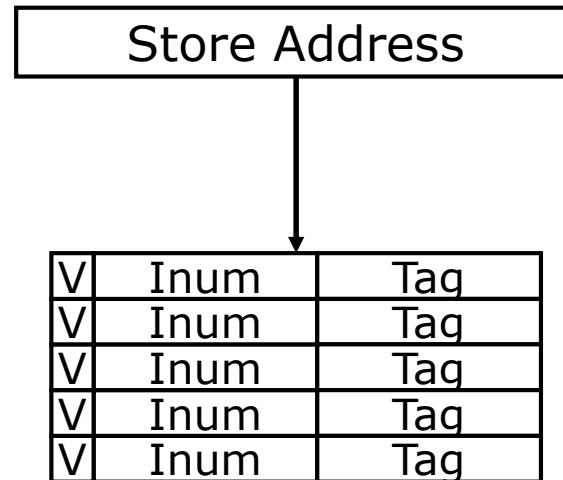
*Does tag match have to be perfect?*



# Speculative Load Buffer

---

*Speculative  
Load Buffer*



- If data in load buffer with instruction younger than store:
    - Speculative violation – abort!
- => Large penalty for inaccurate address speculation

*Does tag match have to be perfect?*

**No!**

# Memory Dependence Prediction

(Alpha 21264)

---

**st r1, (r2)**  
**ld r3, (r4)**

1. Guess that  $r4 \neq r2$  and execute load before store
2. If later find  $r4 == r2$ , 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

# Memory Dependence Prediction

(Alpha 21264)

---

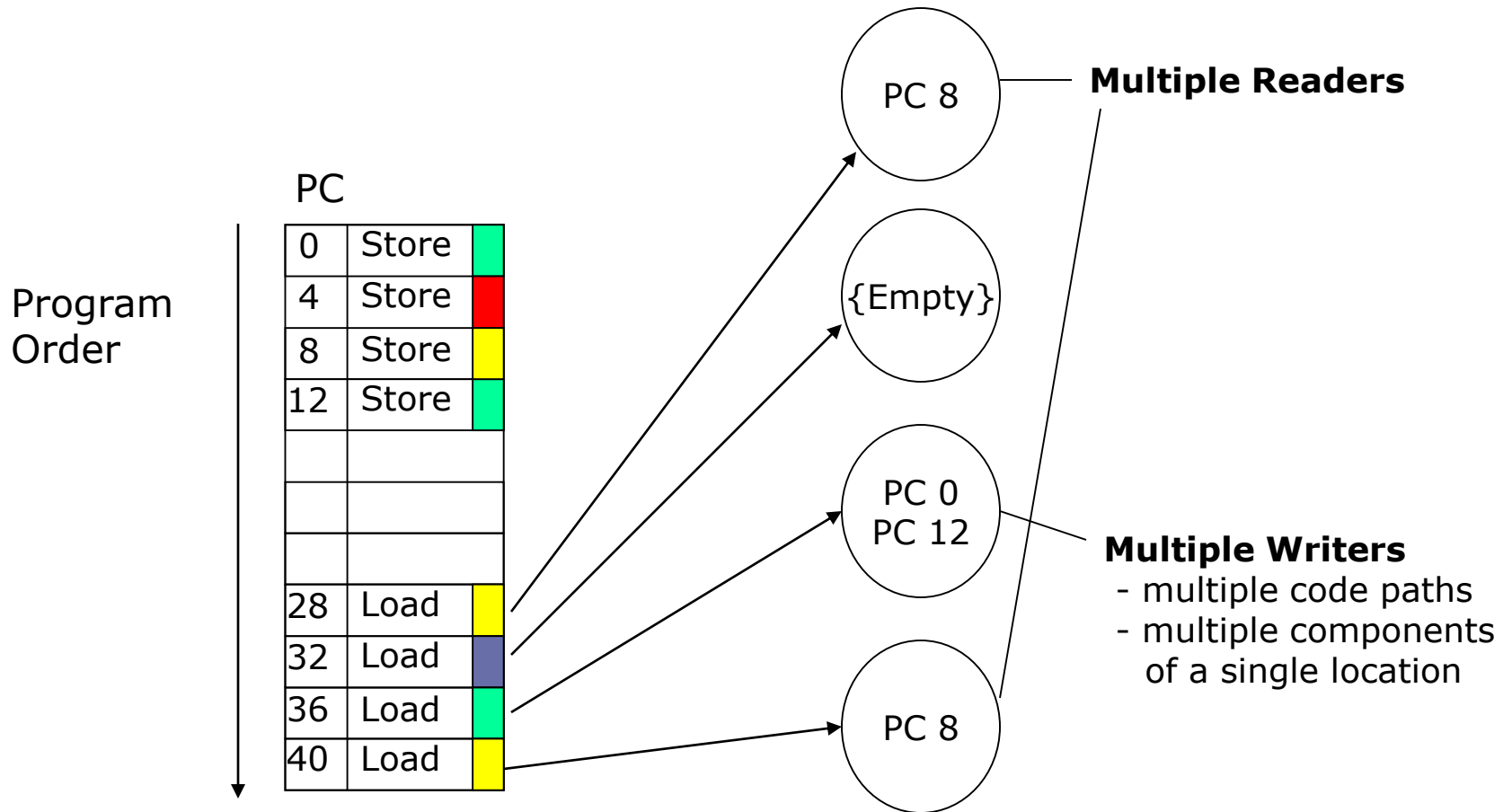
**st r1, (r2)**  
**ld r3, (r4)**

1. Guess that  $r4 \neq r2$  and execute load before store
2. If later find  $r4 == r2$ , 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

Notice the general problem of predictors that learn something but can't unlearn it

# Store Sets

(Alpha 21464)

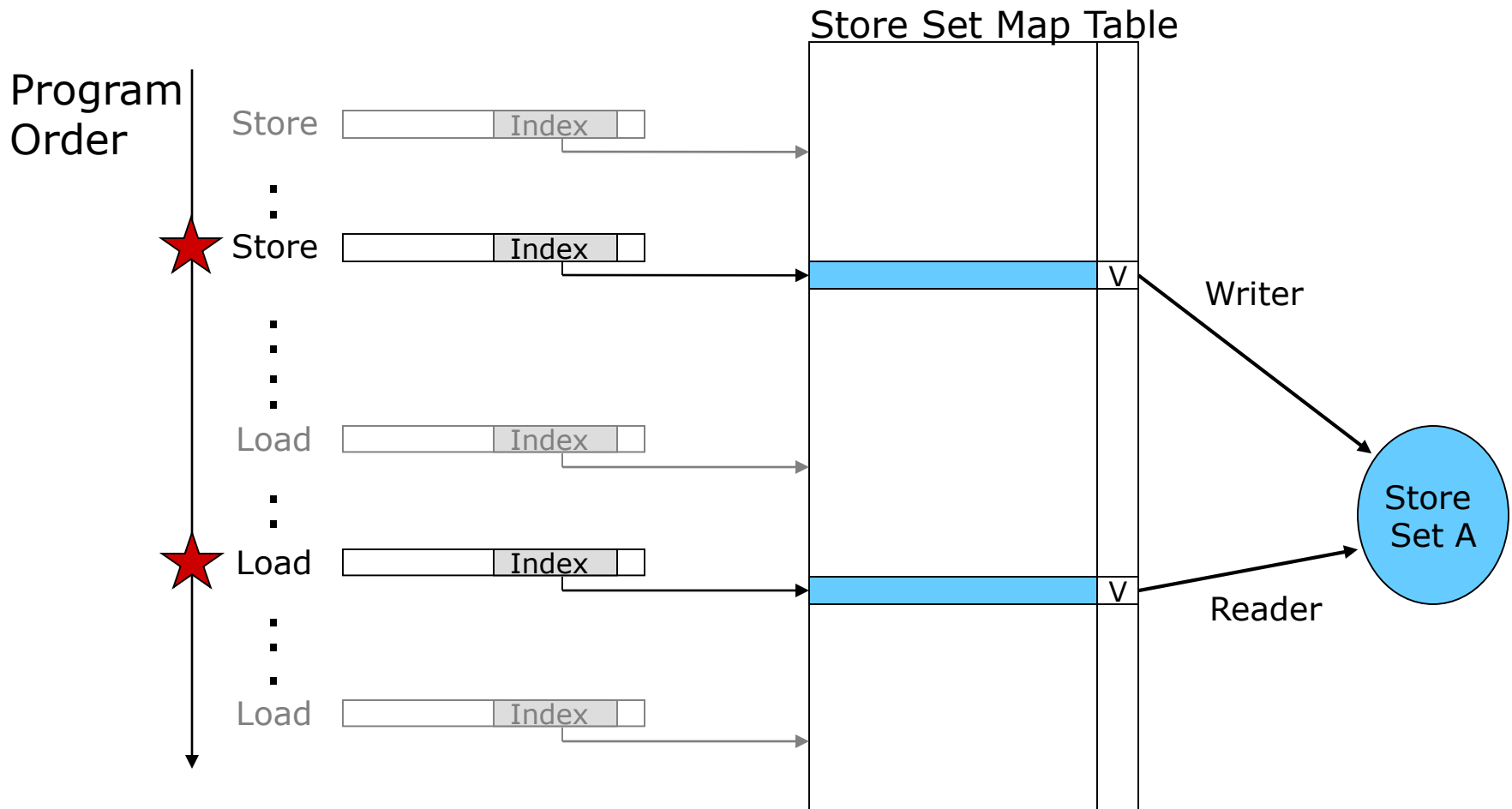


# Memory Dependence Prediction using Store Sets

---

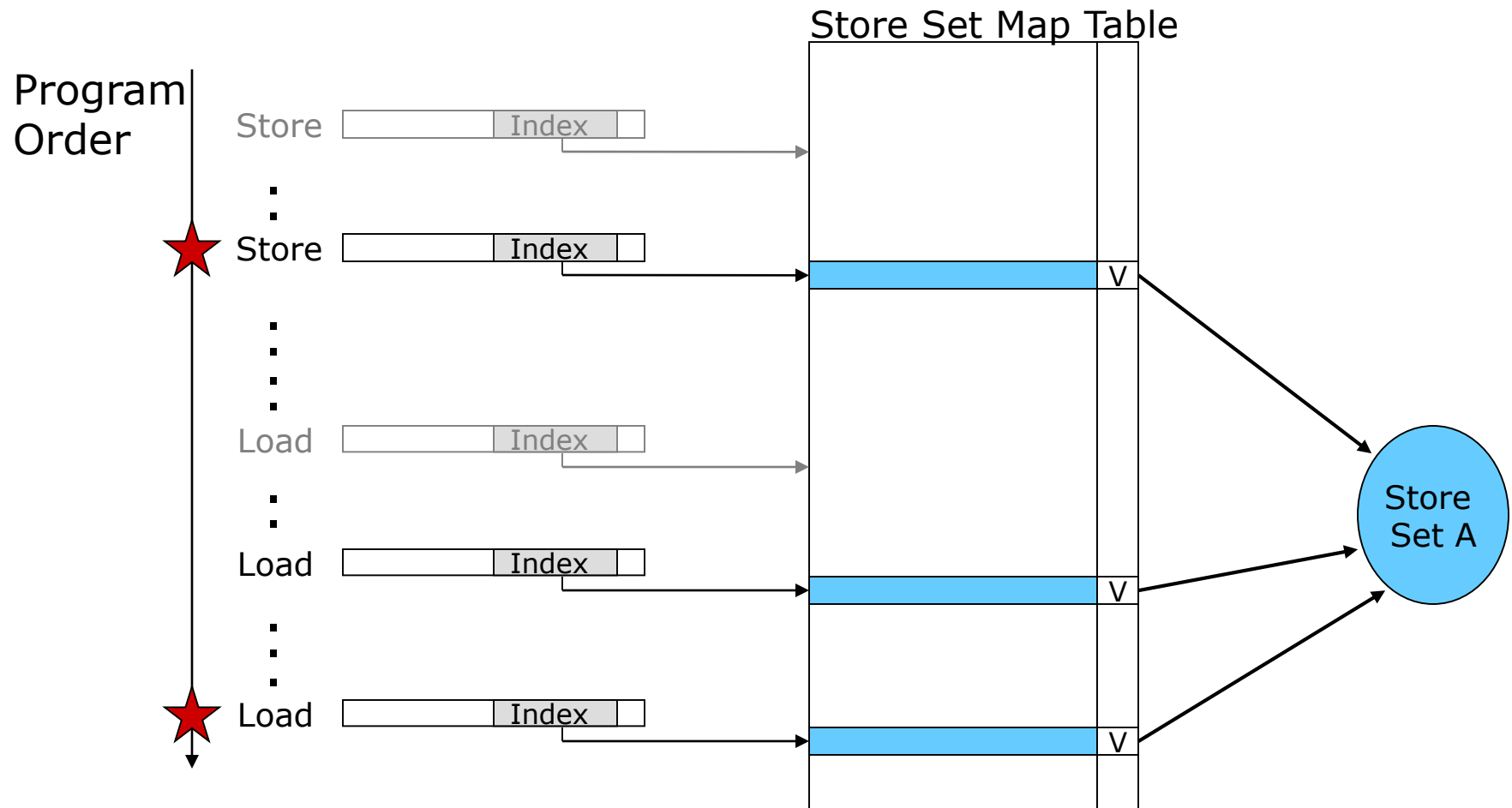
- A load must wait for any stores in its *store set* that have not yet executed
- The processor approximates each load's *store set* by initially allowing naïve speculation and recording memory-order violations

# The Store Set Map Table



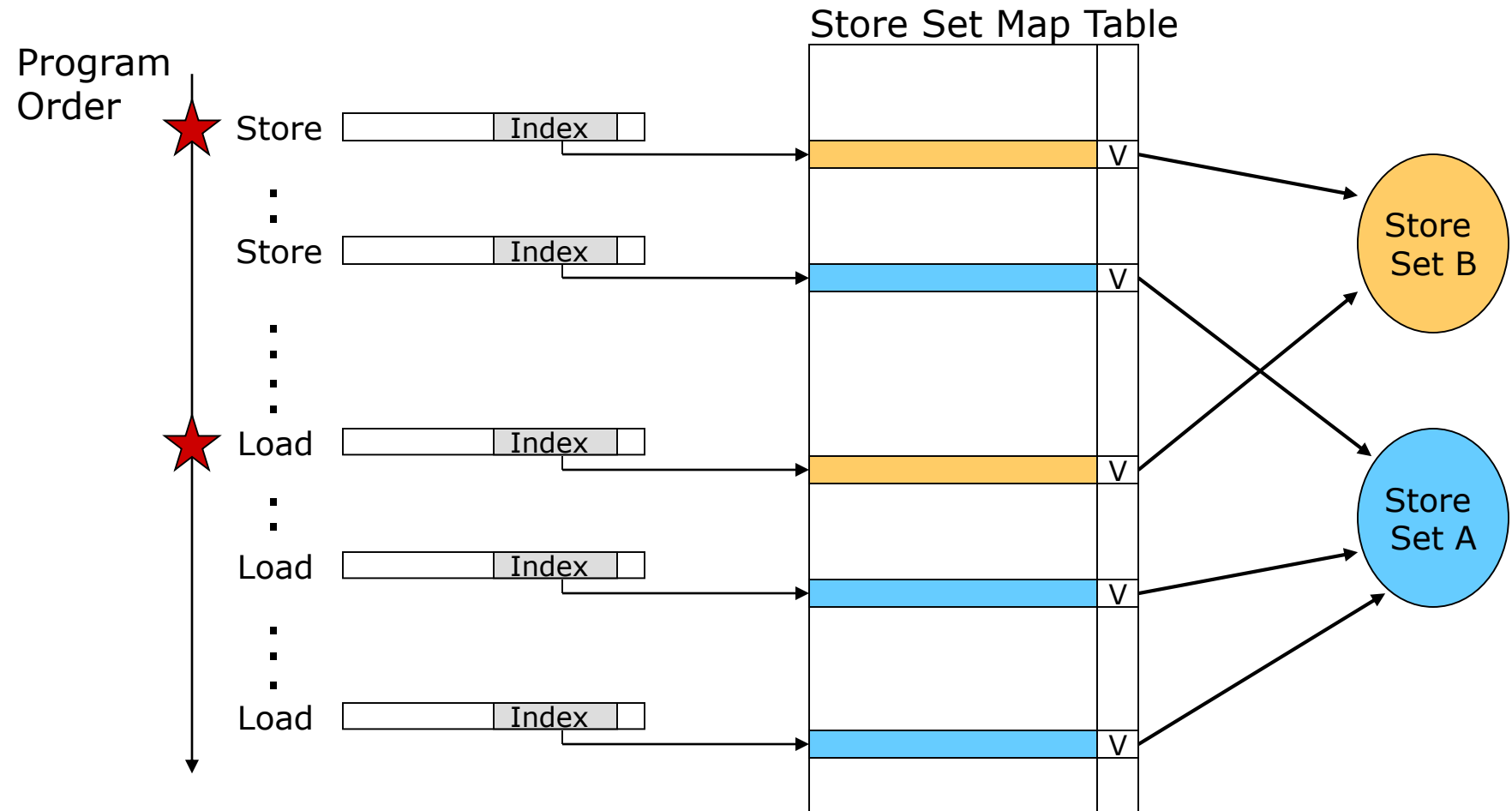
★ - Store/Load Pair causing Memory Order Violation

# Store Set Sharing for Multiple Readers



★ - Store/Load Pair causing Memory Order Violation

# Store Set Map Table, cont.



★ - Store/Load Pair causing Memory Order Violation



# Prefetching

---

- Execution of a load 'depends' on the data it needs being in the cache...

# Prefetching

---

- Execution of a load 'depends' on the data it needs being in the cache...
- Speculate on future instruction and data accesses and fetch them into cache(s)
  - Instruction accesses easier to predict than data accesses

# Prefetching

---

- Execution of a load 'depends' on the data it needs being in the cache...
- Speculate on future instruction and data accesses and fetch them into cache(s)
  - Instruction accesses easier to predict than data accesses
- Varieties of prefetching
  - Hardware prefetching
  - Software prefetching
  - Mixed schemes

# Prefetching

---

- Execution of a load 'depends' on the data it needs being in the cache...
- Speculate on future instruction and data accesses and fetch them into cache(s)
  - Instruction accesses easier to predict than data accesses
- Varieties of prefetching
  - Hardware prefetching
  - Software prefetching
  - Mixed schemes
- *How does prefetching affect cache misses?*

# Prefetching

---

- Execution of a load 'depends' on the data it needs being in the cache...
- Speculate on future instruction and data accesses and fetch them into cache(s)
  - Instruction accesses easier to predict than data accesses
- Varieties of prefetching
  - Hardware prefetching
  - Software prefetching
  - Mixed schemes
- *How does prefetching affect cache misses?*

Compulsory

# Prefetching

---

- Execution of a load 'depends' on the data it needs being in the cache...
- Speculate on future instruction and data accesses and fetch them into cache(s)
  - Instruction accesses easier to predict than data accesses
- Varieties of prefetching
  - Hardware prefetching
  - Software prefetching
  - Mixed schemes
- *How does prefetching affect cache misses?*

Compulsory

Reduce

# Prefetching

---

- Execution of a load 'depends' on the data it needs being in the cache...
- Speculate on future instruction and data accesses and fetch them into cache(s)
  - Instruction accesses easier to predict than data accesses
- Varieties of prefetching
  - Hardware prefetching
  - Software prefetching
  - Mixed schemes
- *How does prefetching affect cache misses?*

Compulsory

Conflict

Reduce

# Prefetching

---

- Execution of a load 'depends' on the data it needs being in the cache...
- Speculate on future instruction and data accesses and fetch them into cache(s)
  - Instruction accesses easier to predict than data accesses
- Varieties of prefetching
  - Hardware prefetching
  - Software prefetching
  - Mixed schemes
- *How does prefetching affect cache misses?*

Compulsory

Reduce

Conflict

Increase



# Prefetching

---

- Execution of a load 'depends' on the data it needs being in the cache...
- Speculate on future instruction and data accesses and fetch them into cache(s)
  - Instruction accesses easier to predict than data accesses
- Varieties of prefetching
  - Hardware prefetching
  - Software prefetching
  - Mixed schemes
- *How does prefetching affect cache misses?*

Compulsory

Reduce

Conflict

Increase

Capacity

# Prefetching

---

- Execution of a load 'depends' on the data it needs being in the cache...
- Speculate on future instruction and data accesses and fetch them into cache(s)
  - Instruction accesses easier to predict than data accesses
- Varieties of prefetching
  - Hardware prefetching
  - Software prefetching
  - Mixed schemes
- *How does prefetching affect cache misses?*

Compulsory

Reduce

Conflict

Increase

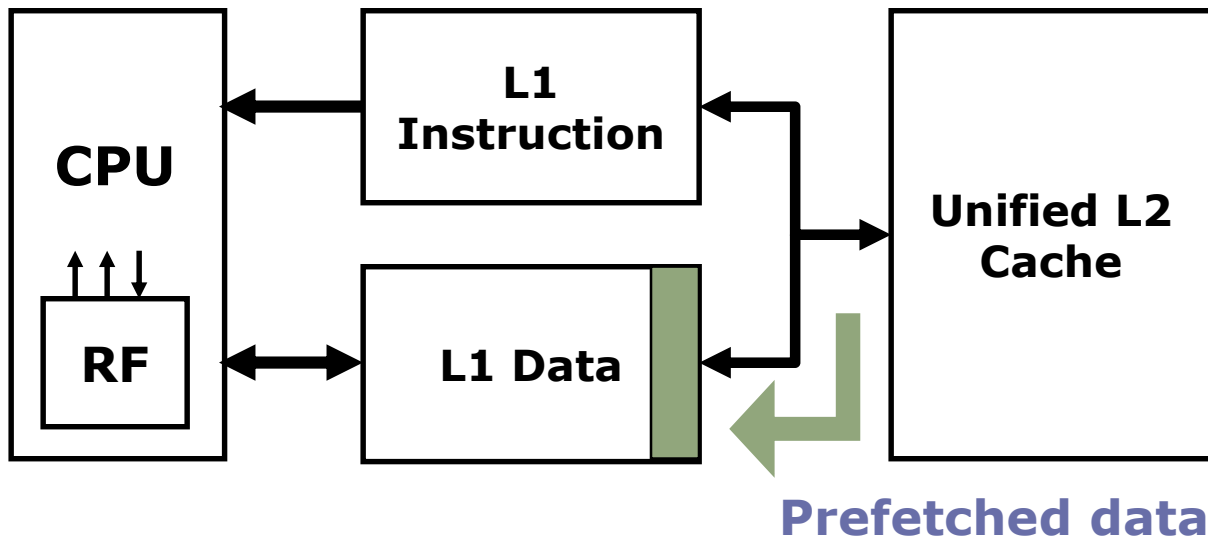
Capacity

Increase

# Issues in Prefetching

---

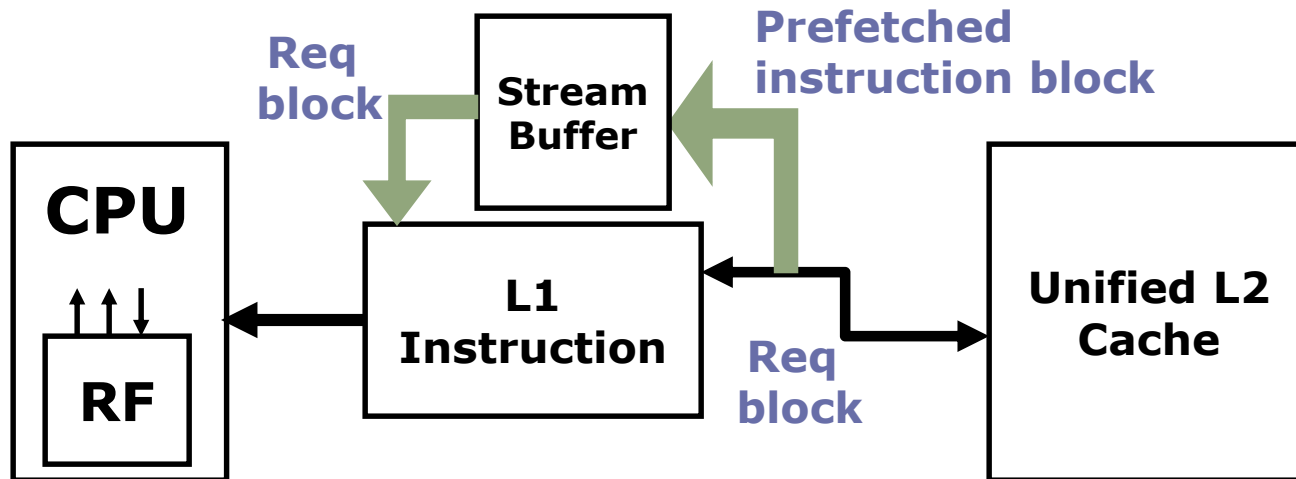
- Usefulness – should produce hits
- Timeliness – not late and not too early
- Cache and bandwidth pollution



# Hardware Instruction Prefetching

## Instruction prefetch in Alpha AXP 21064

- Fetch two blocks on a miss; the requested block (i) and the next consecutive block (i+1)
- Requested block placed in cache, and next block in instruction stream buffer
- If miss in cache but hit in stream buffer, move stream buffer block into cache and prefetch next block (i+2)



# Hardware Data Prefetching

---

- Prefetch-on-miss:
  - Prefetch  $b + 1$  upon miss on  $b$
- One Block Lookahead (OBL) scheme
  - Initiate prefetch for block  $b + 1$  when block  $b$  is accessed
  - *Why is this different from doubling block size?*
  - Can extend to  $N$ -block lookahead (called *stream prefetching*)
- Strided prefetch
  - If observe sequence of accesses to block  $b, b+N, b+2N,$  then prefetch  $b+3N$  etc.

**Example:** IBM Power 5 [2003] supports eight independent streams of strided prefetch per processor, prefetching 12 lines ahead of current access

*Thank you!*

*Next lecture:  
Cache Coherence*