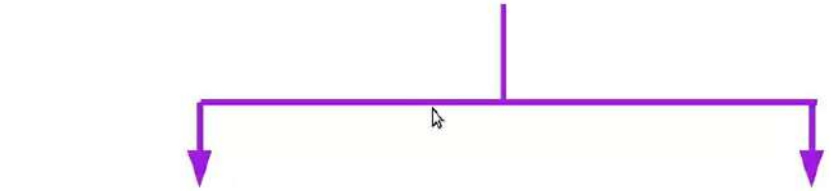
Snooping protocols



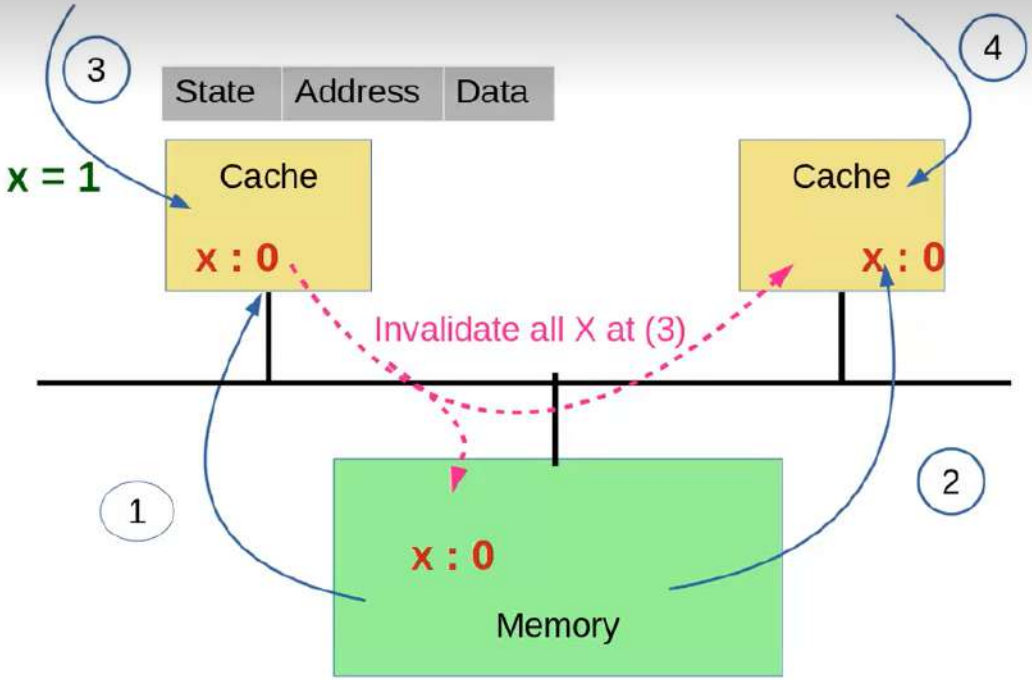
Write invalidate

Write update

Write-invalidate protocol

- Processor has exclusive access to data item before it writes to it
- i.e. a write by one processor, invalidates all other copies of that data
- -ve: readers will incur a miss
- Example= <next slide>
- +ve multiple writes (to same location) can be done by the processor without generating additional traffic
- +ve also, clear out copies that will never be used again

Example: Wr-inv protocol



(1) A reads X=0

(2) B read X=0

(3) A writes X=1;
Inv all copies,
memory updated

(4) B reads X
=> results in
cache miss

Cache containts
<state, addr, data>

Write-update protocol

- When shared item is written, update all cached copies
- It must broadcast the write to all
- This uses considerable bandwidth, therefore most recent systems use write-invalidate protocols
- +ve avoids misses on later references
- -ve multiple useless updates

Latest Copy search?

- The latest copy depends on type of cache: Write-through vs write-back
- Write-back
  - Search is hard, as latest copy may be in some other cache
  - -ve: this is a slow retrieval of data compared to memory read
  - +ve: less memory bandwidth needed in normal operation. Therefore can support large number of multiprocessors => more commonly used

Design space for Snooping Protocols

- In a write back cache, the processor holds the block while modifying it

  => called modified or dirty-block

  - Such a block is also said to be in **exclusive** mode
  - The cache is called **owner** of the block, as it must **supply the data** upon request for that block

- A cache has exclusive copy of the block if it is the only cache with a valid copy of the block
  - The memory may/may-not have valid copy

    => in this state the cache can modify data without informing others

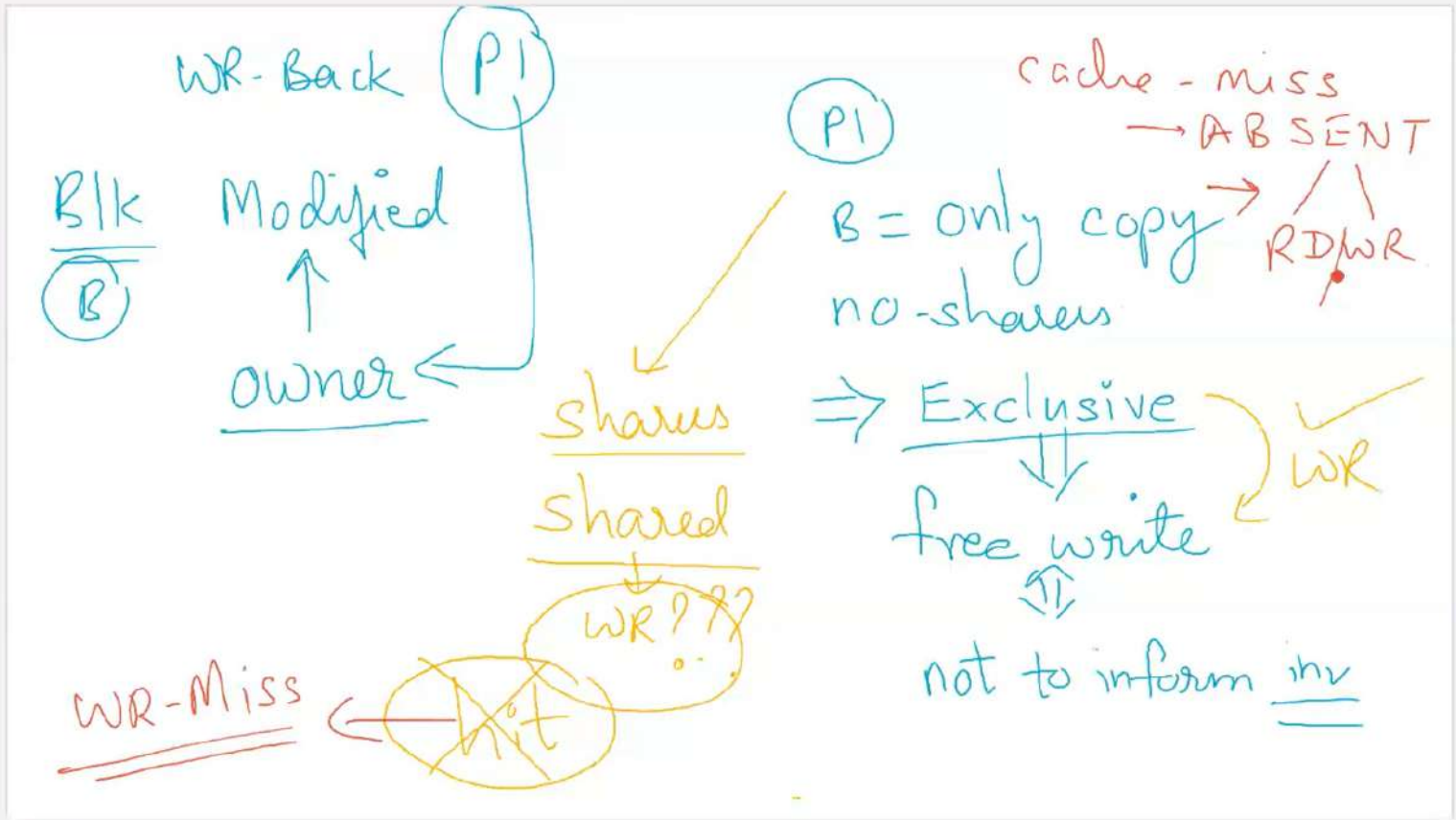
- If block is not exclusive, then need to inform others by generating bus transaction. This transaction is called a **write-miss**
- On a write-miss, in invalidation protocol, a read-exclusive transaction is generated to **tell other** caches of the impending write and to acquire the block in exclusive ownership

13 August 2021 - Goo × 06_Aug - Mentimeter +

jamboard.google.com/d/17PzgLpiDMGMwOzBQkYwX3AP0SrpzSlPyOMgXIjmkxj8/viewer?f=7

13 August 2021 8/8 Share

Set background Clear frame

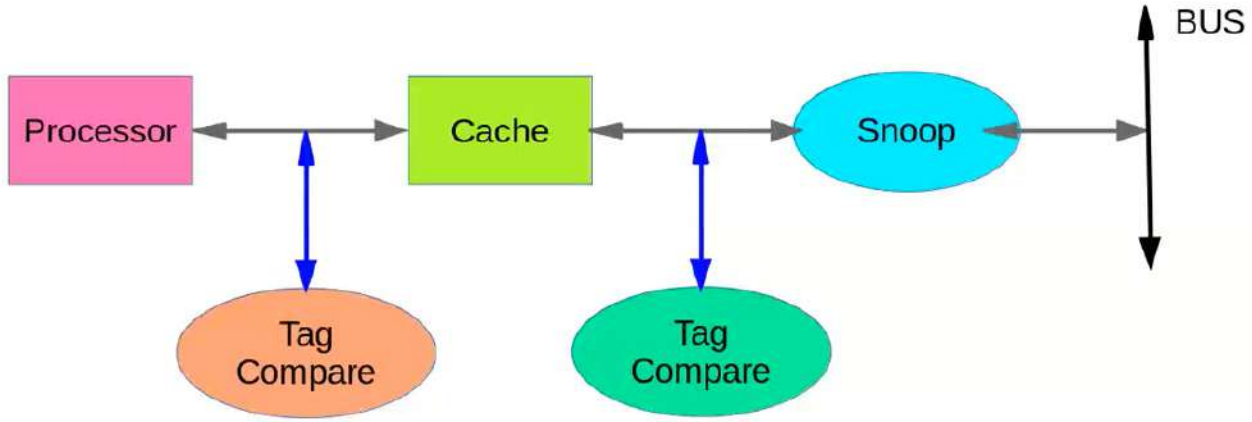


76% (2:46) Mon, August 16, 15:14

+20 AS DG DEVANSHI GUPTA NISHANK SIDDHARTH SS Syam Sankar SP SARASWATULA PHANI SAI PRANAV VA VARHADE AMEY ANANT IMLIJUNGLA LONGCHAR Hemangee Kalpesh Kapoor

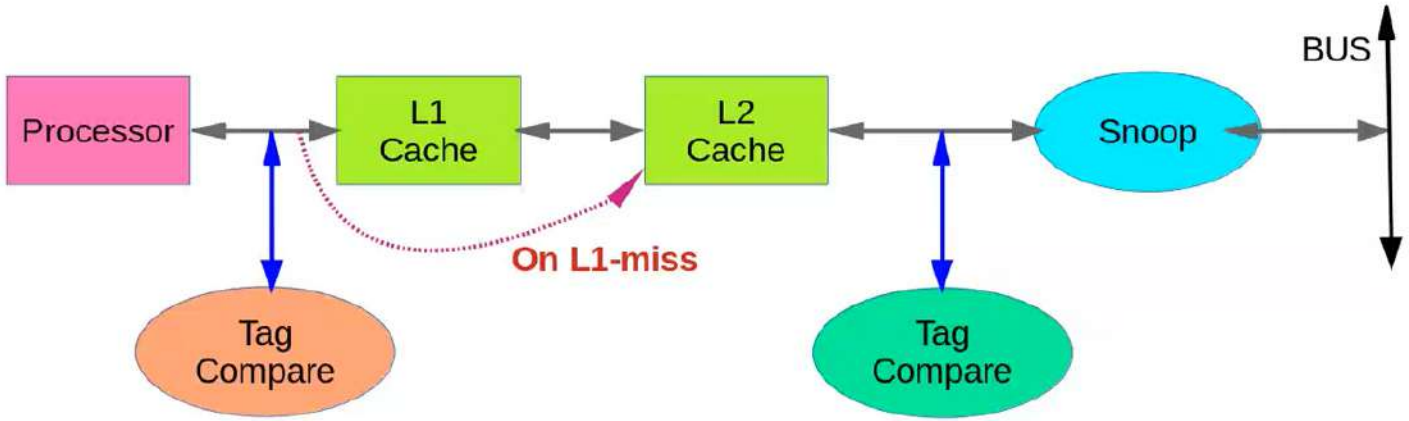
Cache behaviour in response to bus

- Every bus transaction must check the cache address tags. This checking of tags can come in the way of the processor accessing tags
- Solution?



Cache behaviour wrt Bus

- Solution-1: Duplicate tags
- Solution-2: Reduce snooping requests to L2-cache. Thus L1-cache is free



Solution-2

- L2 accessed only on L1-miss
- Therefore less interference
- Works in inclusive property (L1 in L2 always)
- If snoop hits in L2 then snoop must arbitrate for L1 to change state in L1 and possibly to retrieve data from L1
- L2-tags may also be duplicated

Snooping Protocols

Snooping Coherence Protocols

(1)2-state (VI)

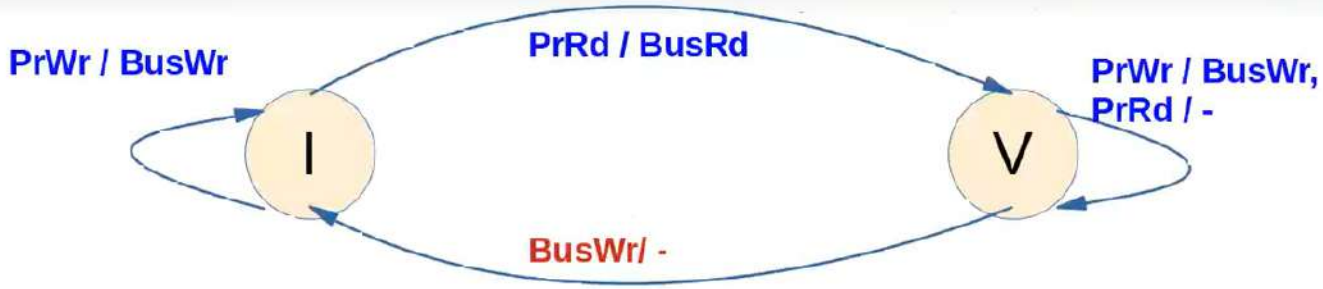
(2)3-state (MSI)

(3)4-state (MESI)

(4)4-state (Dragon), wr-back update

2-state: VI

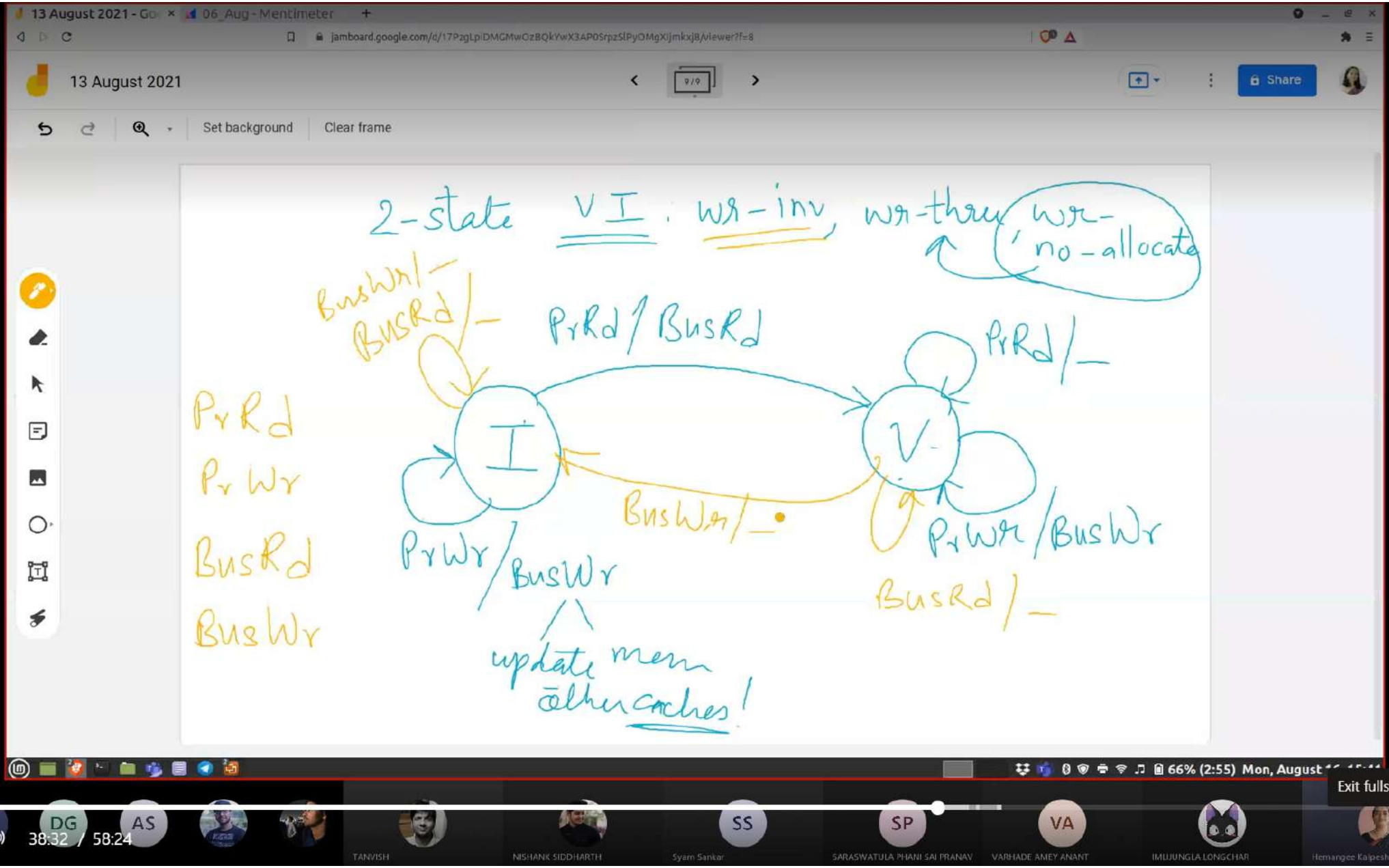
2-state Valid-Invalid protocol



- Can have multiple simultaneous readers of block
- But write invalidates them
- Wr-thru: all writes go to cache+memory (next level)
- Wr-no-allocate: if only WR do not load the block in cache

2-state protocol transition table

State	This Processor		Other Processor	
	LOAD	STORE	LOAD miss	STORE miss
			BusRD	BusWr
Invalid (I)	Miss → V	Wr-thru → I	-	-
Valid (V)	Hit	Hit + wr-thru	- (will load from memory)	Change state to I (data up-to-date in memory)


13 August 2021 - Go
06_Aug - MentiMeter
jamboard.google.com/d/17PzgLpiDMGMwOzBQkYwX3AP0SrpzSlPyOMgXIjmkxJ8/viewer?f=8
13 August 2021
9/9
Share
Set background
Clear frame
2-state VI : wr-inv, wr-thru
wr-no-allocate
BusWr/_
BusRd/_
PrRd/BusRd
PrRd/_
PrRd
PrWr
BusRd
BusWr
I
V
BusWr/_
PrWr/BusWr
PrWr/BusWr
BusRd/_
update mem
& other caches!
66% (2:55) Mon, August
Exit fullscreen (F)
38:32 / 58:24
TANVISH
NISHANK SIDDHARTH
Syam Sankar
SARASWATULA PHANI SAI PRANAV
VARHADE AMEY ANANT
IMLIJUNGLA LONGCHAR
Hemangee Kalpesh Kapoor

VI protocol satisfies coherence

- We need to show that for any execution under the protocol a total order on the memory operations for a location can be constructed that satisfies the program order and write serialisation conditions
- *Assume: atomic bus, only one transaction at a time, all actions related to the transaction complete before next can begin, processor waits for a memory operation to complete before issuing the next one, all caches apply invalidations immediately as part of the bus transaction, memory handles reads and writes in the order in which they appear on the bus*

- As it is wr-thru all writes go on the bus and As only one bus transaction is in progress at a time, in any execution all writes to a location are serialised by the order in which they appear on the shared bus, called the bus order
- Invalidations are also performed in bus order as all snooping caches inv when they see the bus transaction

VI correctness (write serialisation)

- Writes are seen by the processor, by issuing reads
- So, for proving write serialisation we must ensure that reads by all processors see the writes in the serialised bus order
- However, not all reads appear on the bus. So how do we insert them in the serial order
- Reads appearing on the bus - read miss
  - will be serialised by the bus along with the writes, and hence it will get the value written by the most recent write in bus order
- Reads not-appearing on the bus - read hits
  - Here the value read was written to the cache by the most recent write done by the same processor OR by this processor's most recent read-miss (in program order)
  - Both these sources of values appear on the bus => read hits also see the values produced in the consistent bus order
- **Thus BUS-order along with PROGRAM order provide enough constraints to satisfy the demands of coherence**

How to construct the total order?

- We can construct the (hypothetical) total order that satisfies coherence by observing the following partial orders imposed by the protocol:
  - A memory operation M2 is subsequent to a memory operation M1 if the operations are issued by the same processor and M2 follows M1 in program order
  - A read operation is subsequent to a write operation W if the read generates a bus transaction that follows that for W
  - A write operation is subsequent to a read or write operation M if M generates a bus transaction and the bus transaction for the write follows that for M
  - A write operation is subsequent to a read operation M if the read does not generate a bus transaction (is a hit) and is not already separated from the write by another bus transaction

How to construct the total order?

- Any serial order that preserves the resulting partial order is coherent
- The subsequent ordering relationship is transitive
- Several read transaction happening inside the processor which do not appear on the bus are not ordered by the bus
- In fact any interleaving of read operations in these internal segments is a valid serial order, as long as it obeys the program order