Advanced Memory Operations
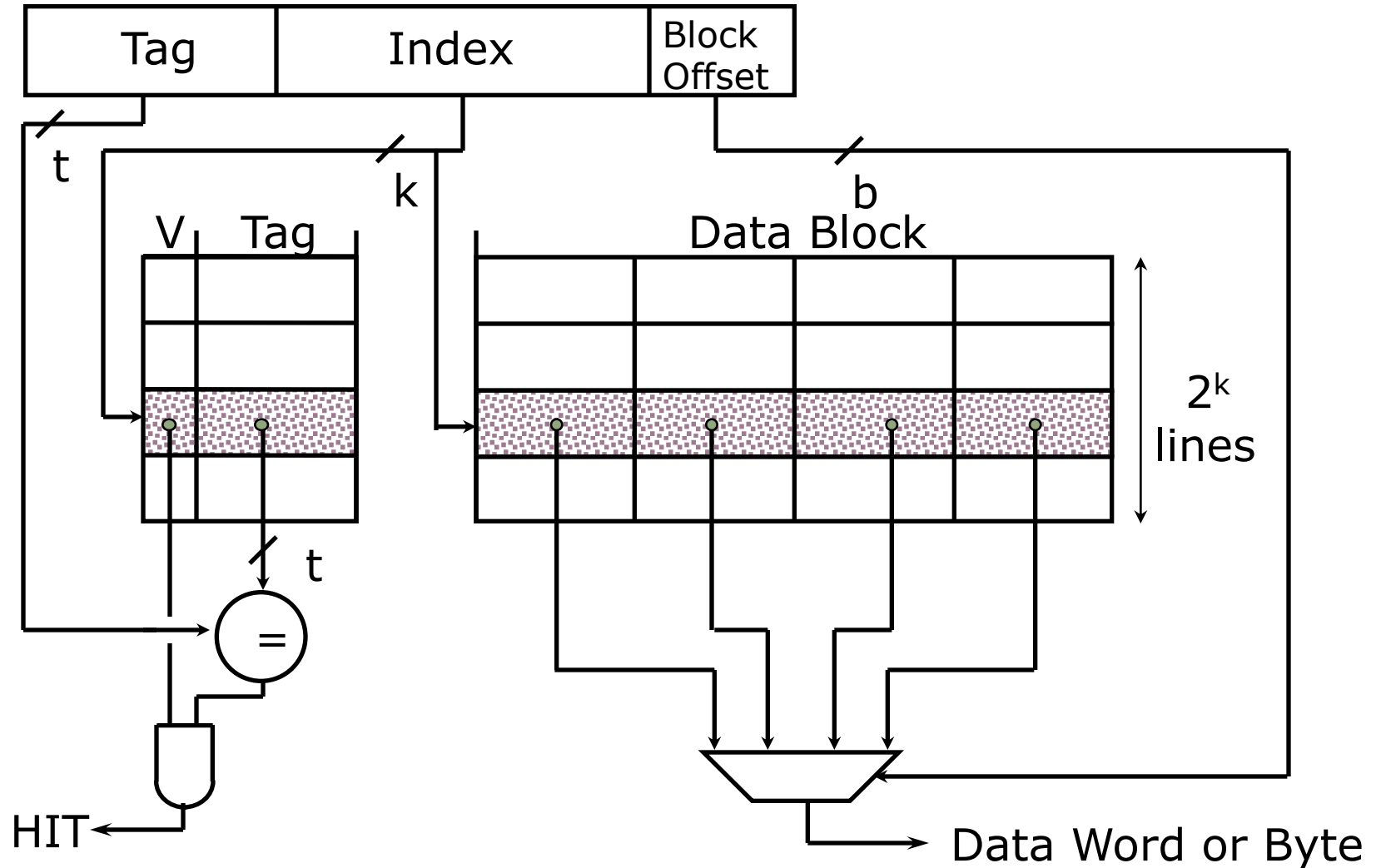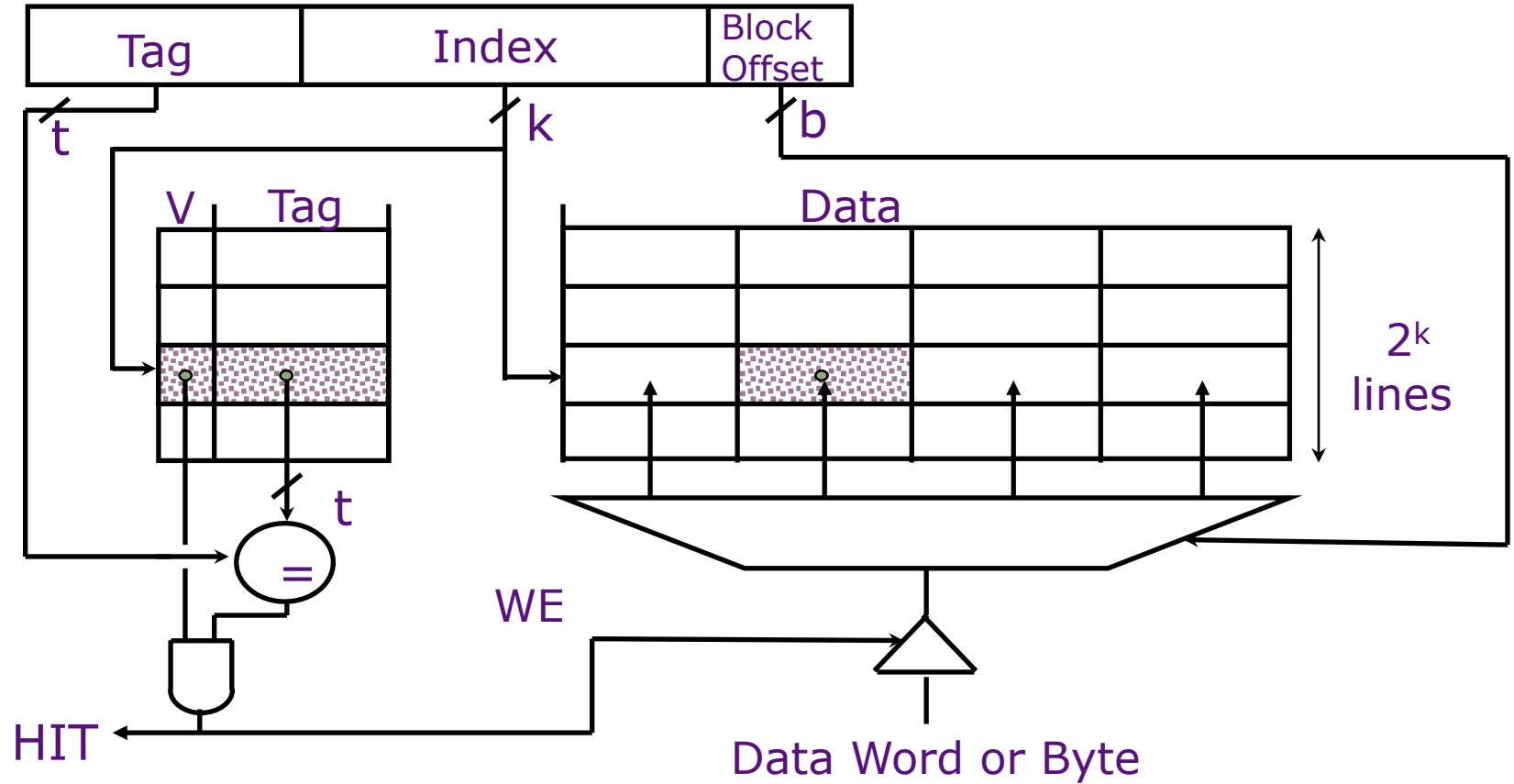
Daniel Sanchez
Computer Science and Artificial Intelligence Laboratory
M.I.T.

Reminder: Direct-Mapped Cache

Tag	Index	Block Offset

t

k

b

V   Tag

Data Block

2^k lines

t

=

HIT

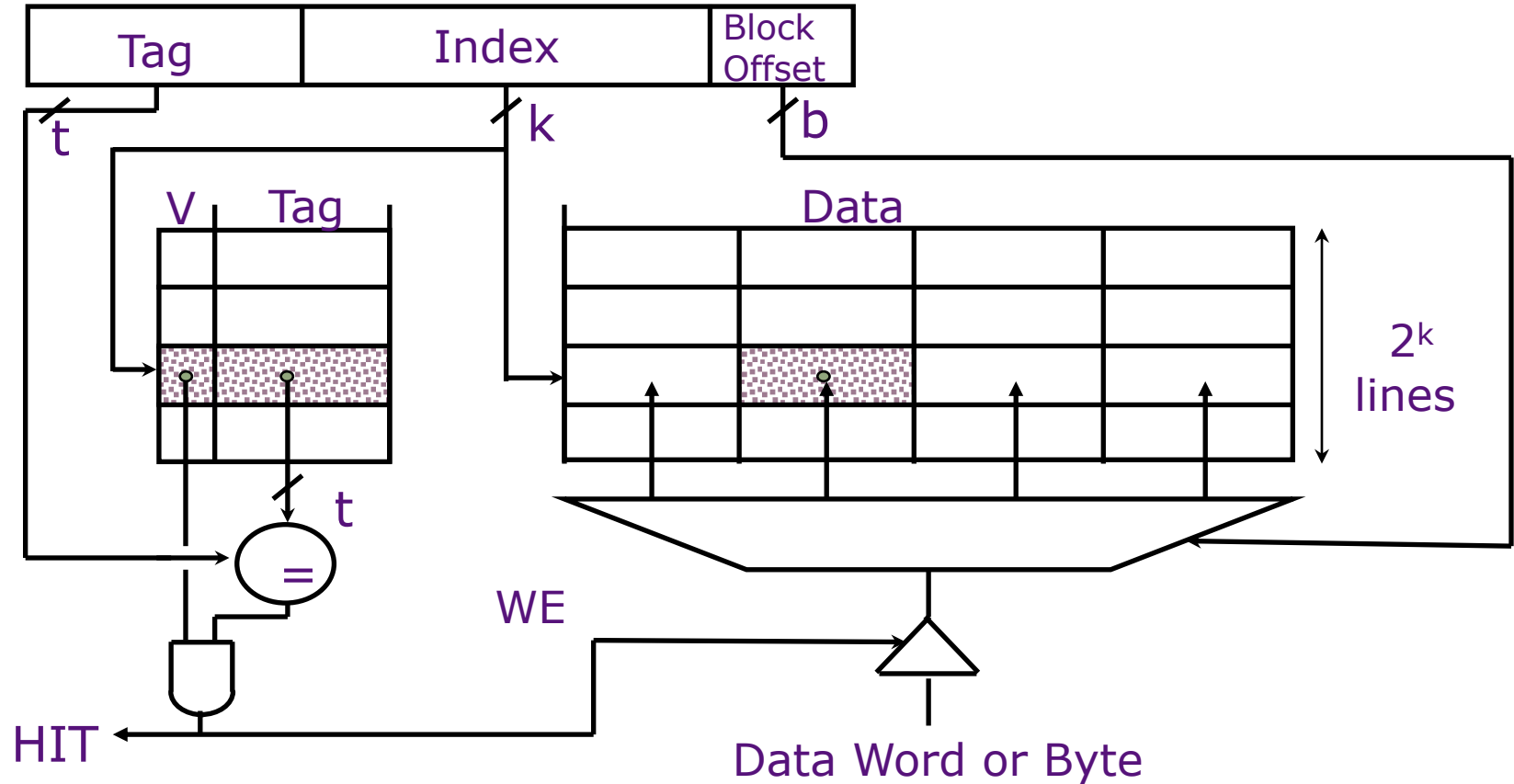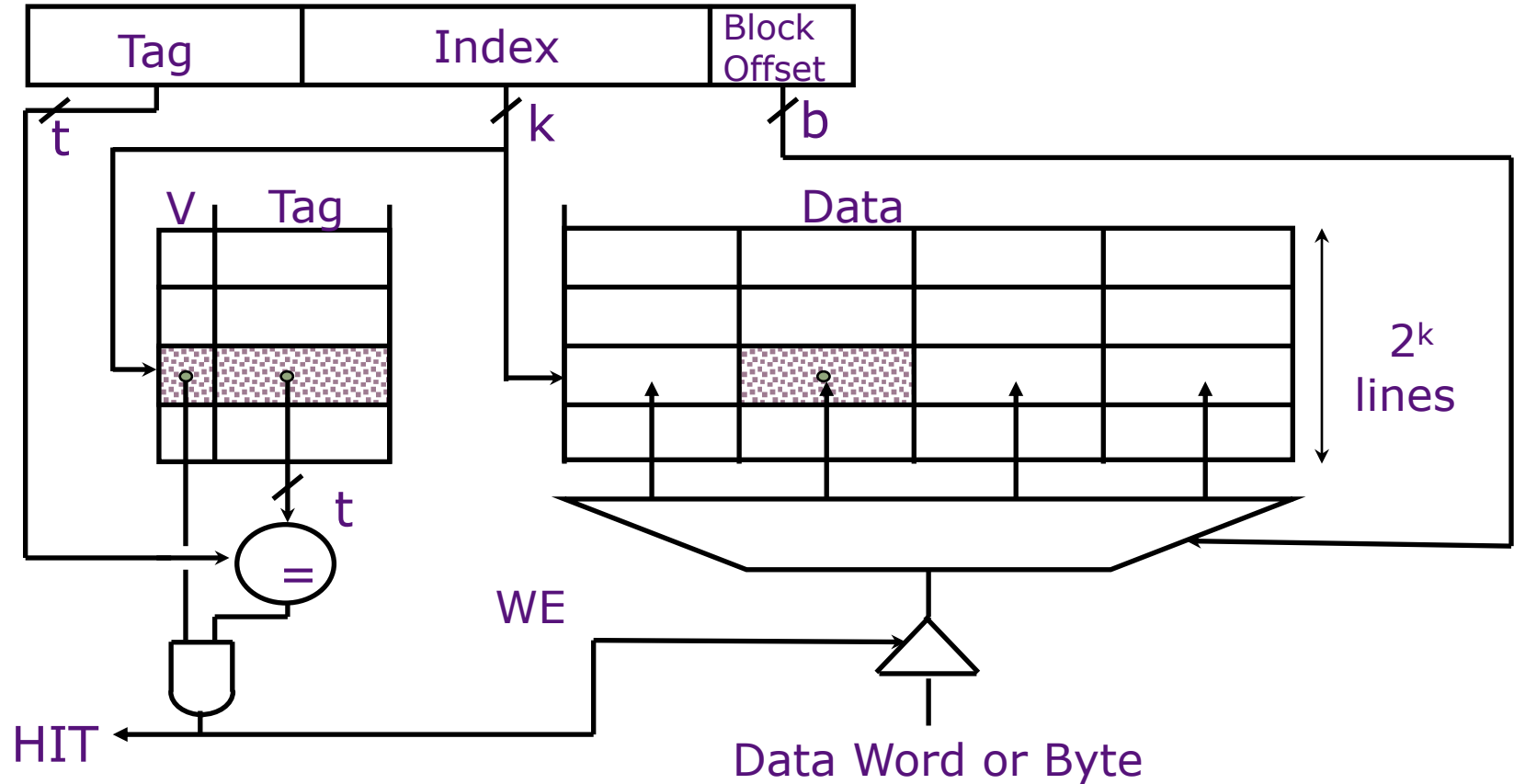Data Word or Byte

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:

- Stall – delivering data as fast as possible:
  - Circuit-level techniques (CAM tags) may allow one-cycle writes

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:

- Stall – delivering data as fast as possible:
    - Circuit-level techniques (CAM tags) may allow one-cycle writes

- 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:
- Stall – delivering data as fast as possible:
  - Circuit-level techniques (CAM tags) may allow one-cycle writes

- 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

Time →

LD_0

ST_1

ST_2

LD_3

ST_4

LD_5

ST_6 miss

Tag						
Data						

Buffer						

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

Time →

LD_0

ST_1

ST_2

LD_3

ST_4

LD_5

ST_6 miss

Tag						
Data						

Buffer						

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

Time ⟶

LD$_0$

ST$_1$

ST$_2$

Tag						
Data						

LD$_3$

ST$_4$

LD$_5$

Buffer						

ST$_6$ miss

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

Time →

LD_0

ST_1

ST_2

LD_3

ST_4

LD_5

ST_6 miss

Tag	LD_0					
Data						

Buffer						

Pipelined/Delayed Write Timing

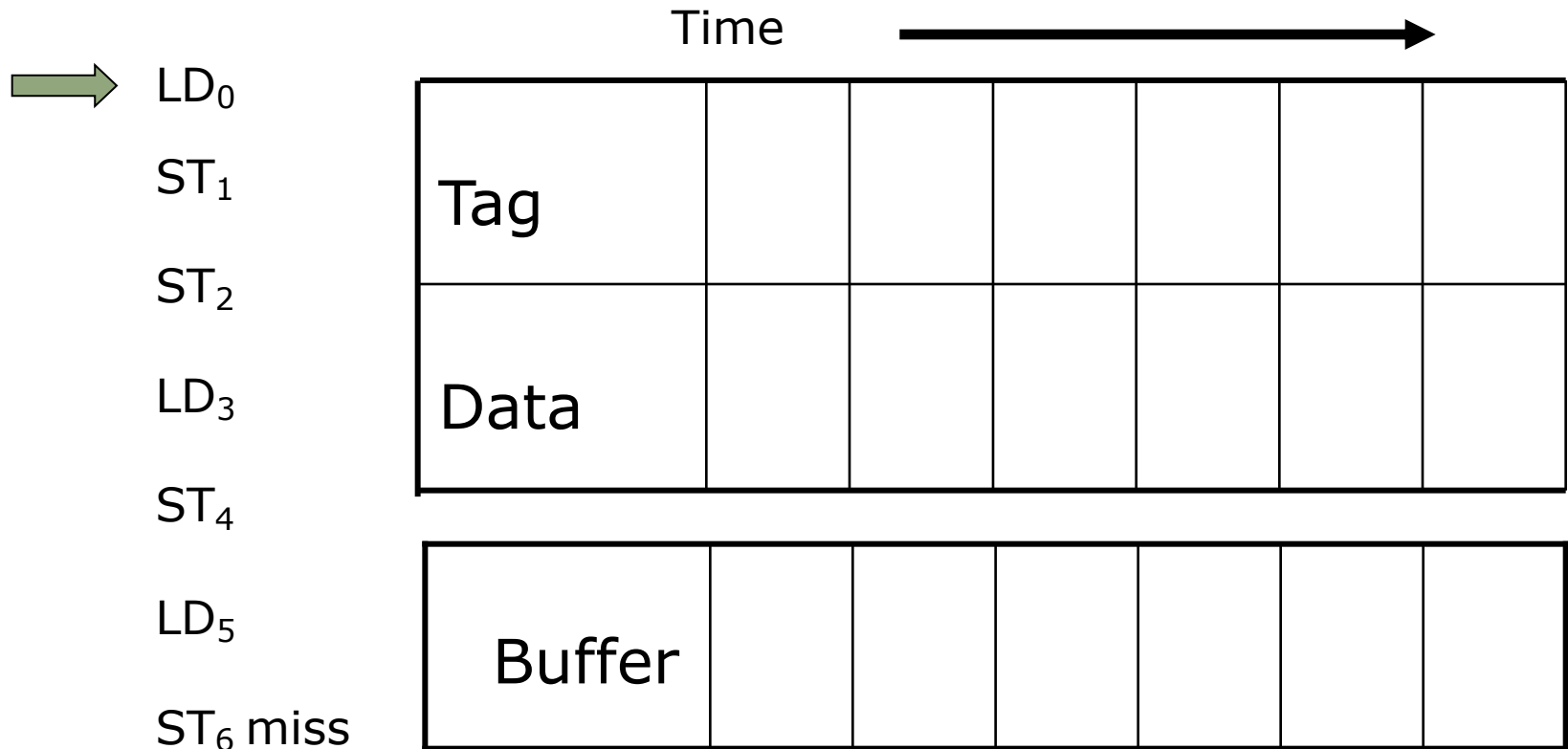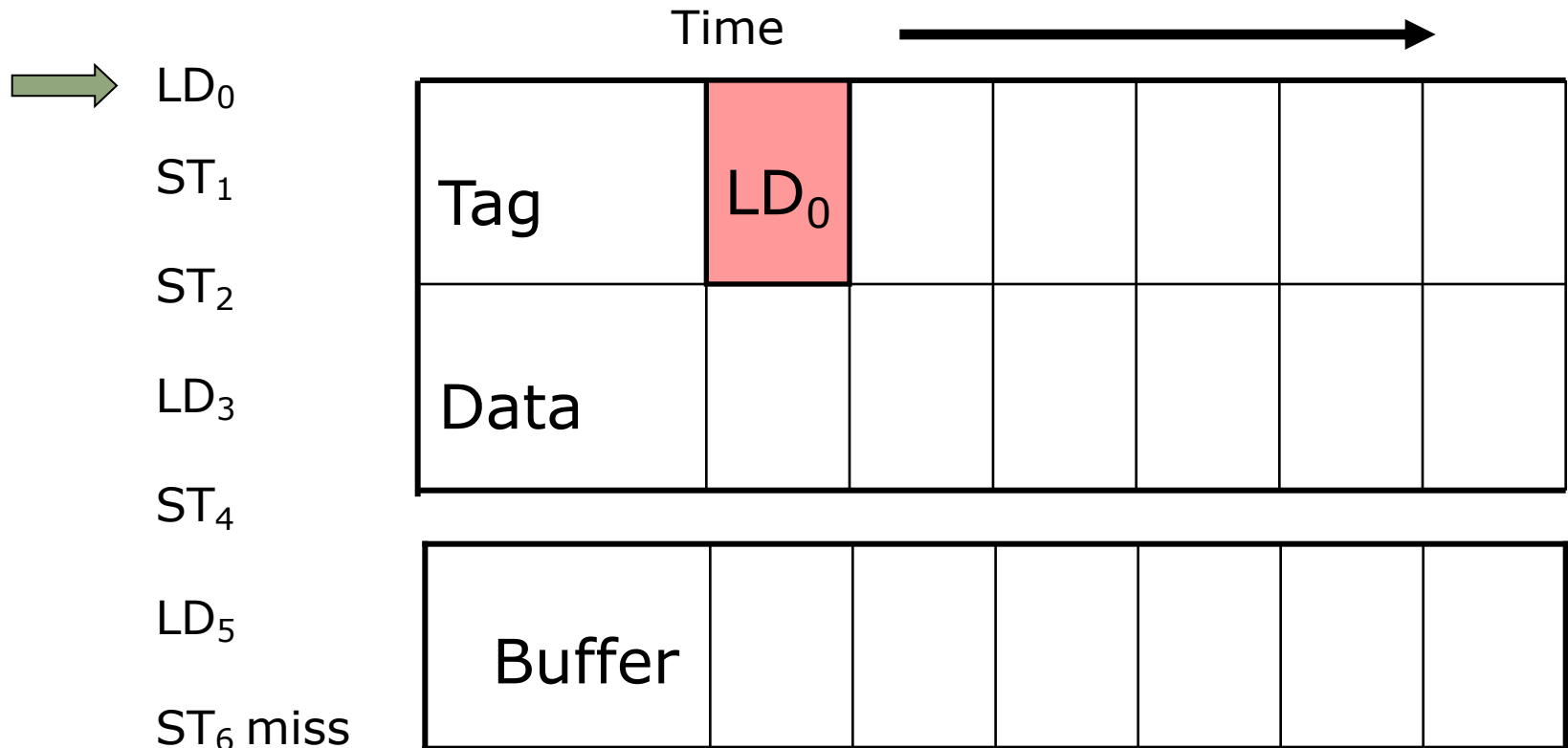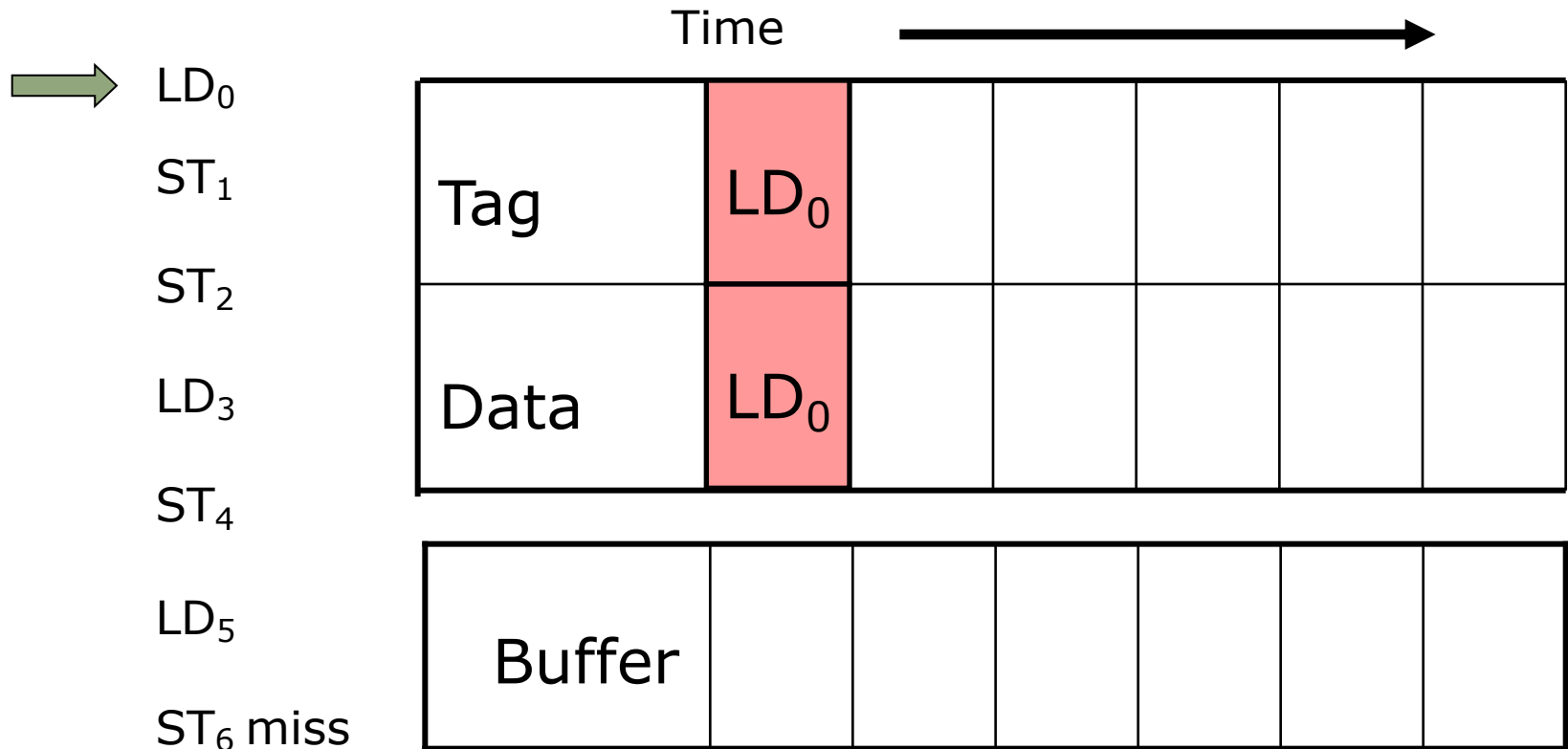Problem: Need to commit lazily saved write data

Solution: Write data during idle data cycle of next store's tag check
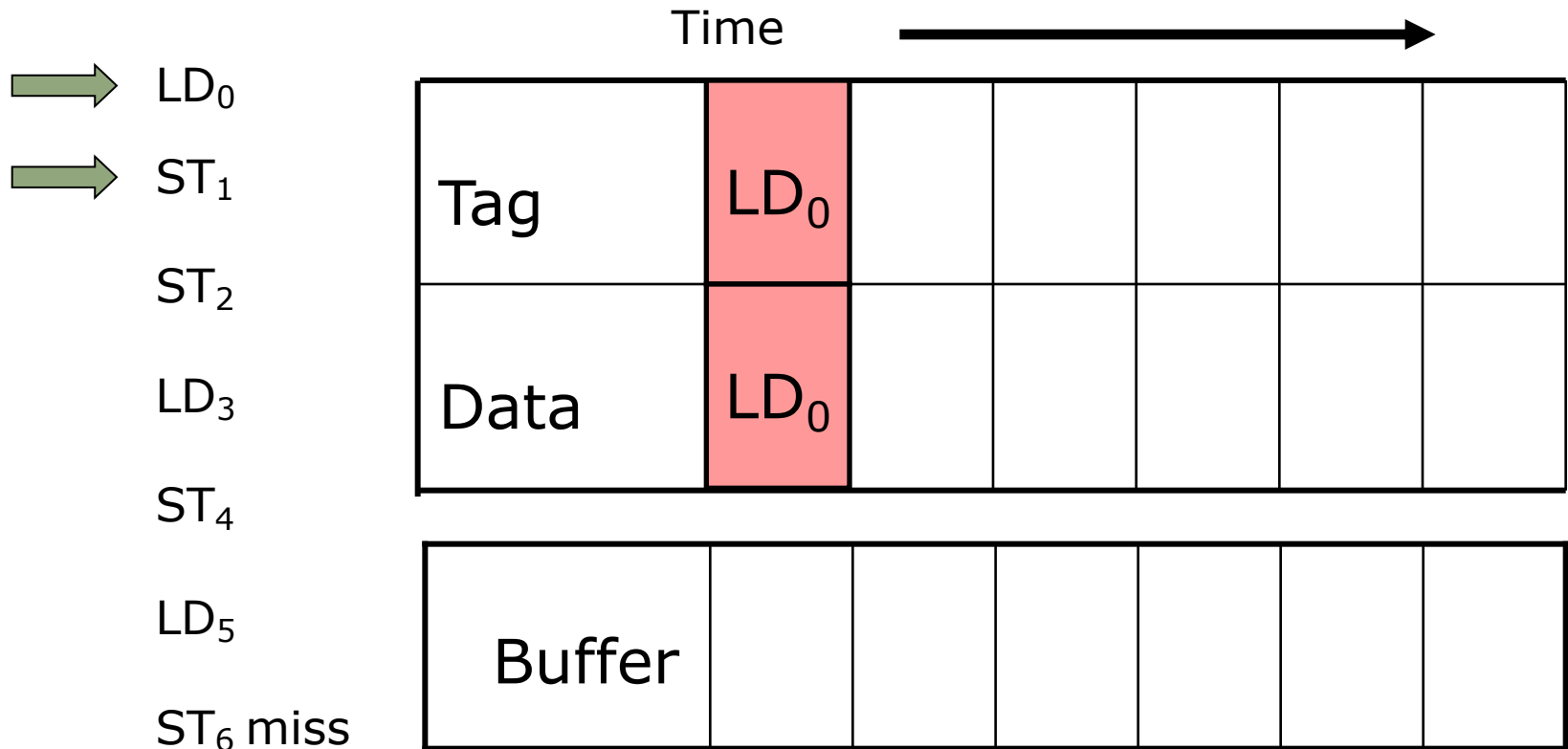
Pipelined/Delayed Write Timing

Problem: Need to commit lazily saved write data

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

Time →

LD$_0$

ST$_1$

ST$_2$

Tag	LD$_0$					
Data	LD$_0$					

LD$_3$

ST$_4$

Buffer						

LD$_5$

ST$_6$ miss

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

Time →

LD$_0$

ST$_1$

ST$_2$

LD$_3$

ST$_4$

LD$_5$

ST$_6$ miss

Tag		LD$_0$	ST$_1$				
Data		LD$_0$					

Buffer							

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

Time →

LD_0

ST_1

ST_2

LD_3

ST_4

LD_5

ST_6 miss

Tag		LD_0	ST_1				
Data		LD_0					

Buffer			ST_1				

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

Time →

Tag		LD_0	ST_1				

Data		LD_0					

Buffer			ST_1				

LD_0

ST_1

ST_2

LD_3

ST_4

LD_5

ST_6 miss

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

Time →

LD_0

ST_1

ST_2

LD_3

ST_4

LD_5

ST_6 miss

Tag		LD_0	ST_1	ST_2			
Data		LD_0					

Buffer			ST_1				

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

Time →

LD_0

ST_1

ST_2

LD_3

ST_4

LD_5

ST_6 miss

Tag			LD_0	ST_1	ST_2			
Data			LD_0		ST_1			

Buffer			ST_1				

Pipelined/Delayed Write Timing

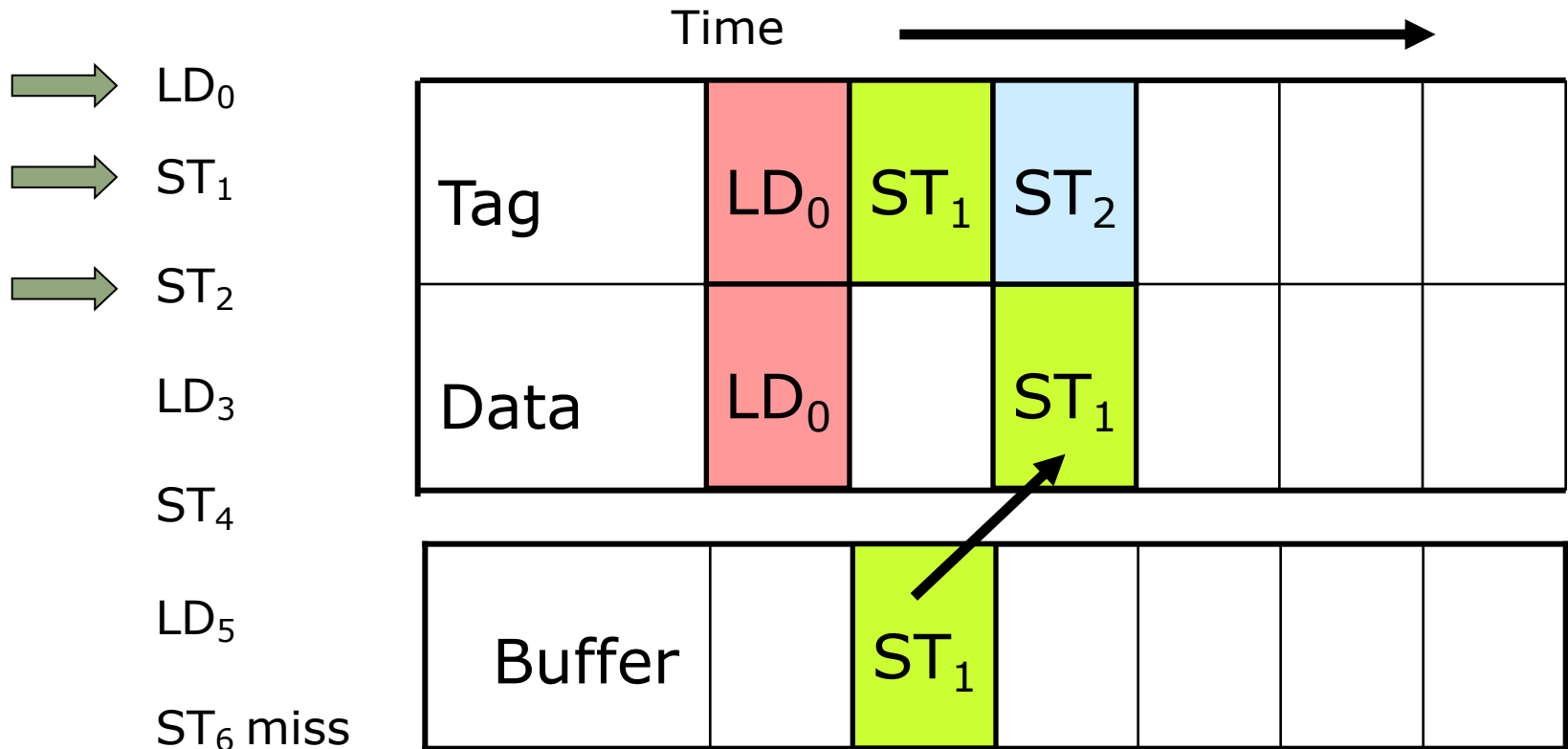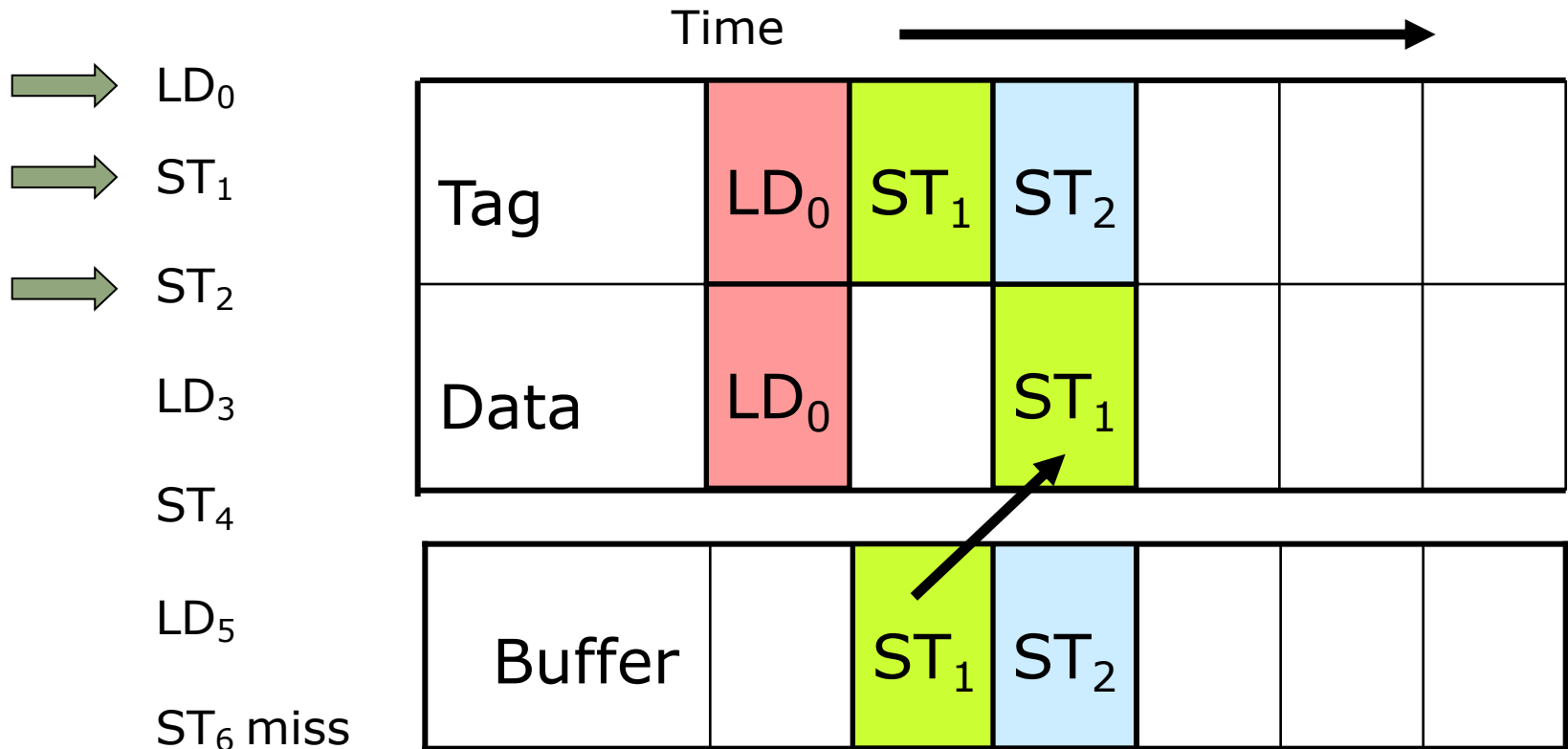Problem: Need to commit lazily saved write data

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

Time →

LD$_0$
ST$_1$
ST$_2$
LD$_3$
ST$_4$
LD$_5$
ST$_6$ miss

		LD$_0$	ST$_1$	ST$_2$			
Tag		LD$_0$	ST$_1$	ST$_2$			
Data		LD$_0$		ST$_1$			

			ST$_1$	ST$_2$			
Buffer			ST$_1$	ST$_2$			

Pipelined/Delayed Write Timing

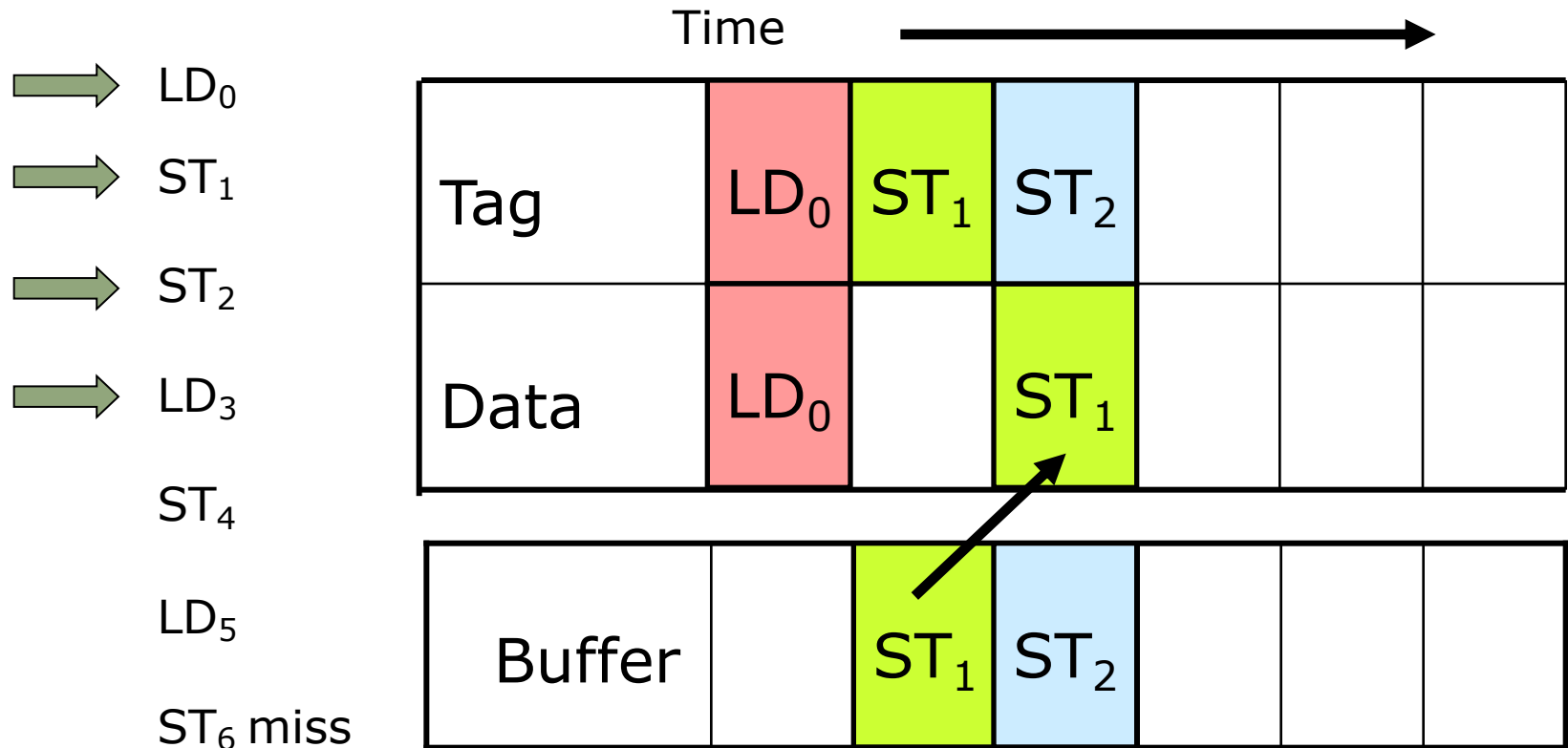Problem: Need to commit lazily saved write data

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

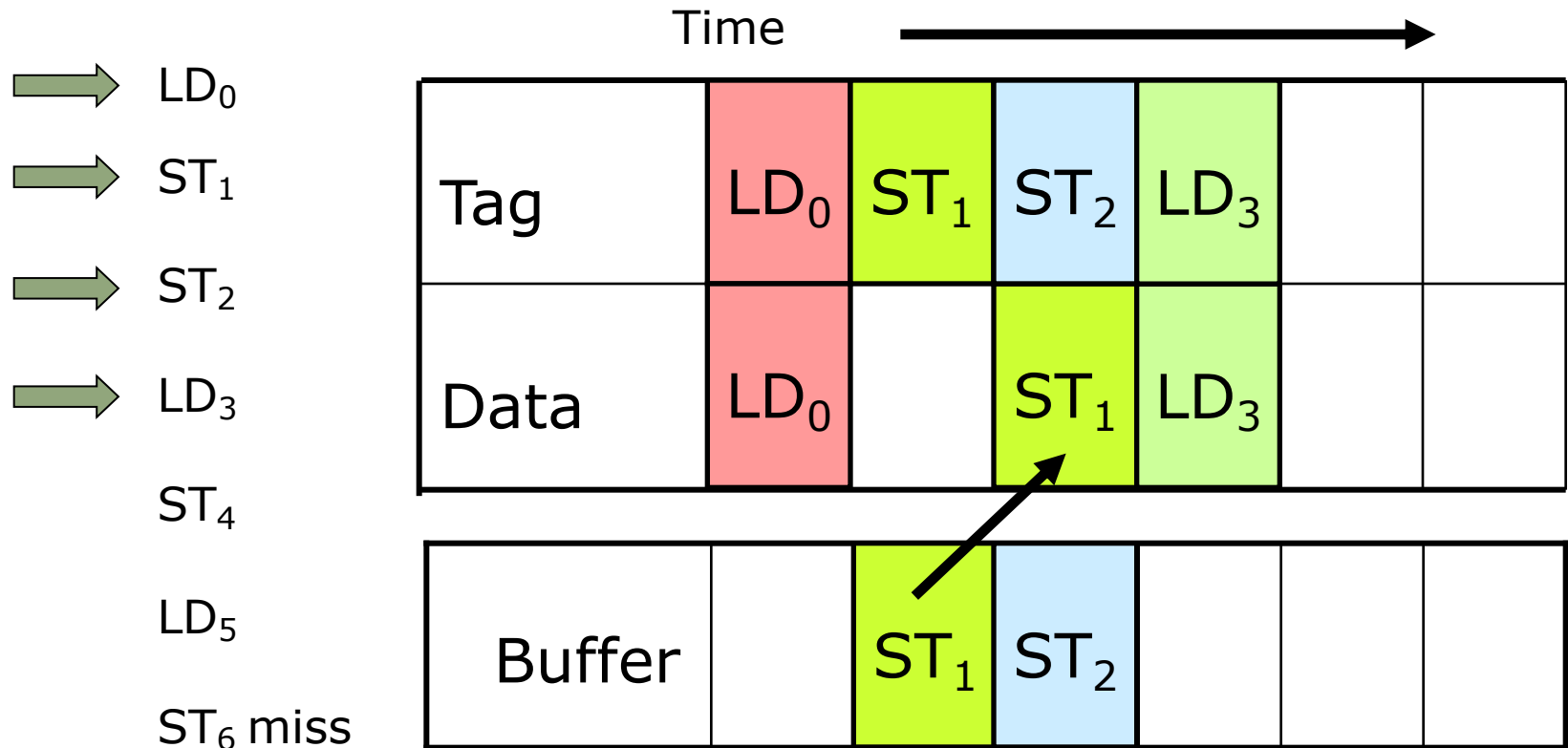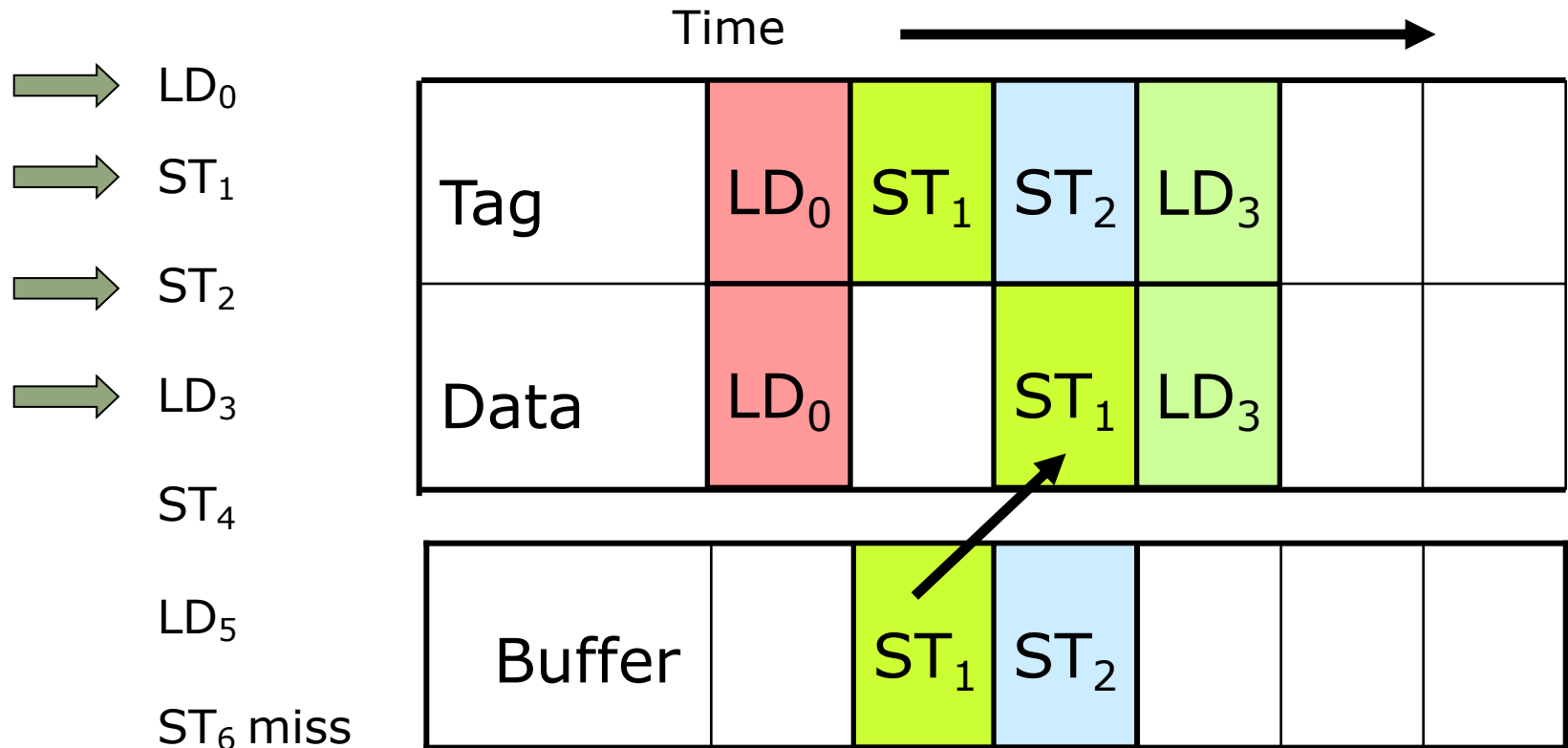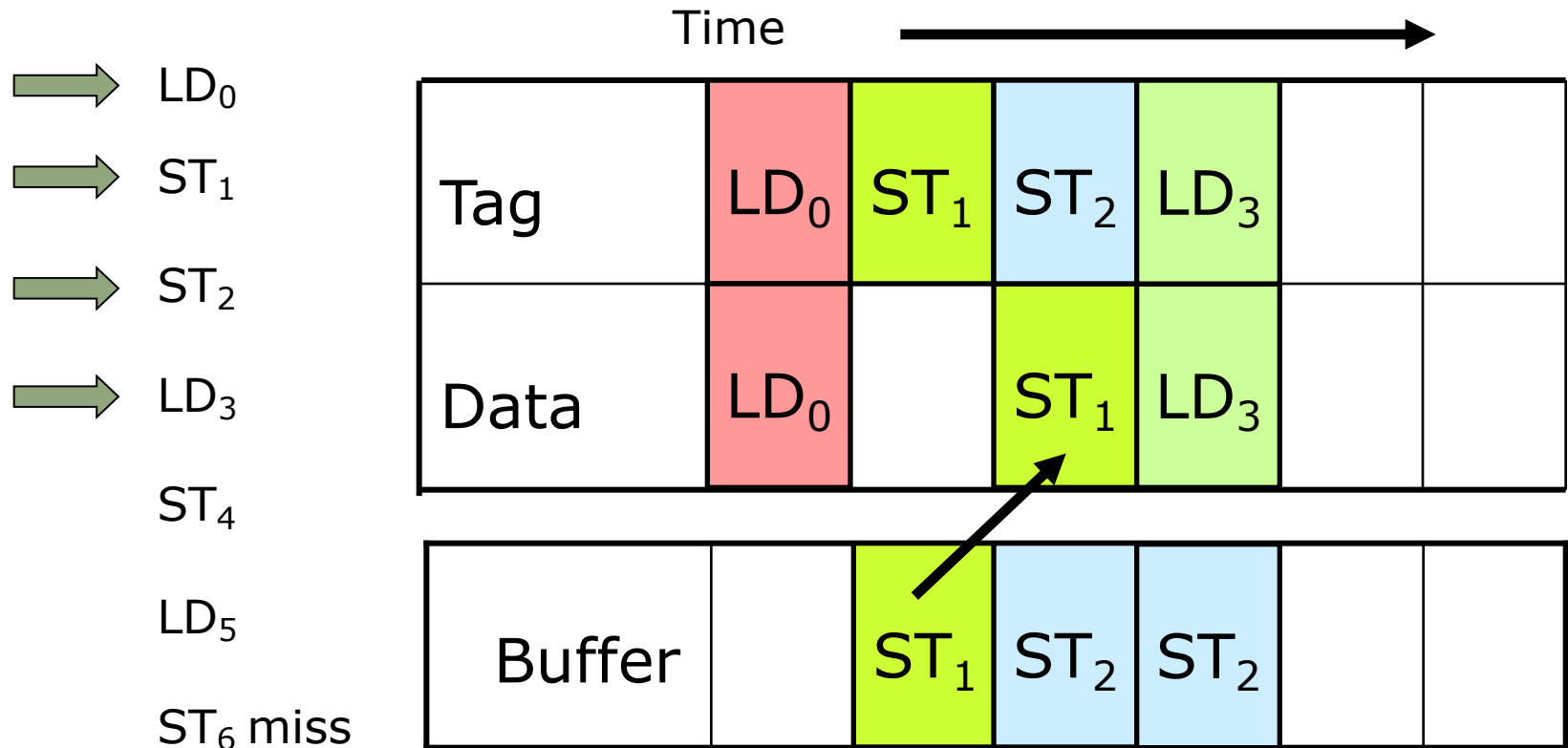Problem: Need to commit lazily saved write data

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

Time →

LD_0

ST_1

ST_2

LD_3

ST_4

LD_5

ST_6 miss

Tag		LD_0	ST_1	ST_2	LD_3		
Data		LD_0		ST_1	LD_3		

Buffer			ST_1	ST_2	ST_2		

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

Time →

LD_0

ST_1

ST_2

LD_3

ST_4

LD_5

ST_6 miss

Tag		LD_0	ST_1	ST_2	LD_3	ST_4	
Data		LD_0		ST_1	LD_3	ST_2	

Buffer			ST_1	ST_2	ST_2	ST_4	

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

Time →

LD₀

ST₁

ST₂

LD₃

ST₄

LD₅

ST₆ miss

		LD₀	ST₁	ST₂	LD₃	ST₄	
Tag		LD₀	ST₁	ST₂	LD₃	ST₄	
Data		LD₀		ST₁	LD₃	ST₂	
Buffer			ST₁	ST₂	ST₂	ST₄	

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

Time →

Tag		LD_0	ST_1	ST_2	LD_3	ST_4	LD_5
Data		LD_0		ST_1	LD_3	ST_2	LD_5
Buffer			ST_1	ST_2	ST_2	ST_4	ST_4

→ LD_0

→ ST_1

→ ST_2

→ LD_3

→ ST_4

→ LD_5

ST_6 miss

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

Time →

LD_0
ST_1
ST_2
LD_3
ST_4
LD_5
ST_6 miss

Tag		LD_0	ST_1	ST_2	LD_3	ST_4	LD_5
Data		LD_0		ST_1	LD_3	ST_2	LD_5
Buffer				ST_1	ST_2	ST_2	ST_4	ST_4

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

Time →

Tag	LD_0	ST_1	ST_2	LD_3	ST_4	LD_5	ST_6
Data	LD_0		ST_1	LD_3	ST_2	LD_5	
Buffer			ST_1	ST_2	ST_2	ST_4	ST_4

⟹ LD_0

⟹ ST_1

⟹ ST_2

⟹ LD_3

⟹ ST_4

⟹ LD_5

⟹ ST_6 miss

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

Time →

		LD_0	ST_1	ST_2	LD_3	ST_4	LD_5	ST_6
Tag		LD_0	ST_1	ST_2	LD_3	ST_4	LD_5	ST_6
Data		LD_0		ST_1	LD_3	ST_2	LD_5	
Buffer			ST_1	ST_2	ST_2	ST_4	ST_4	ST_6

- LD_0
- ST_1
- ST_2
- LD_3
- ST_4
- LD_5
- ST_6 miss

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

Time →

Tag		LD_0	ST_1	ST_2	LD_3	ST_4	LD_5	ST_6
Data		LD_0		ST_1	LD_3	ST_2	LD_5	ST_4
Buffer				ST_1	ST_2	ST_2	ST_4	ST_4	ST_6

⇒ LD_0

⇒ ST_1

⇒ ST_2

⇒ LD_3

⇒ ST_4

⇒ LD_5

⇒ ST_6 miss

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

What is condition for bypass?

Pipelining Cache Writes

What if instruction needs data in delayed write buffer?   Bypass
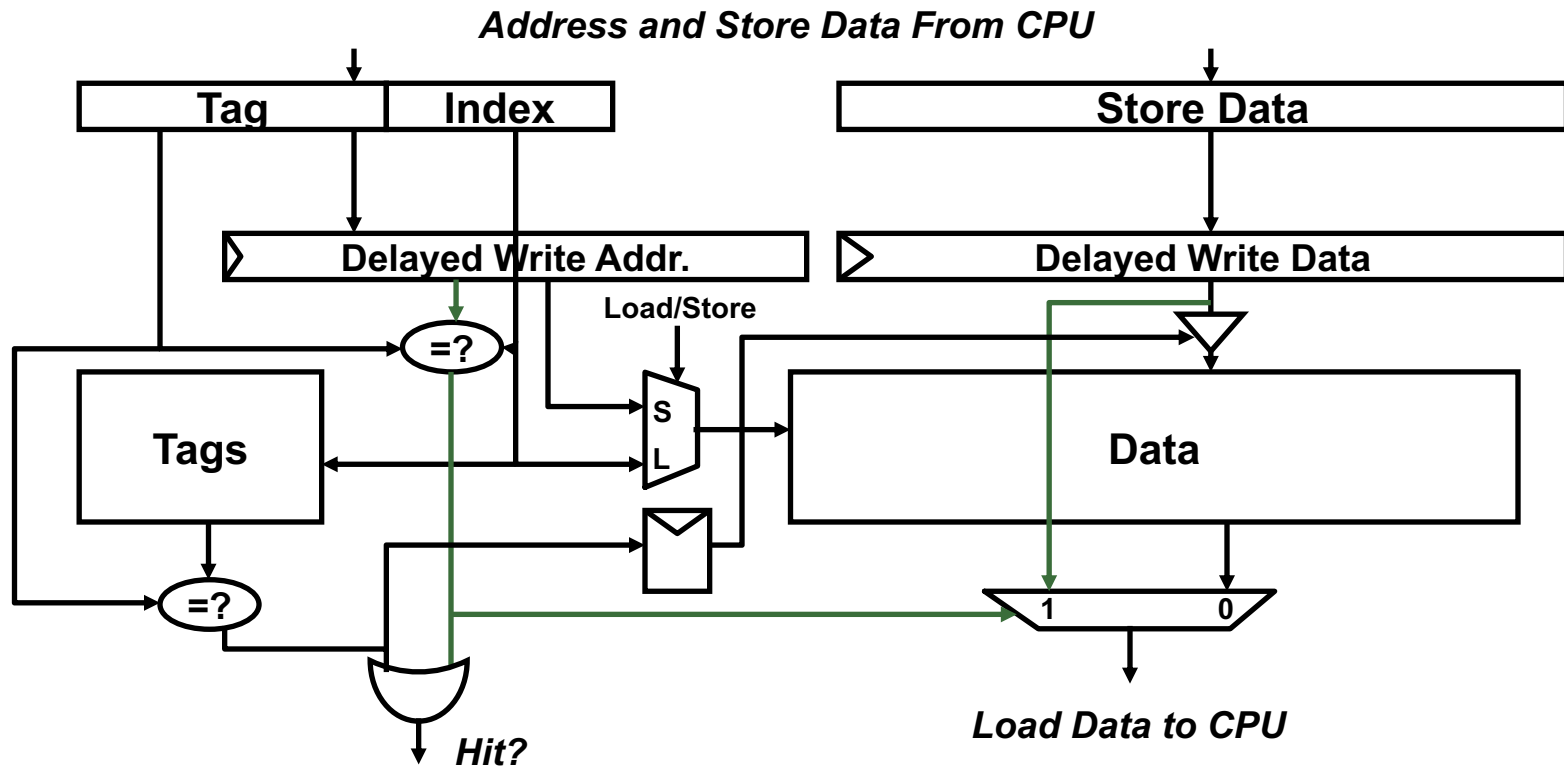
What is condition for bypass?   Tag and index match "delayed write addr"

Pipelining Cache Writes

What if instruction needs data in delayed write buffer?   Bypass

What is condition for bypass?   Tag and index match "delayed write addr"

Address and Store Data From CPU

| Tag | Index | | Store Data |

Delayed Write Addr. Delayed Write Data

Load/Store

=?

Tags

S

L

Data

=?

1   0

Hit?

Load Data to CPU

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
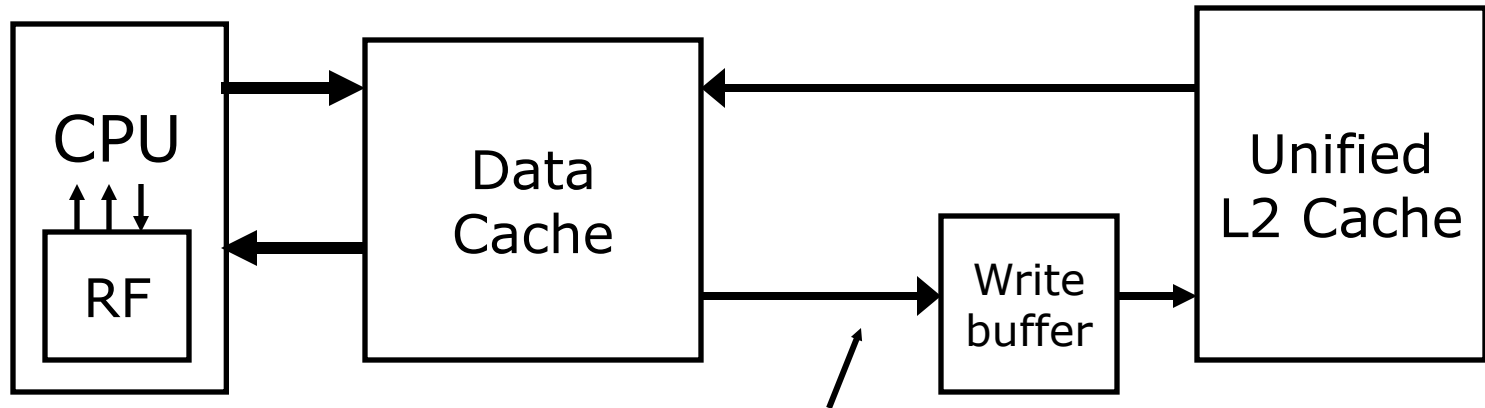
Reducing Read Miss Penalty
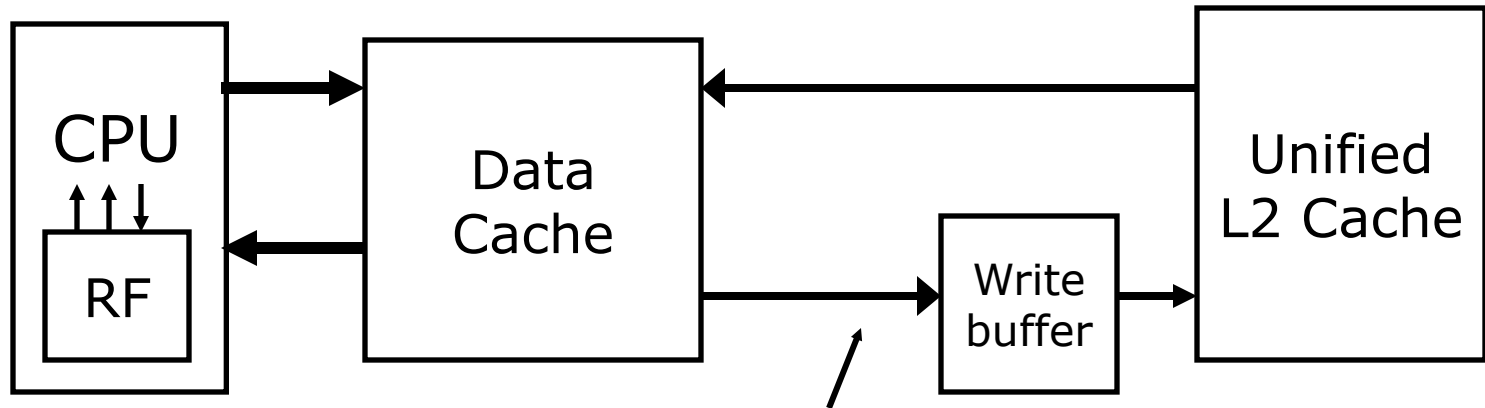
CPU
RF
Data Cache
Write buffer
Unified L2 Cache

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

CPU / RF → Data Cache ← Unified L2 Cache

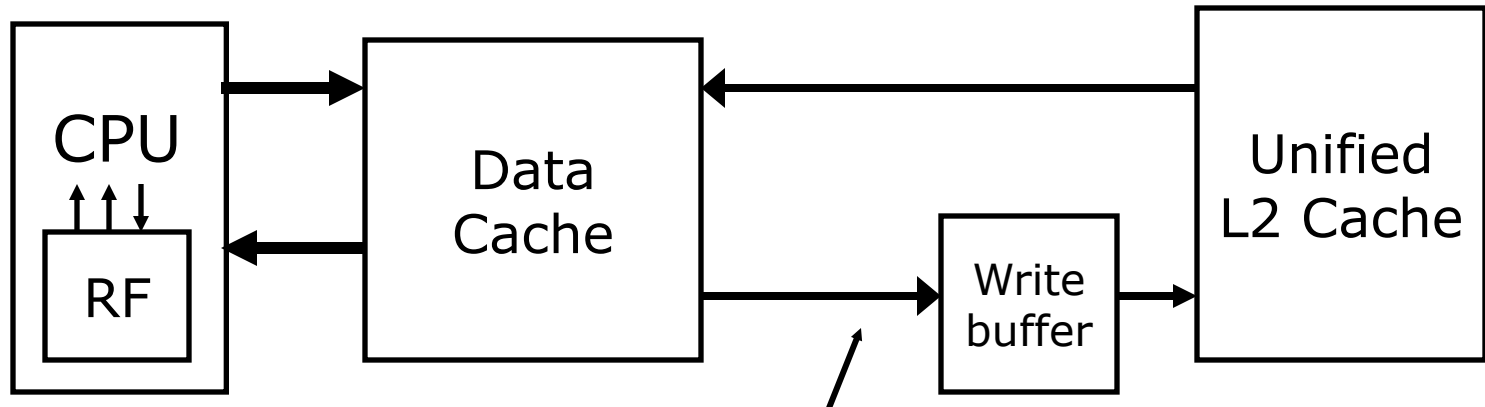Data Cache → Write buffer → Unified L2 Cache

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

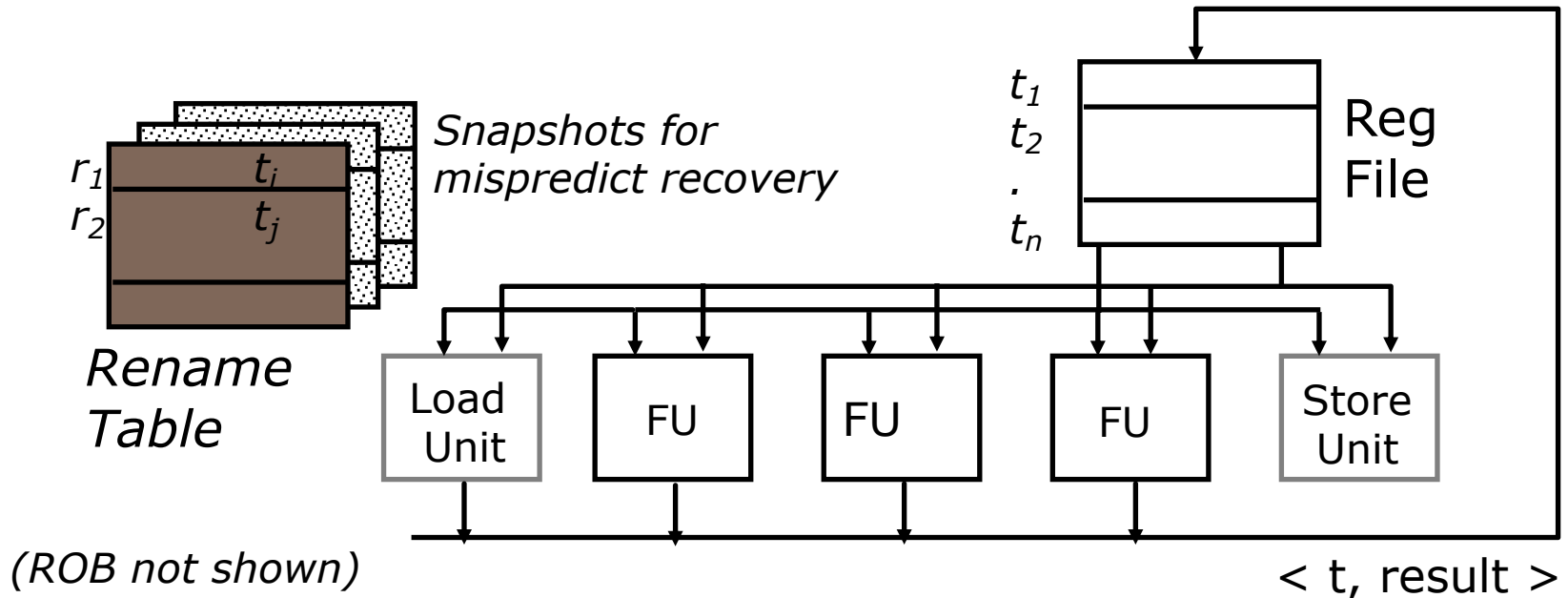Evicted dirty lines for writeback cache
OR
All writes in writethrough cache

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)

Snapshots for mispredict recovery

r_1 t_i
r_2 t_j

Rename Table

t_1
t_2
.
t_n

Reg File

Load Unit   FU   FU   FU   Store Unit

(ROB not shown)

< t, result >

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

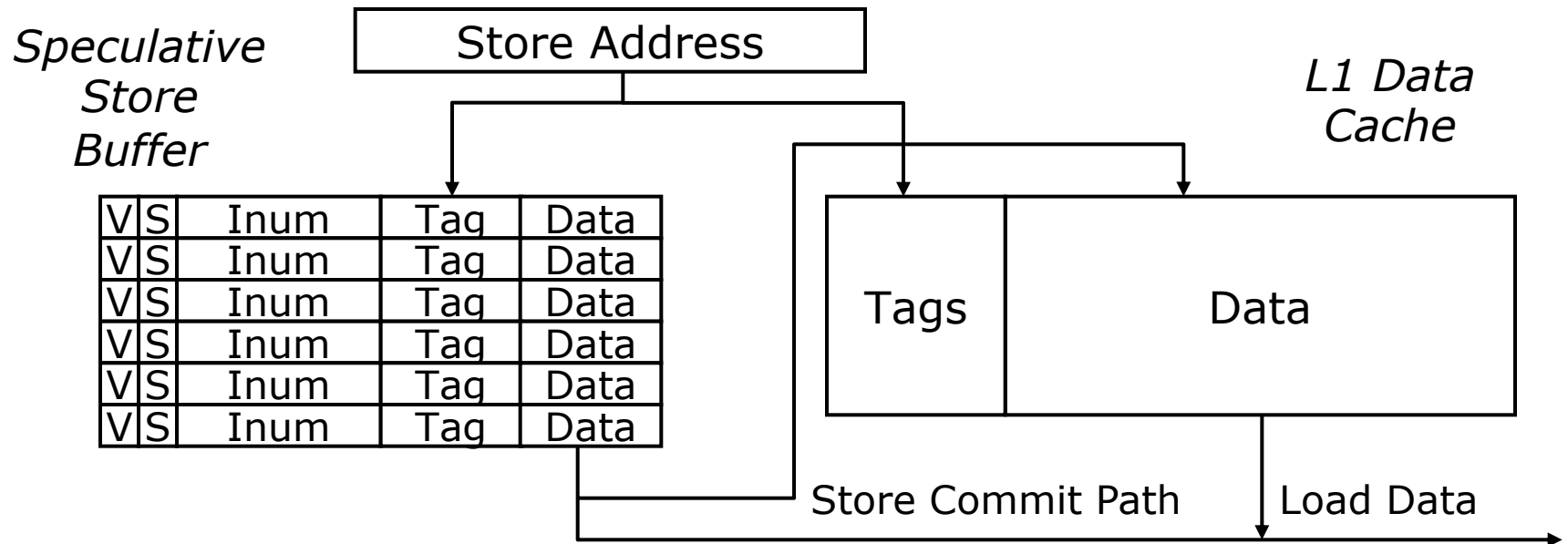*Commits are generally done in order – why?*

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
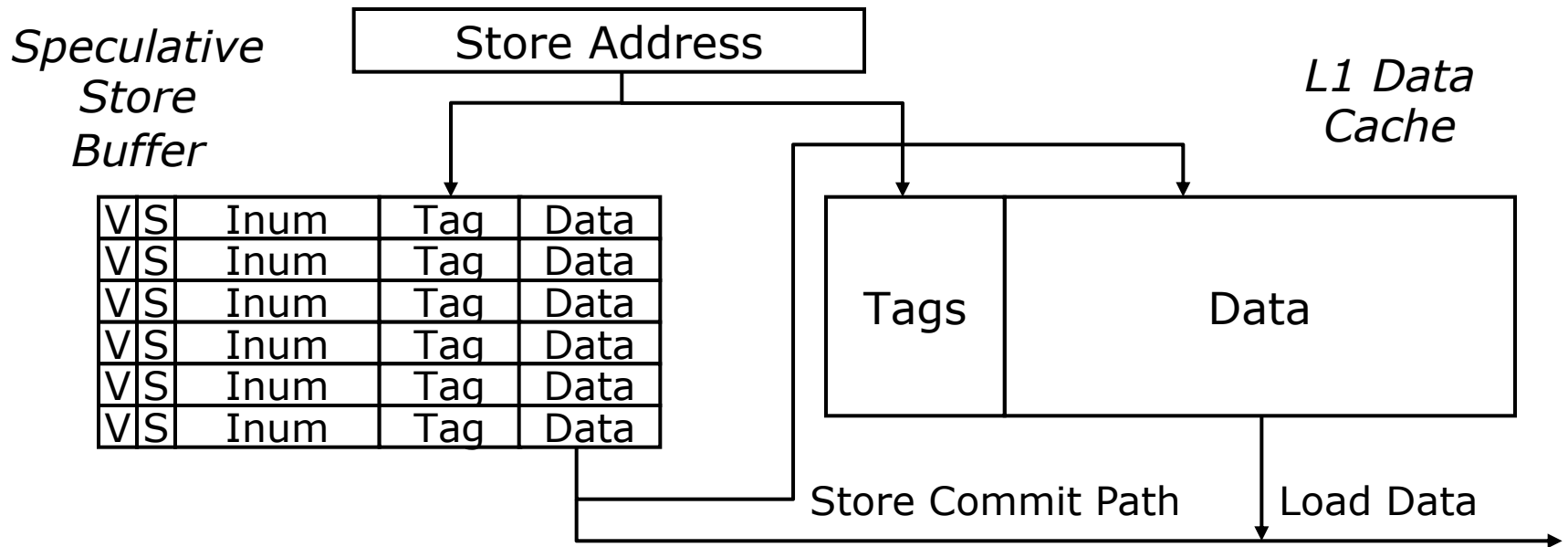
Commits are generally done in order – why?

WAW Hazards

Store Buffer – Lazy data management

Speculative Store Buffer

Store Address

L1 Data Cache

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

Tags

Data

Store Commit Path

Load Data

- On store execute:

Store Buffer – Lazy data management

Speculative Store Buffer	Store Address				L1 Data Cache

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

Tags   Data
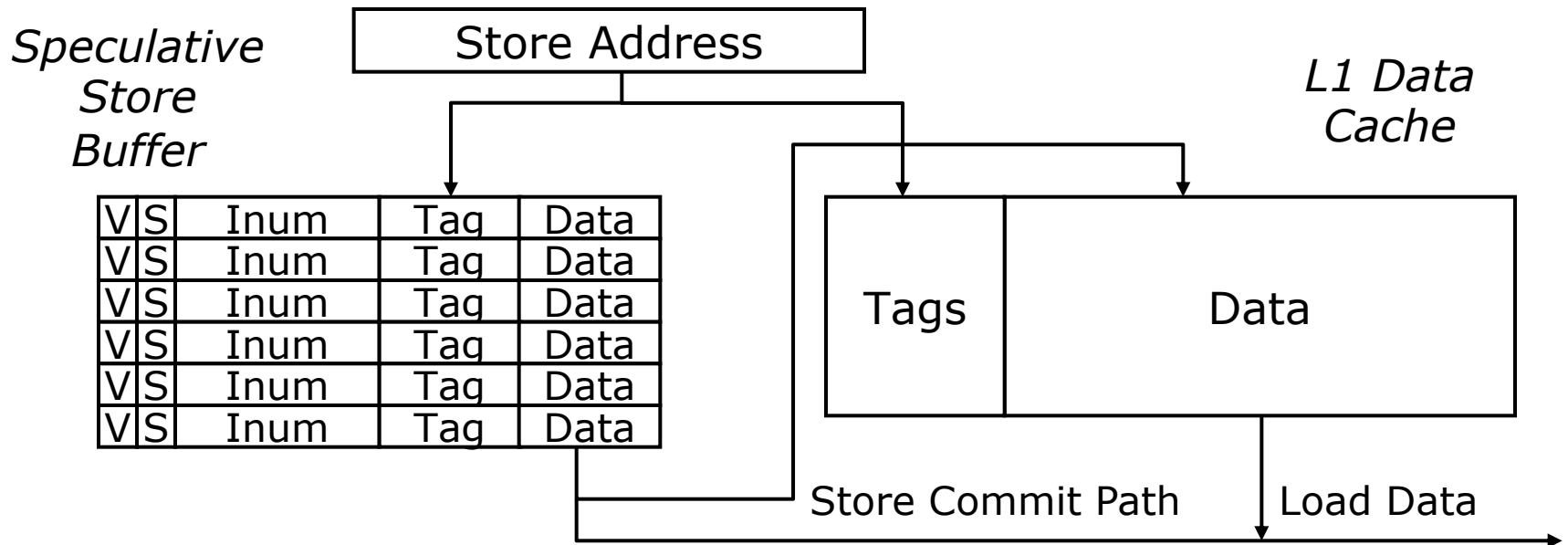
Store Commit Path   Load Data

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

Store Buffer – Lazy data management
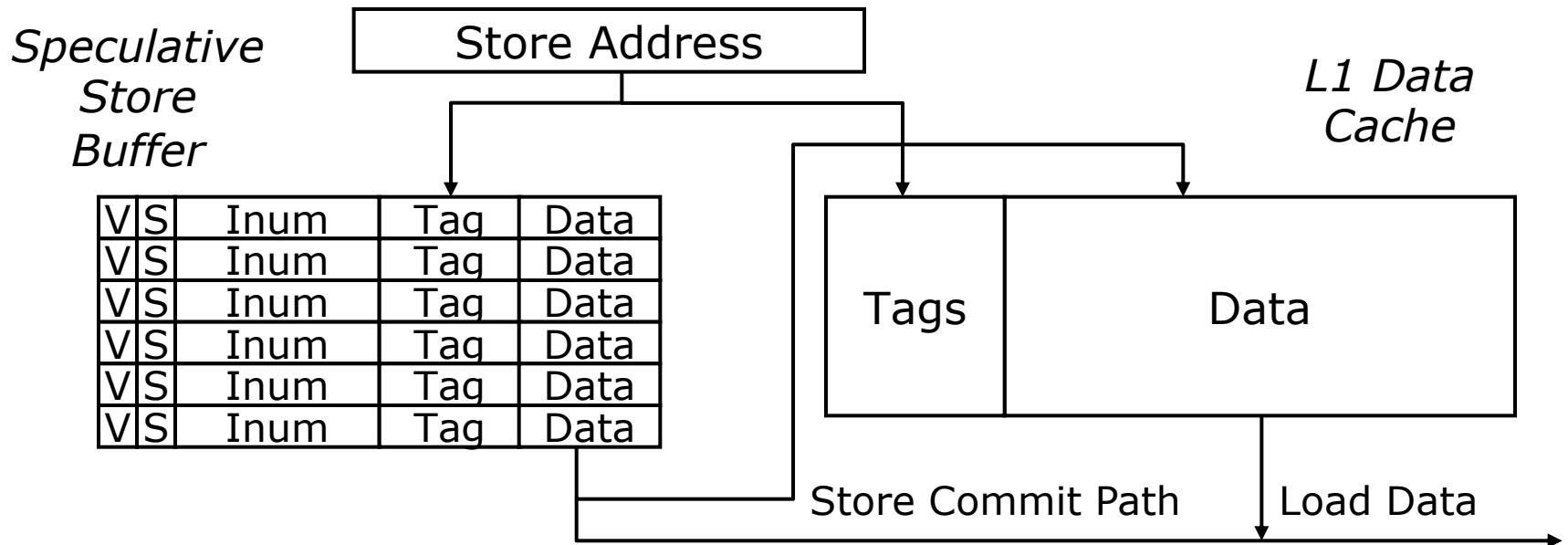
		Speculative Store Buffer						*L1 Data Cache*

Store Address

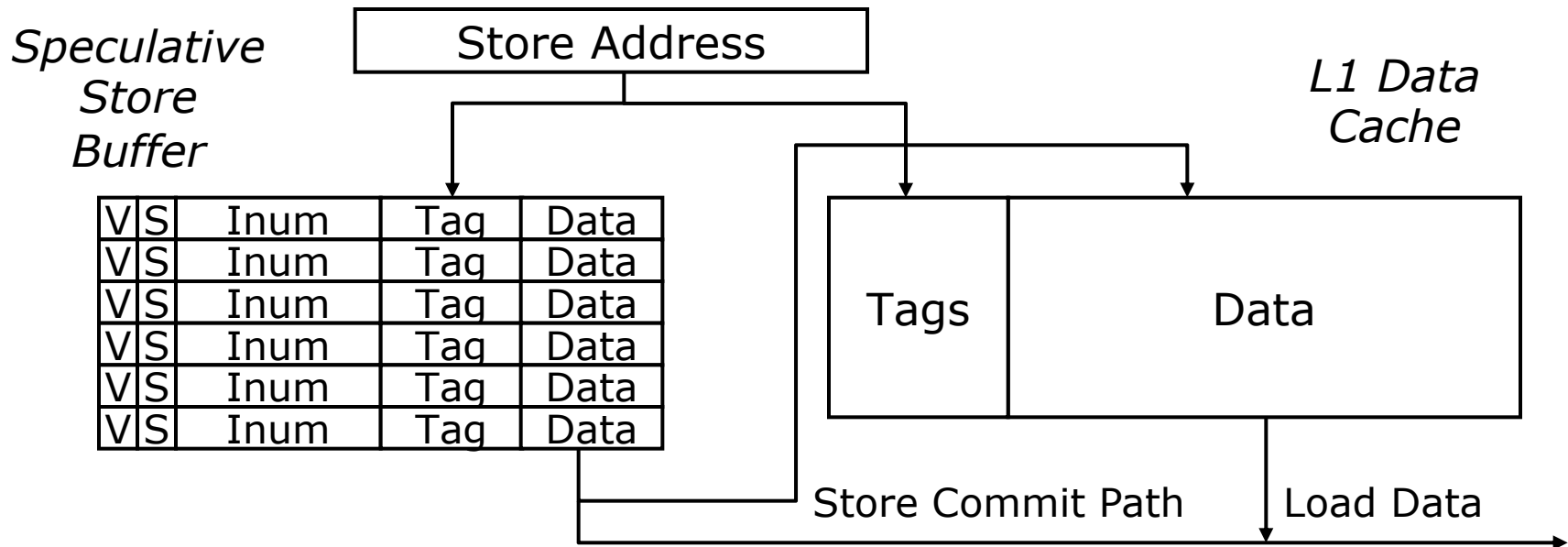| V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data
V	S	Inum	Tag	Data

Tags — Data

Store Commit Path   Load Data

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

Store Buffer – Lazy data management

Speculative Store Buffer

Store Address

L1 Data Cache

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

Tags

Data
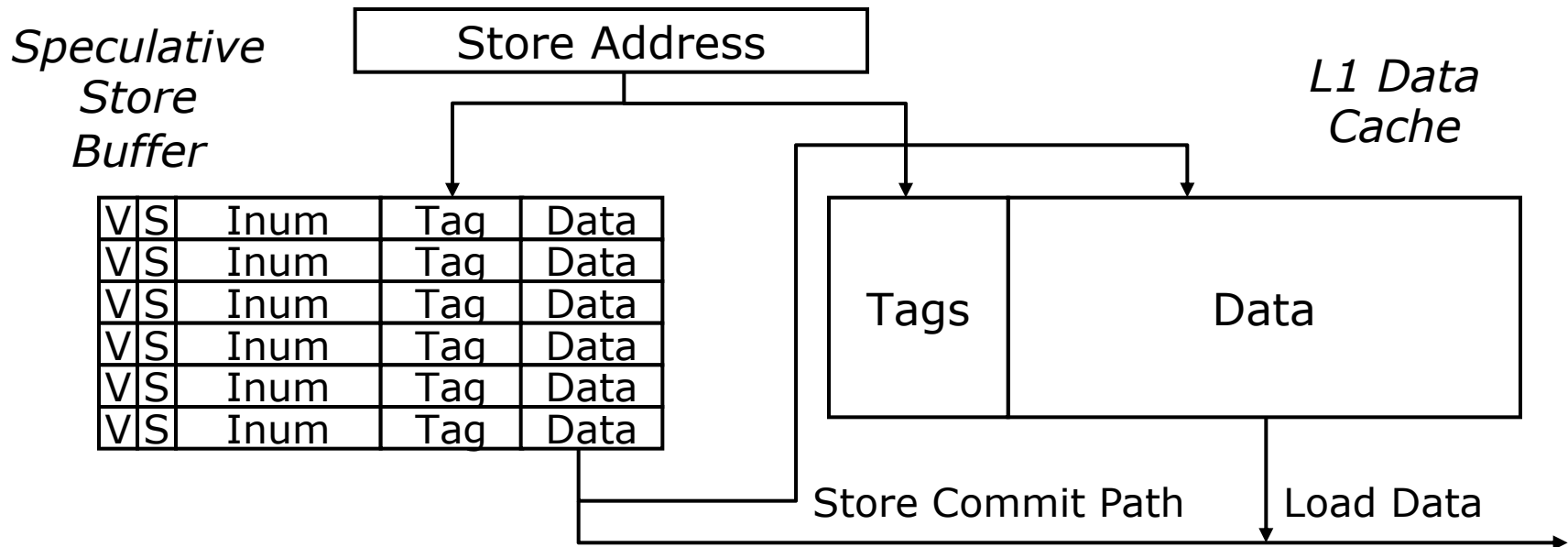
Store Commit Path   Load Data

- 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

Speculative Store Buffer	Store Address			L1 Data Cache

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

Tags   Data

Store Commit Path   Load Data

- **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
V	S	Inum	Tag	Data

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
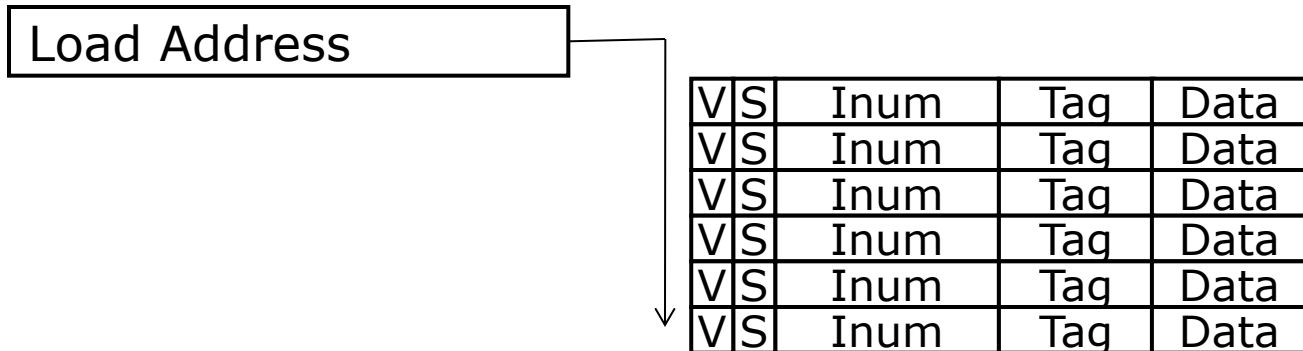
Store Buffer - Bypassing

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

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
V	S	Inum	Tag	Data

Load Address

What fields must be examined for bypassing?

Valid, Inum, and tag

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

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

What fields must be examined for bypassing?

Valid, Inum, and tag

- 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

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
V	S	Inum	Tag	Data

What fields must be examined for bypassing?

Valid, Inum, and tag

- 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

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

What fields must be examined for bypassing?

Valid, Inum, and tag

- 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

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
V	S	Inum	Tag	Data

What fields must be examined for bypassing?

Valid, Inum, and tag

- 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	

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
V	S	Inum	Tag	Data

What fields must be examined for bypassing?

Valid, Inum, and tag

- 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?

sw x1, (x2)
lw x3, (x4)

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?

sw x1, (x2)

lw x3, (x4)

When is the load dependent on the store?

When x2 == x4

Memory Dependencies

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

sw x1, (x2)

lw x3, (x4)

When is the load dependent on the store?

When x2 == x4

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?

sw x1, (x2)

lw x3, (x4)

When is the load dependent on the store?

When x2 == x4

Does our ROB know this at issue time? No

In-Order Memory Queue

sw x1, (x2)
lw x3, (x4)

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

sw x1, (x2)
lw x3, (x4)

Stall intelligently:

- Split execution of store instruction into two phases: address calculation and data write

- 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, 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

sw x1, (x2)
lw x3, (x4)

Address Speculation

sw x1, (x2)
lw x3, (x4)

1. Guess that x4 != x2, and execute load before store address known

Address Speculation

sw x1, (x2)
lw x3, (x4)

1. Guess that x4 != x2, and execute load before store address known

2. If x4 != x2 commit…

Address Speculation

sw x1, (x2)
lw x3, (x4)

1. Guess that x4 != x2, and execute load before store address known

2. If x4 != x2 commit…

3. But if x4==x2, 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

sw x1, (x2)
lw x3, (x4)

1. Guess that x4 != x2, and execute load before store address known

2. If x4 != x2 commit...

3. But if x4==x2, 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

sw x1, (x2)
lw x3, (x4)

1. Guess that x4 != x2, and execute load before store address known

2. If x4 != x2 commit…

3. But if x4==x2, 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
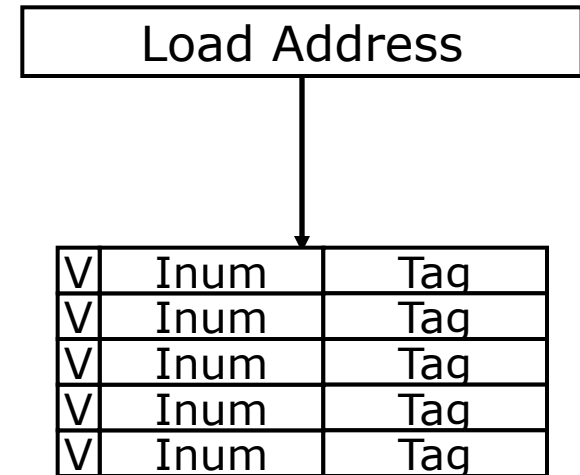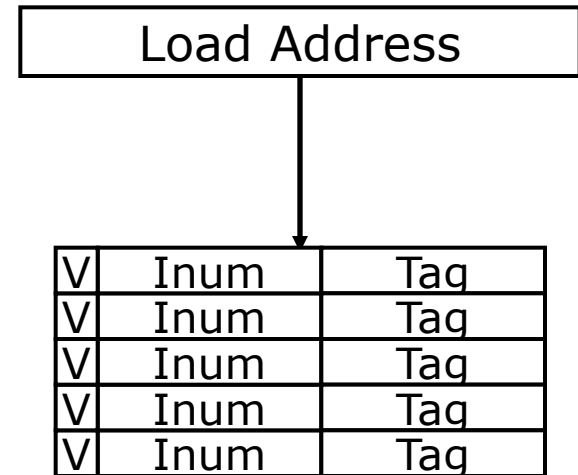
Speculative Load Buffer

Load Address		

V	Inum	Tag
V	Inum	Tag
V	Inum	Tag
V	Inum	Tag
V	Inum	Tag
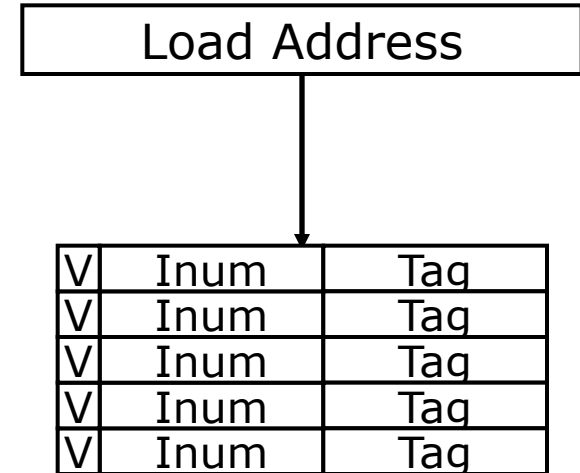
- On load execute:

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

Load Address

V	Inum	Tag
V	Inum	Tag
V	Inum	Tag
V	Inum	Tag
V	Inum	Tag

- On load execute:
  - mark entry valid, and set instruction number and address tag.

Speculative Load Buffer

Speculation check: Detect if a load has executed before an earlier store to the same address – missed RAW hazard
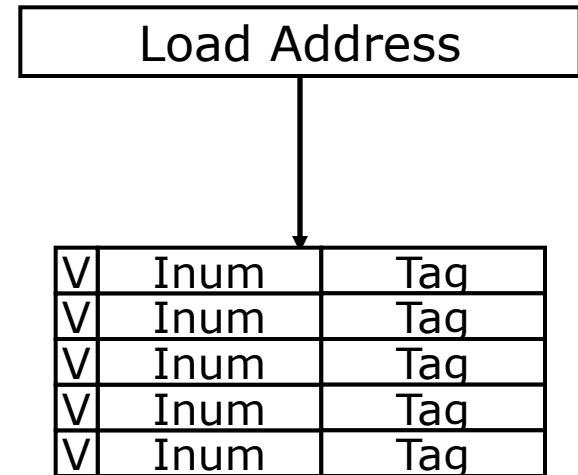
Speculative Load Buffer

	Load Address	

V	Inum	Tag
V	Inum	Tag
V	Inum	Tag
V	Inum	Tag
V	Inum	Tag

- On load execute:
  - mark entry valid, and set instruction number and address tag.
- On load commit:

Speculative Load Buffer

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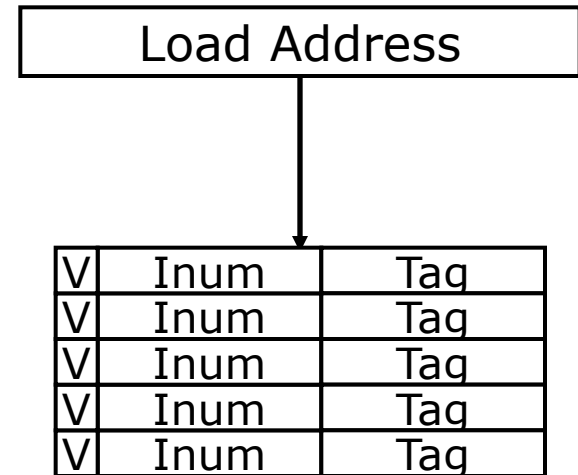RAW hazard

Speculative
Load Buffer

Load Address

V	Inum	Tag
V	Inum	Tag
V	Inum	Tag
V	Inum	Tag
V	Inum	Tag

- On load execute:
  - mark entry valid, and set instruction number and address tag.
- 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
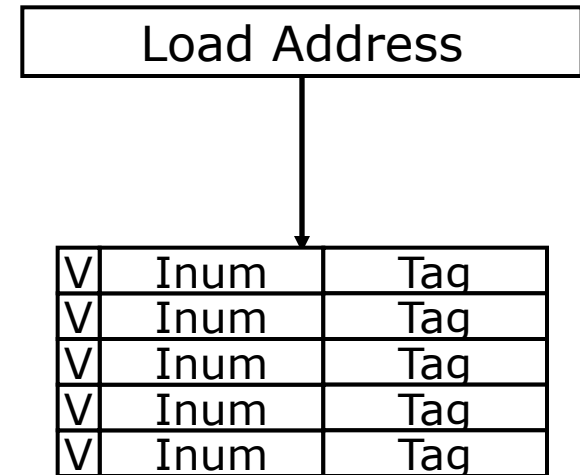
Speculative Load Buffer

	Load Address	

V	Inum	Tag
V	Inum	Tag
V	Inum	Tag
V	Inum	Tag
V	Inum	Tag

- On load execute:
  - mark entry valid, and set instruction number and address tag.
- 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
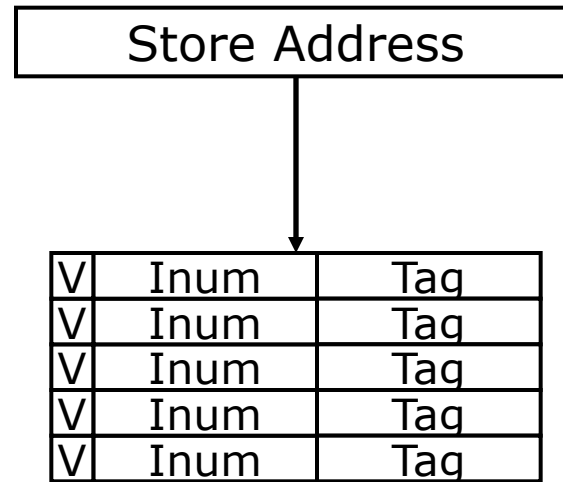
Speculative Load Buffer

Load Address

V	Inum	Tag
V	Inum	Tag
V	Inum	Tag
V	Inum	Tag
V	Inum	Tag

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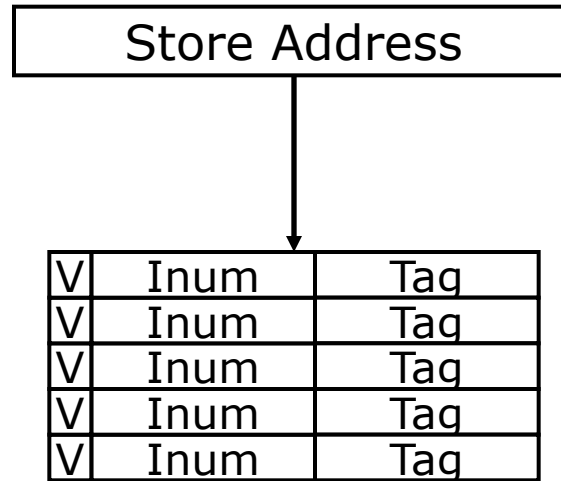 - clear valid bit

Speculative Load Buffer

Speculative Load Buffer

	Store Address	

V	Inum	Tag
V	Inum	Tag
V	Inum	Tag
V	Inum	Tag
V	Inum	Tag

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

Speculative Load Buffer

Speculative Load Buffer

Store Address

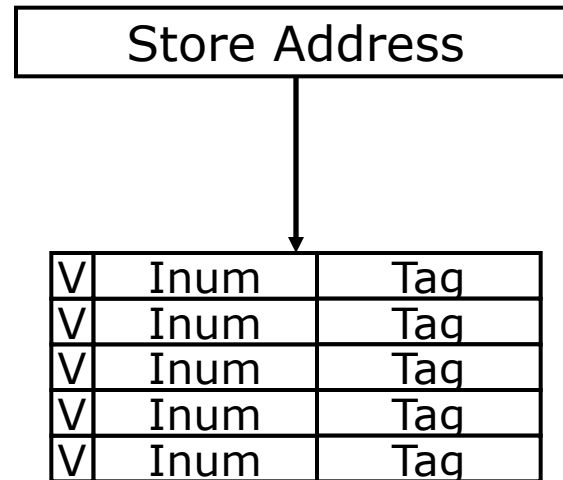| V	Inum	Tag
V	Inum	Tag
V	Inum	Tag
V	Inum	Tag
V	Inum	Tag

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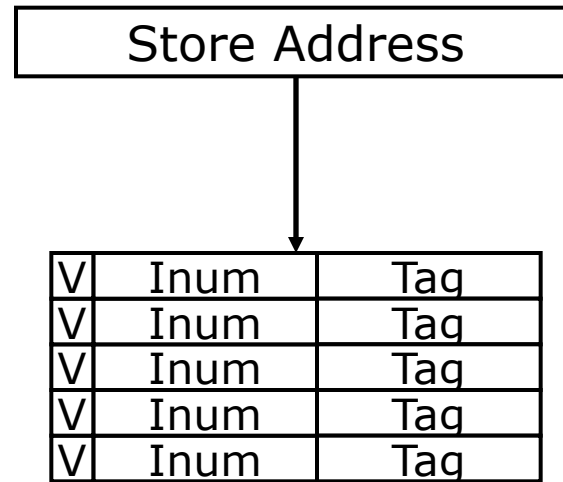   - Speculative violation – abort!

Speculative Load Buffer

Speculative
Load Buffer

Store Address

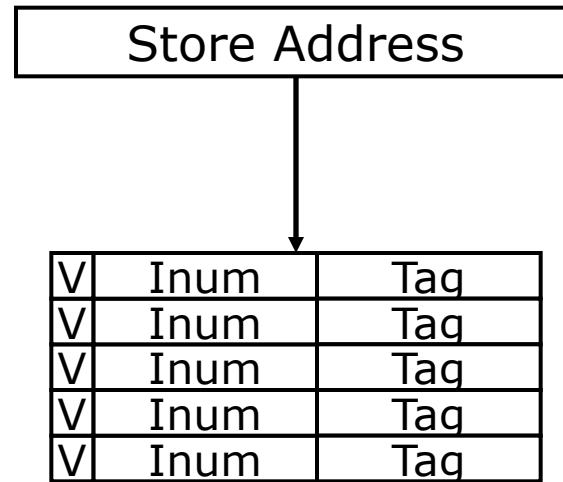| V	Inum	Tag
V	Inum	Tag
V	Inum	Tag
V	Inum	Tag
V	Inum	Tag

- **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

	Store Address	

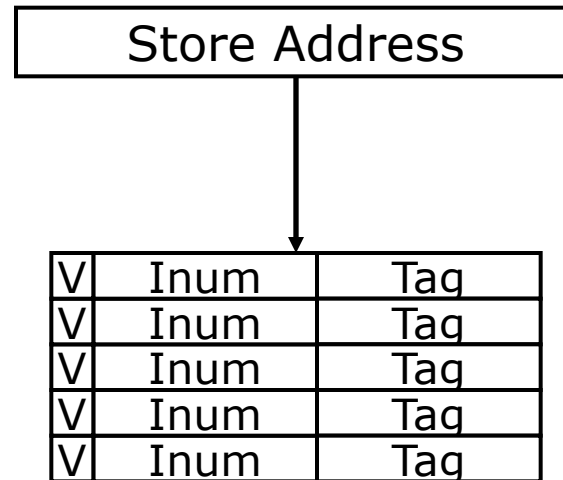|		
V	Inum	Tag
V	Inum	Tag
V	Inum	Tag
V	Inum	Tag
V	Inum	Tag

- **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

	Store Address	

V	Inum	Tag
V	Inum	Tag
V	Inum	Tag
V	Inum	Tag
V	Inum	Tag

- **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!

Speculative Load Buffer

Speculative Load Buffer

Store Address

V	Inum	Tag
V	Inum	Tag
V	Inum	Tag
V	Inum	Tag
V	Inum	Tag

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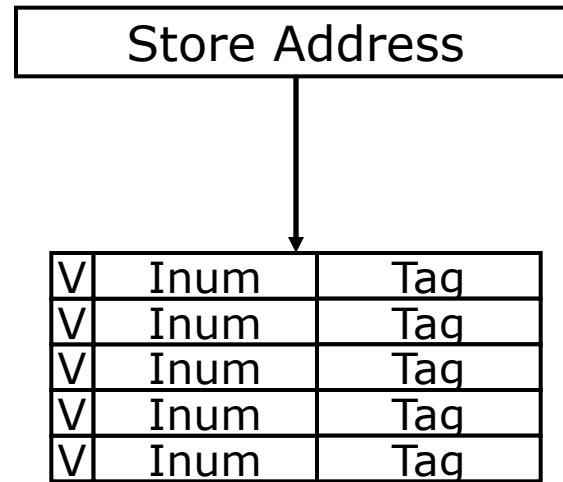 - Speculative violation – abort!

=> Large penalty for inaccurate address speculation

Does tag match have to be perfect?   No!

How do we reduce the chance of mis-speculation?

Speculative Load Buffer

Speculative Load Buffer

	Store Address	

V	Inum	Tag
V	Inum	Tag
V	Inum	Tag
V	Inum	Tag
V	Inum	Tag

- ## 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!

How do we reduce the chance of mis-speculation?   Predict

Memory Dependence Prediction
(Alpha 21264)

sw x1, (x2)
lw x3, (x4)

1. Guess that x4 != x2 and execute load before store

2. 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
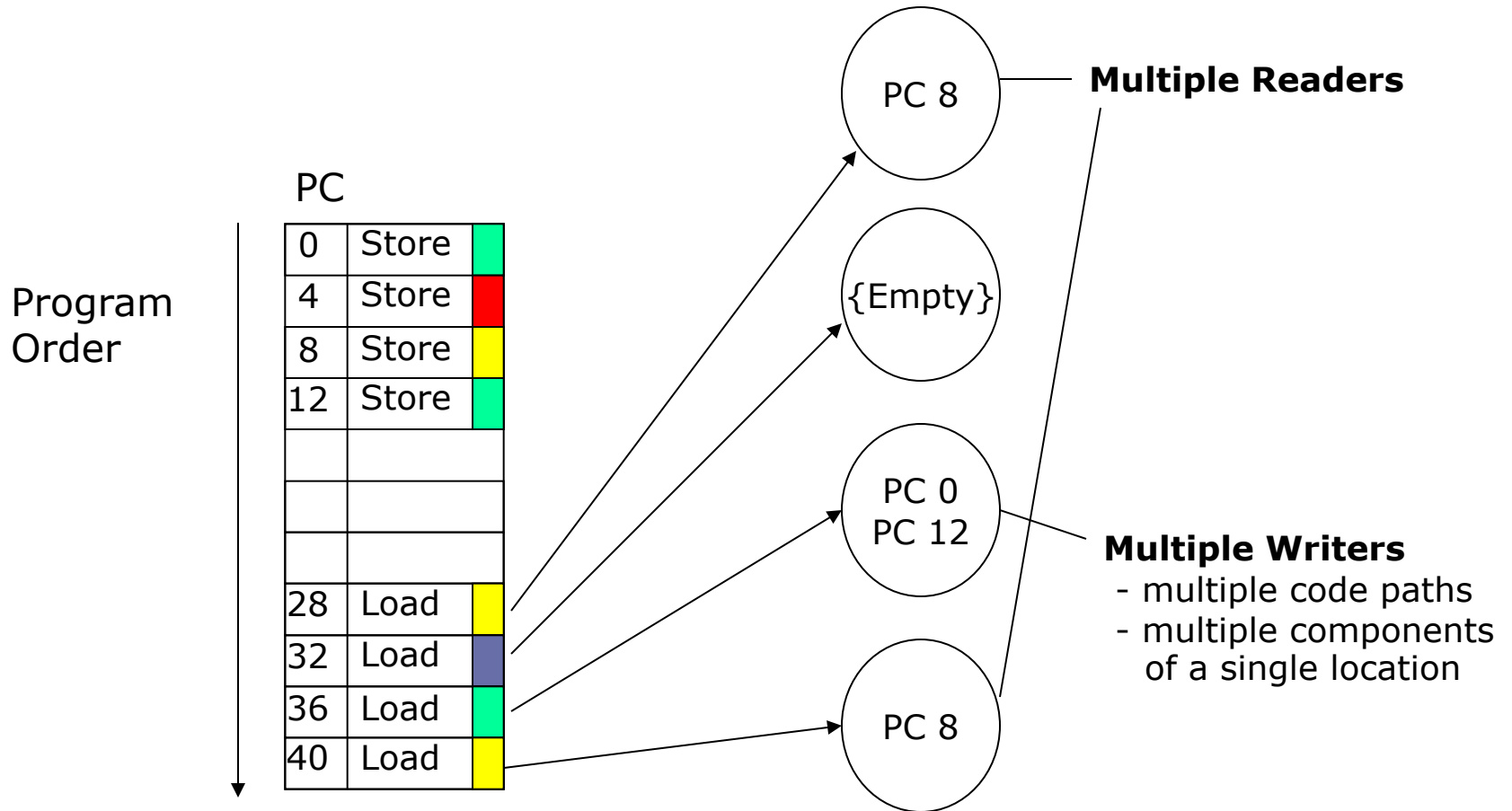
Memory Dependence Prediction
(Alpha 21264)

<div align="center">

sw x1, (x2)
lw x3, (x4)

</div>

1. Guess that x4 != x2 and execute load before store

2. 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

Notice the general problem of predictors that learn something but can't unlearn it
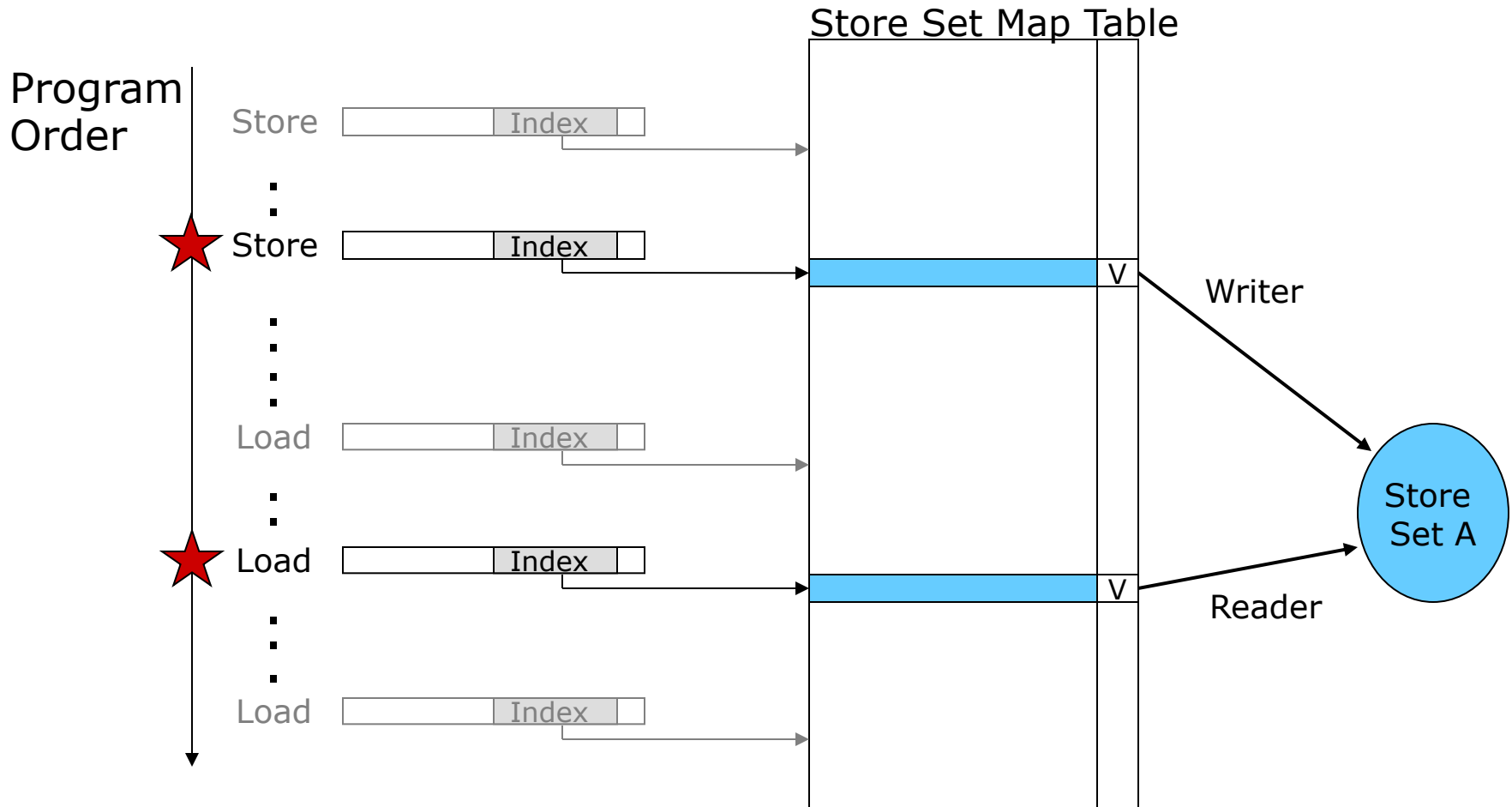
Store Sets
(Alpha 21464)

PC

0	Store	🟩
4	Store	🟥
8	Store	🟨
12	Store	🟩
28	Load	🟨
32	Load	🟦
36	Load	🟩
40	Load	🟨

Program Order

Multiple Readers

PC 8

{Empty}

PC 0
PC 12

PC 8

Multiple Writers
- multiple code paths
- multiple components of a single location

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
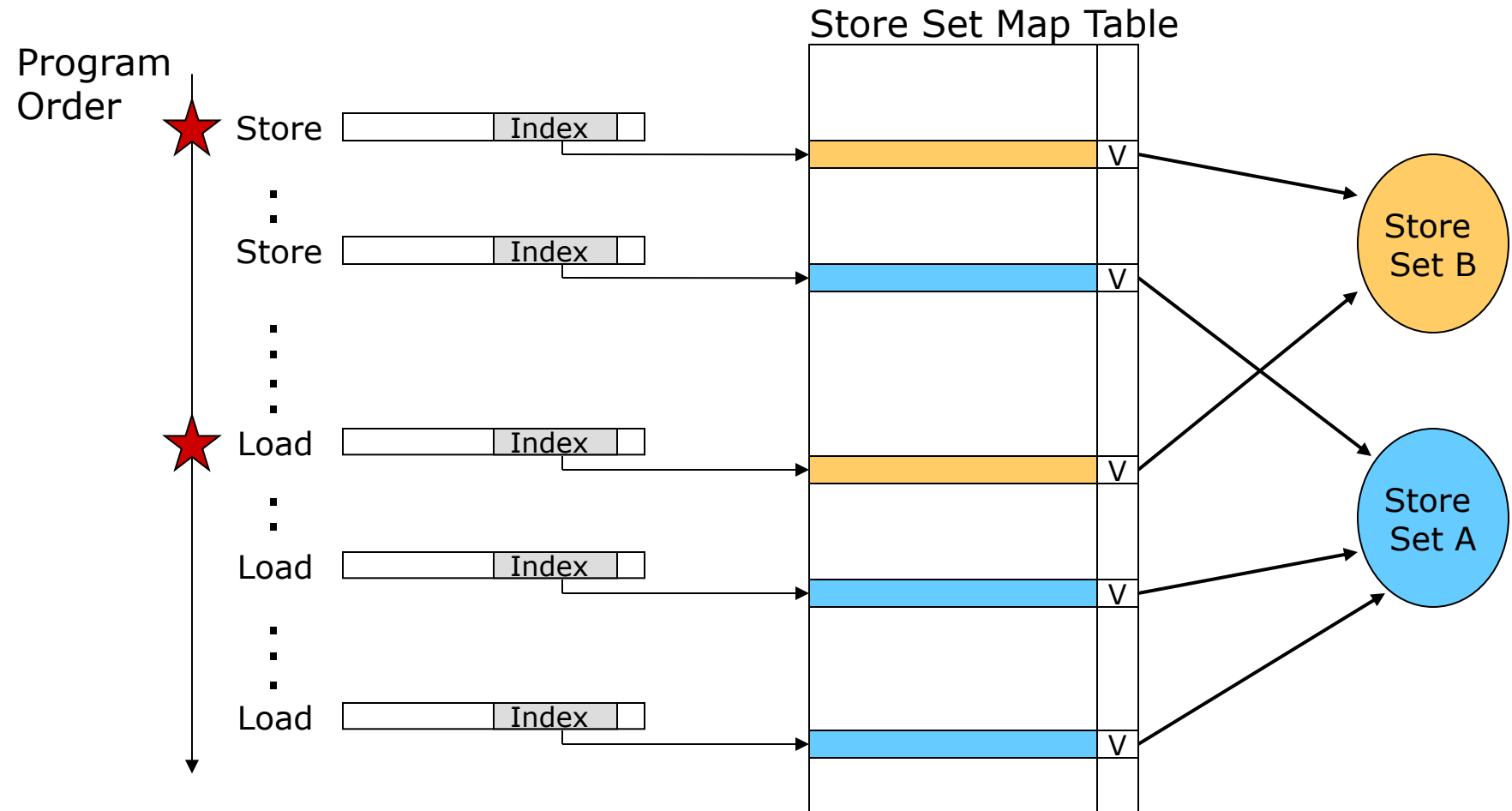
The Store Set Map Table

Store Set Map Table

Program Order

Store — Index

Store — Index — V → Writer

Load — Index

Load — Index — V → Reader

Load — Index

Store Set A

⭐ - Store/Load Pair causing Memory Order Violation

Store Set Sharing for Multiple Readers

Store Set Map Table

Program Order

Store Index

Store Index

Load Index

Load Index

Load Index
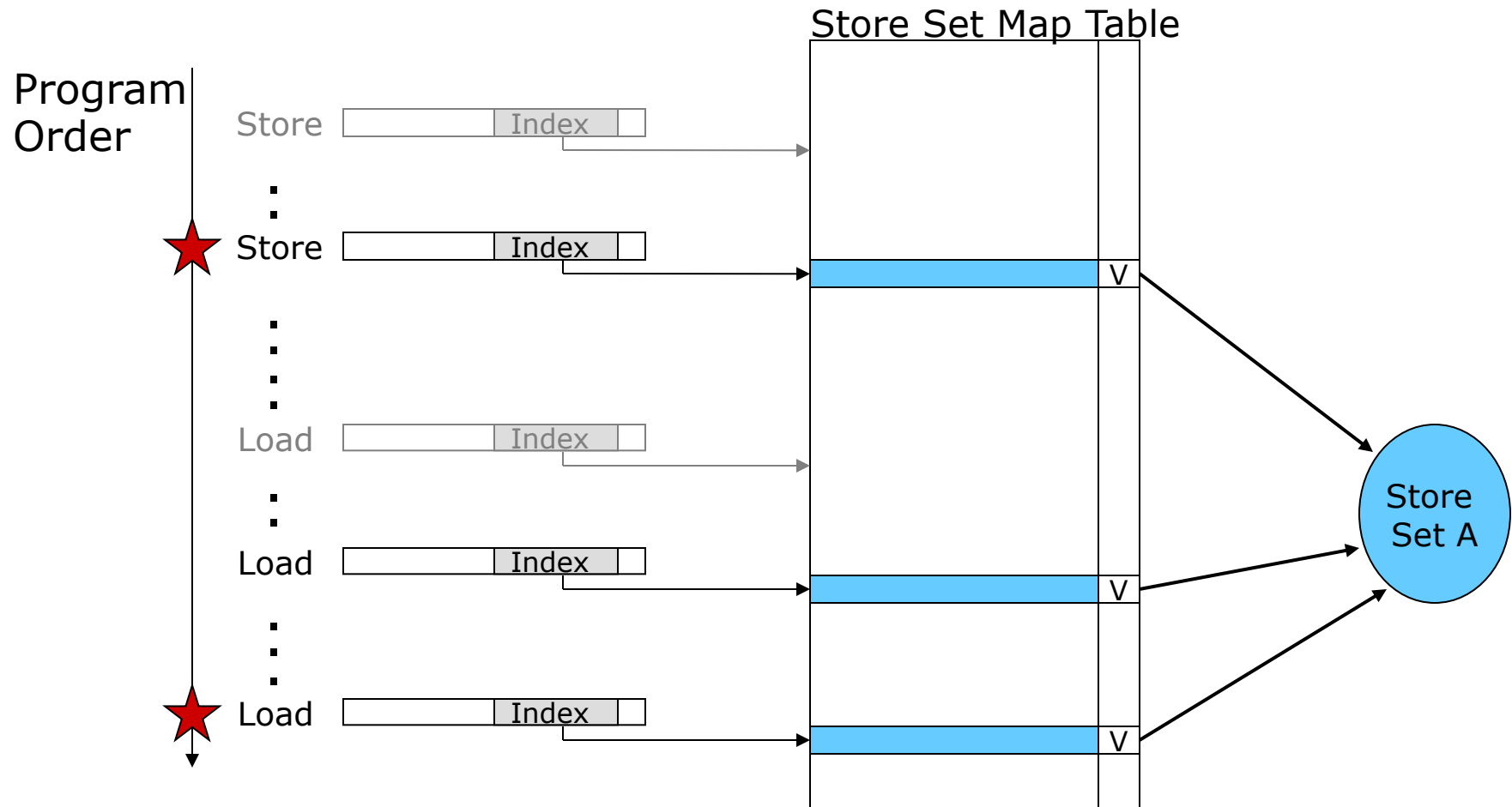
Store Set A

★ - Store/Load Pair causing Memory Order Violation

Store Set Map Table, cont.

Store Set Map Table

Program
Order

Store   [Index]

Store   [Index]

Load   [Index]

Load   [Index]

Load   [Index]

Store
Set B

Store
Set A

★ - 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   Conflict
  Reduce   ~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   Conflict   Capacity

  Reduce   ~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

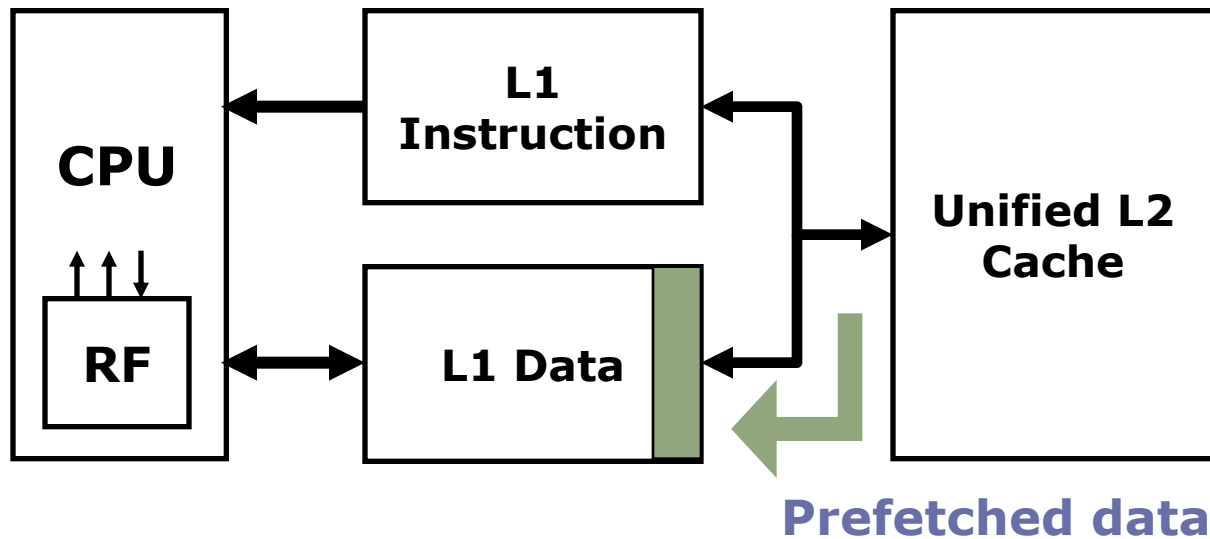- *How does prefetching affect cache misses?*

Compulsory	Conflict	Capacity
Reduce	~Increase	~Increase

Issues in Prefetching

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

Prefetched data

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)
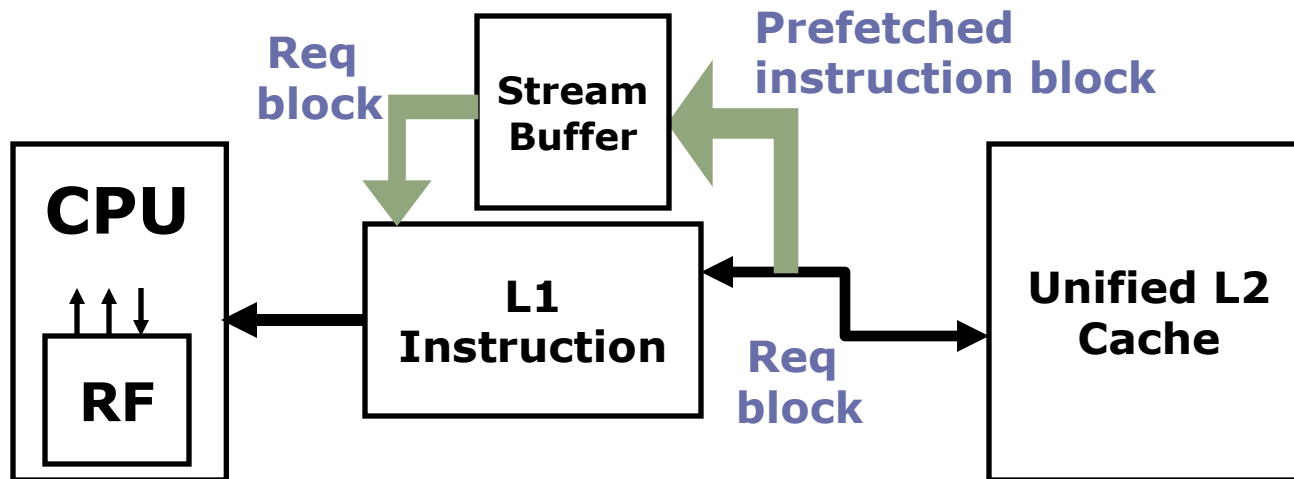
Hardware Data Prefetching

- 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

Good Luck on Quiz 1!

Next lecture:
Multithreading