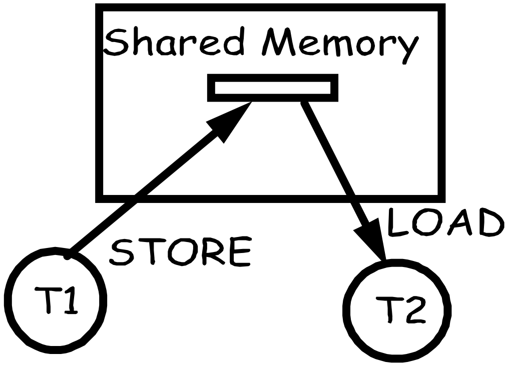
CHAPTER 7

COHERENCE, SYNCHRONIZATION AND MEMORY CONSISTENCY

- SYNCHRONIZATION
- COHERENCE AND STORE ATOMICITY
- SEQUENTIAL CONSISTENCY
- MEMORY CONSISTENCY MODELS
- SPECULATIVE VIOLATIONS OF MCMs

SHARED-MEMORY COMMUNICATION

- **IMPLICITELY VIA MEMORY**



- PROCESSORS SHARE SOME MEMORY
- COMMUNICATION IS IMPLICIT THROUGH LOADS AND STORES
  - NEED TO SYNCHRONIZE
  - NEED TO KNOW HOW THE HARDWARE INTERLEAVES ACCESSES FROM DIFFERENT PROCESSORS

NO ASSUMPTION ON THE RELATIVE SPEED OF PROCESSORS

SYNCHRONIZATION

- Need for “Mutual Exclusion”
- Assume the following statements are executed by 2 threads, T1 and T2, on A

```
T1                      T2
A<- A+1                 A<- A+1
```

- The programmer’s expectation is that, whatever the order of execution of the two statements is, the final result will be that A is incremented by 2
- However program statements are not executed in an atomic fashion.

- Compiled code on a RISC machine will include several instructions
- A possible interleaving is:

```
        T1                  T2
r1 <- A
                        r1 <- A
r1 <- r1 + 1
                        r1 <- r1 + 1
A <- r1
                        A <- r1
```

- At the end the result is that A has been incremented by 1 (NOT 2)

MUTUAL EXCLUSION

- We must have a way to make program statements appear atomic
- Critical sections
  - provided by lock and unlock primitives framing the statement(s)
  - modifications are "released" atomically at the end of the critical section
- So the code should be:

```
        T1                       T2
  lock(La)                 lock(La)                       /acquire
  A<- A+1                  A<- A+1
  unlock(La)               unlock(La)                     /release
```

HONOR SYSTEM

DEKKER'S ALGORITHMS FOR LOCKING

- ASSUME A AND B ARE BOTH 0 INITIALLY

```
   T1                          T2
 A:=1                        B:=1                          /acquire
 while(B==1);                while(A==1);
 <critical section>          <critical section>
 A:=0                        B:=0                          /release
```

- At most one process can be in the critical section at any one time.
- Deadlock
- Complex (to solve deadlock and synchronize more than 2 threads)

USE HARDWARE PRIMITIVES TO SYNCHRONIZE

BARRIER SYNCHRONIZATION

- **Global synchronization among all threads**
- **ALL threads must reach the barrier before ANY thread is allowed to execute beyond the barrier**

```
P1                                  P2
...                                 ...
BAR := BAR+1;                   BAR := BAR +1;
while (BAR < 2);                while (BAR < 2);
```

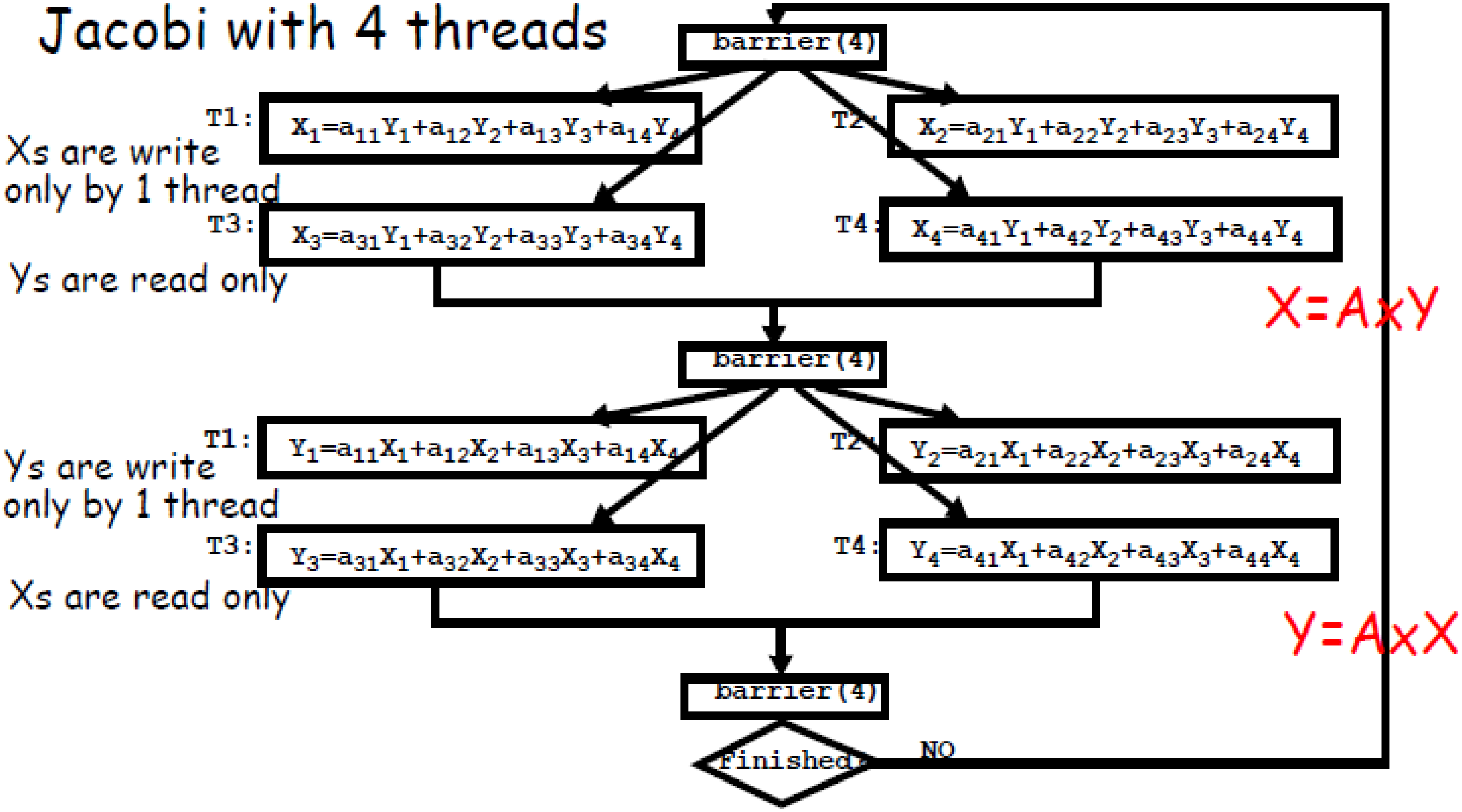
- **Note: need a critical section to increment BAR**
  - no need of a critical section to read BAR in the while statement
- **In practice, more complex because barrier count must be reset for the next iteration**
- **Barriers can effectively implement critical sections**

POINT-TO-POINT SYNCHRONIZATION

```
T1                                  T2
                                    A = 1;
                                    FLAG = 1;          /release
while (FLAG==0);                                       /acquire
print A
```

- **Note: no need for critical sections to update and read FLAG**
- **Signal sent by T1 to T2 through FLAG (Producer/Consumer synchronization)**

EXAMPLE



Jacobi with 4 threads

barrier(4)

T1: $X_1=a_{11}Y_1+a_{12}Y_2+a_{13}Y_3+a_{14}Y_4$

T2: $X_2=a_{21}Y_1+a_{22}Y_2+a_{23}Y_3+a_{24}Y_4$

T3: $X_3=a_{31}Y_1+a_{32}Y_2+a_{33}Y_3+a_{34}Y_4$

T4: $X_4=a_{41}Y_1+a_{42}Y_2+a_{43}Y_3+a_{44}Y_4$

Xs are write only by 1 thread

Ys are read only

X=AxY

barrier(4)

T1: $Y_1=a_{11}X_1+a_{12}X_2+a_{13}X_3+a_{14}X_4$

T2: $Y_2=a_{21}X_1+a_{22}X_2+a_{23}X_3+a_{24}X_4$

T3: $Y_3=a_{31}X_1+a_{32}X_2+a_{33}X_3+a_{34}X_4$

T4: $Y_4=a_{41}X_1+a_{42}X_2+a_{43}X_3+a_{44}X_4$

Ys are write only by 1 thread

Xs are read only

Y=AxX

barrier(4)

Finished?

NO

- In 1st phase accesses to Xi's are mutually exclusive
- In 2nd phase, multiple accesses to Xi's (read-only)
- Opposite is true for Yi's

SYNCHRONIZATION

- **ACQUIRE METHOD**
  - ACQUIRE ACCESSES RIGHT TO THE SYNCHRONIZATION (ENTER CRITICAL SECTION, GO PAST EVENT)
- **RELEASE METHOD**
  - ENABLE OTHER PROCESSORS TO ACQUIRE THE RIGHT TO THE SYNCHRONIZATION
- **WAITING ALGORITHM**
  - BLOCKING
    - WAITING PROCESSES ARE DESCHEDULED
    - HIGH OVERHEAD
    - ALLOWS PROCESSOR TO WORK ON SOMETHING ELSE
  - BUSY WAITING
    - WAITING PROCESSES REPEATEDLY TEST A LOCATION UNTIL IT CHANGES VALUE
    - RELEASING PROCESS SETS THE LOCATION
    - LOW OVERHEAD BUT HOLDS THE PROCESSOR
    - MAY HAVE HIGH MEMORY/NETWORK TRAFFIC
  - IN HARWARE MULTITHREADED CORES, CONSIDER IT A LONG LATENCY EVENT AND SWITCH TO OTHER
- **BUSY-WAITING IS BETTER WHEN**
  - SCHEDULING OVERHEAD IS LARGER THAN EXPECTED WAIT TIME
  - NO OTHER TASK TO RUN
  - OS KERNEL

LOCKS

- **HARDWARE LOCKS**
  - SEPARATE LOCK LINES ON THE BUS: HOLDER OF A LOCK ASSERT THE LINE
  - LOCK REGISTERS
    - SET OF SHARED REGISTERS
  - INFLEXIBLE
    - NOT GOOD FOR GENERAL PURPOSE USE
    - HARDWIRED WAITING ALGORITHM
- **ISA SUPPORT: MOST MODERN MACHINES USE A FORM OF ATOMIC READ-MODIFY-WRITE**
  - IBM 370: ATOMIC COMPARE AND SWAP
  - X86: ANY INSTRUCTION CAN BE PREFIXED WITH A LOCK
  - SPARC: SWAP
  - MIPS, PowerPC: SUPPORT FROM PAIRS OF INSTRUCTIONS
    - LOAD-LOCKED, STORE-CONDITIONAL

THESE BASIC MECHANISMS ARE USED TO BUILD SOFTWARE LOCKS

SIMPLE SOFTWARE LOCKS

```
Lock:       LW R2, lock
            BNEZ R2, Lock
            SW R1, lock                 /R1 = 1
            RET

Unlock:     SW R0, lock
            RET
```

- **PROBLEM: LOCK IS NOT ATOMIC--TWO THREADS CAN GAIN THE LOCK AT THE SAME TIME**

- **SOLUTION: ATOMIC READ/WRITE OR SWAP INSTRUCTION**
  - ATOMICALLY READ THE VALUE OF THE LOCATION AND SET IT TO ANOTHER VALUE
  - RETURN SUCCESS OR FAILURE

- **SIMPLEST ONE:** TEST_AND_SET (T&S)
  - T&S R1, lock
    - READ LOCK IN R1
    - WRITE 1 IN LOCK
    - SUCCESS IF VALUE READ IN R1 IS 0
    - FAILURE IF IT IS 1

MUST BE ATOMIC

SOFTWARE LOCKS

```
Lock:     T&S R1, lock
          BNEZ R1, Lock
          RET

Unlock:   SW R0, lock
          RET
```

- **OTHER R/M/W ATOMIC OPERATIONS**
  - SWAP R1, MEM_LOC: EXCHANGE THE CONTENT OF R1 AND MEM_LOC
  - FETCH&OP
    - EXAMPLE: F&A (R1, MEM_LOC, CONST), WHERE CONST IS A SMALL VALUE.
    - FETCH MEM_LOC IN R1, THEN ADD CONST TO MEM_LOC.
  - COMPARE&SWAP
    - CAS (R1, R2, MEM_LOC)
    - COMPARE MEM_LOC TO R1 AND IF THEY ARE EQUAL SWAP R2 AND MEM_LOC

REDUCE FREQUENCY OF ISSUING T&S

- **T&S WITH BACKOFF**
  - INCREASE THE DELAY UNTIL THE NEXT TRIAL AFTER EVERY FAILURE
  - E.G., EXPONENTIAL BACKOFF
  - BACKOFF BY k X c^i AT THE ith TRIAL

- **TEST AND TEST&SET LOCK**
  - TEST WITH ORDINARY LOADS
  - WHEN VALUE CHANGES TO 0, TRY TO OBTAIN LOCK WITH T&S
  - WORKS WELL WITH CACHE

```
Lock:     LW R1,lock
          BNEZ R1,Lock
          T&S R1,lock
          BNEZ R1,Lock
          RET

Unlock:   SW R0,lock
          RET
```

LOAD-LOCKED AND STORE CONDITIONAL

```
T&S(Rx,lock):   ADDI R1,R0,1
                LL Rx,lock
                SC R1,lock
                BEQZ R1, T&S
                RET
```

- **LOAD-LOCKED or LOAD-LINKED (LL)**
  - LL READS lock IN REGISTER Rx
- **STORE CONDITIONAL (SC)**
  - TRIES TO STORE 1 IN lock:
  - SUCCEEDS IF NO OTHER THREAD HAS WRITTEN INTO lock SINCE LL
  - IF SC SUCCEEDS THE SEQUENCE LL-SC WAS ATOMIC
  - IF SC FAILS, IT DOES NOT WRITE TO MEMORY; RATHER IT SETS R1 TO 0
- **SC CAN FAIL IF**
  - IT DETECTS INTERVENING WRITES TO lock SINCE LL
  - IT TRIES TO GET THE BUS, BUT ANOTHER SC SUCCEEDS FIRST
- **TWO ADVANTAGES:**
  - EASIER TO IMPLEMENT IN A PIPELINE
  - FLEXIBILITY

LOAD-LOCKED AND STORE CONDITIONAL

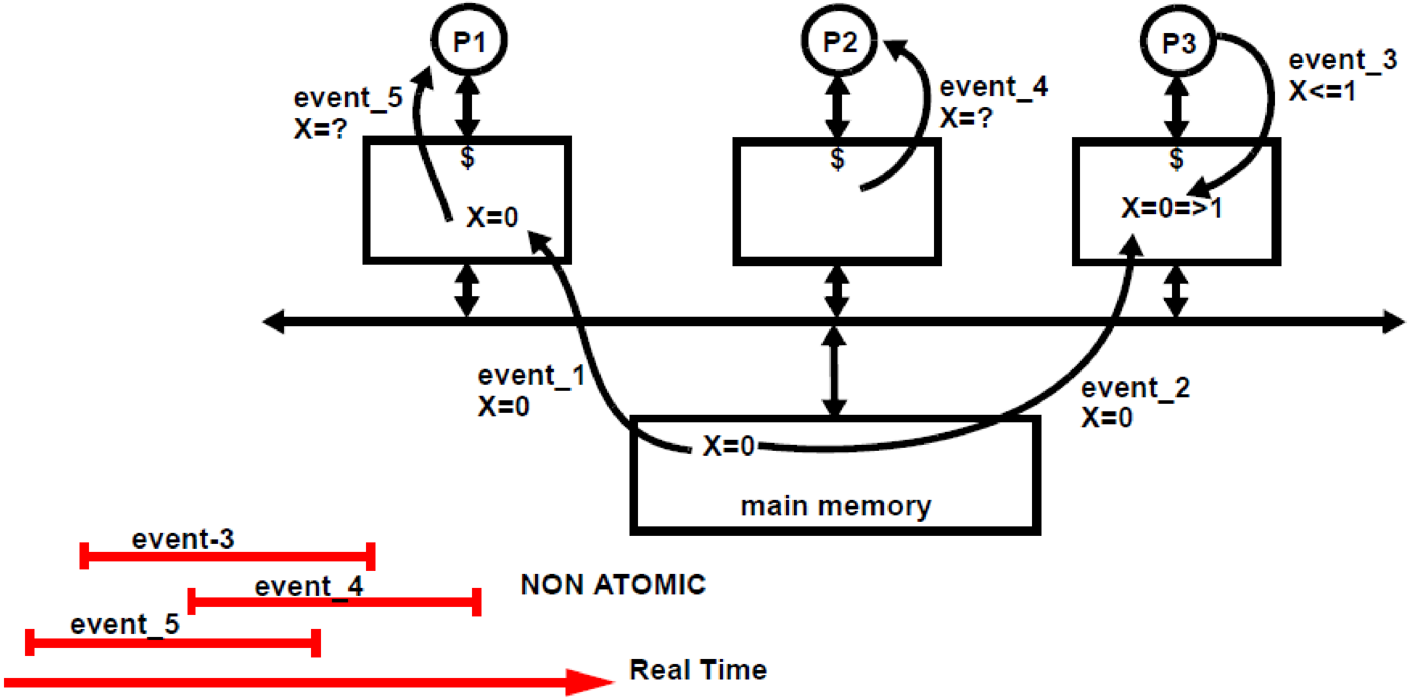
- **FANCIER ATOMIC OPS CAN BE IMPLEMENTED BY ADDING CODE BETWEEN LL AND SC**
  - KEEP IT SIMPLE SO THAT SC IS LIKELY TO SUCCEED
  - AVOID INSTRUCTIONS THAT CANNOT BE UNDONE (eg, STORE, INSTRUCTIONS CAUSING EXCEPTIONS)

- **EXAMPLE: CAS**

```
CAS(Rx,Ry,X)        ADD R2,Ry,R0                     /save Ry
                    LL R1,X
                    BNE Rx,R1,return
                    SC R2,X                      /attempt to store Ry
                    BEQZ R2,CAS
                    ADD Ry,R1,R0                     /return X in Ry
                    RET
```

- **IMPLEMENTATION**
  - LL-BIT IS SET WHEN LL IS EXECUTED
  - BUS INTERFACE SNOOPS UPDATE OR INVALIDATE SIGNALS AND RESETS LL-BIT
  - SC TESTS LL-BIT, AND FAILS IF RESET
  - COULD HAVE MULTIPLE LL-BITS FOR DIFFERENT ADDRESSES.

MEMORY COHERENCE: WHAT'S THE PROBLEM



- CAUSED BY MULTIPLE COPIES OF THE SAME DATA
- ASSUME THAT EVENTS 1,2,3,4, AND 5
  - DO NOT OVERLAP IN TIME, OR
  - ARE ATOMIC (TAKE ZERO TIME)
- PROCESSORS P1 AND P2 "SEE" DIFFERENT VALUES OF X AFTER EVENT 3

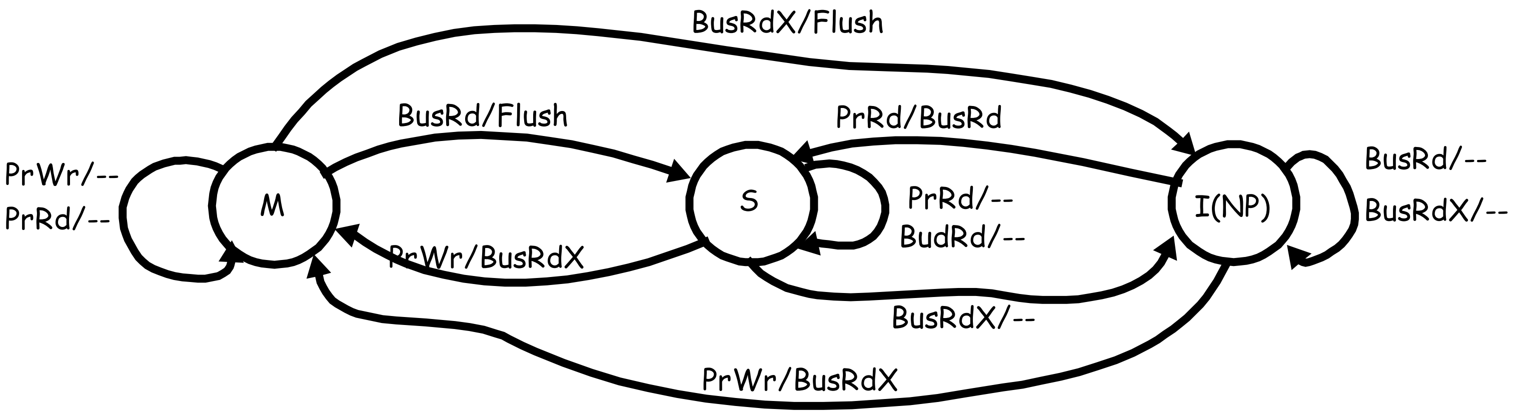
SEEMS SIMPLE TO SOLVE: SIMPLY INFORM COPIES ON EVERY UPDATE

COHERENCE: WHY IT IS SO HARD

- **AFTER EVENT 3, THE CACHES CONTAIN DIFFERENT COPIES. IS THIS STILL COHERENT?**
  - THE ANSWER IS "YES", FOR AS LONG AS P1 OR P2 DO NOT EXECUTE THEIR LOADs
  - A SYSTEM SHOULD REMAIN COHERENT FOR AS LONG AS INCOHERENCE IS NOT DETECTED. OTHERWISE SAME RESULT.
- **CAN THE LOADs IN EVENTS 4 AND 5 RETURN 0 OR 1?COHERENT?**
  - YES, EITHER WOULD STILL BE COHERENT
  - BECAUSE SOFTWARE CANNOT DETECT THE DIFFERENCE
  - P1 OR P2 COULD HAVE RUN SLIGHTLY FASTER, SO THAT EVENTS 4 AND 5 OCCUR BEFORE EVENT 3 IN TIME.
- **IN PRACTICE, MUCH MORE COMPLEX BECAUSE MEMORY EVENTS ARE NOT ATOMIC AND DO SOMETIMES OVERLAP IN TIME**
  - FOR EXAMPLE, EVENTS 3, 4, AND 5 MAY BE TRIGGERED IN THE SAME CLOCK
  - THEY CONFLICT IN SOME PARTS OF THE HARDWARE WHERE THEY ARE SERIALIZED
  - HOWEVER, THEY CAN PROCEED INDEPENDENTLY IN PARALLEL (TEMPORAL OVERLAP) ON THEIR OWN HARDWARE PATHS
- **LET'S ASSUME FIRST THAT COHERENCE TRANSITIONS ARE ATOMIC OR DO NOT OVERLAP IN TIME**
  - HELPS UNDERSTAND HOW TO PROPAGATE VALUES EFFECTIVELY
  - PROTOCOL DESCRIPTION AT THE "BEHAVIORIAL" LEVEL (HOW IT'S SUPPOSED TO BEHAVE; NO IMPLEMENTATION DETAILS)
  - SEE CHAPTER 5 FOR PROTOCOL DESCRIPTIONS

WRITE-BACK: MSI INVALIDATE PROTOCOL

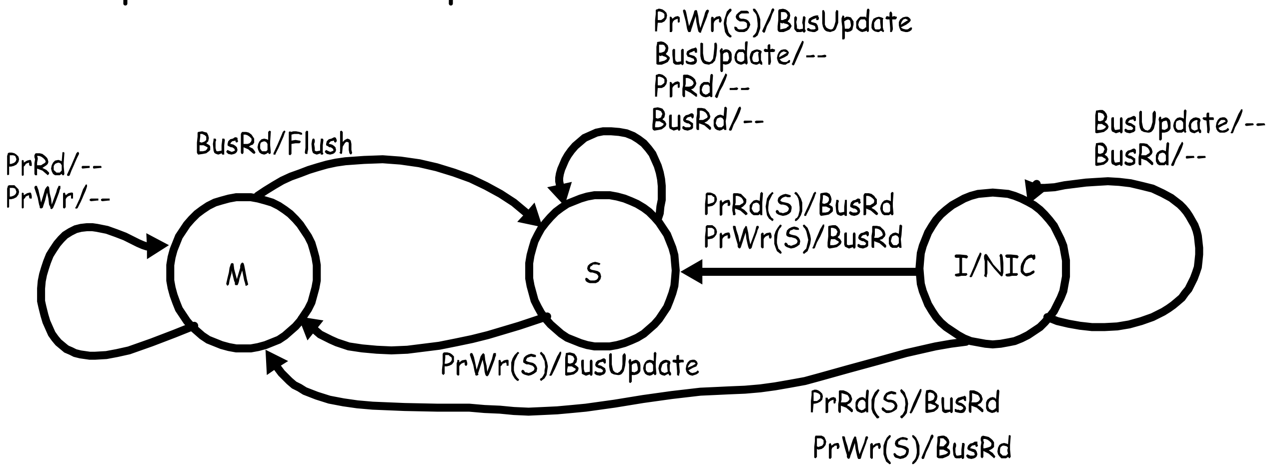
- **BLOCK STATES:**
  - Invalid (I);
  - Shared (S): one copy or more, memory is clean;
  - Dirty (D) or Modified (M): one copy, memory is stale
- **PROCESSOR REQUESTS**
  - PrRd or PrWr
- **BUS TRANSACTIONS**
  - BusRd: requests copy with no intent to modify
  - BusRdX: requests copy with intent to modify



- WRITE TO SHARED BLOCK: COULD USE BusUpgr INSTEAD OF BusRdX
- FLUSH: FORWARD BLOCK COPY TO REQUESTER (MEMORY COPY IS STALE)
  - MEMORY SHOULD BE UPDATED AT THE SAME TIME

WRITE-BACK: MSI UPDATE PROTOCOL

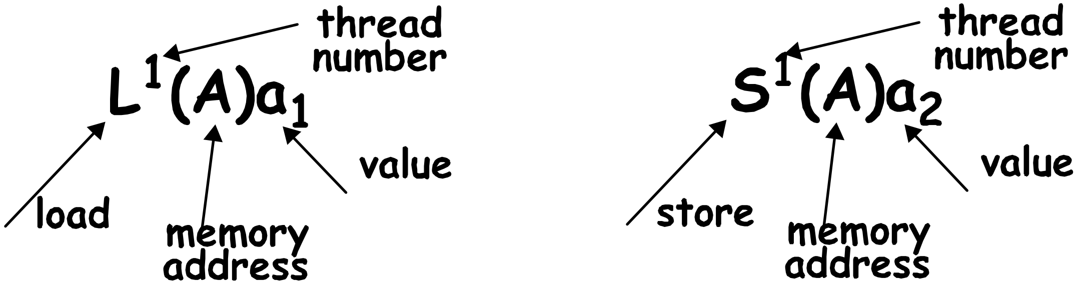
- **BLOCK STATES:**
  - Invalid (I);
  - Shared (S): multiple copies, memory is clean;
  - Dirty (D) or Modified (M): one copy, memory is stale
- **PROCESSOR REQUESTS**
  - PrRd or Pr Wr
- **BUS TRANSACTIONS**
  - BusRd: requests copy
  - BusUpdate: update remote copies



- **SHARED BUS LINE (S): INDICATES WHETHER REMOTE COPIES EXIST**

STRICT COHERENCE

- **STRICT COHERENCE**
  - "A memory system is coherent if the value returned on a Load instruction is always the value given by the latest Store instruction with the same address"
    - SAME AS DEFINITION FOR UNIPROCESSORS
  - DIFFICULT TO EXTEND AS SUCH TO MULTIPROCESSORS
    - EXECUTION TIMES ARE UNPREDICTABLE
    - NO SUPER-FAST COMMUNICATION LINKS BETWEEN PROCESSORS
  - CAN BE APPLIED IF
    - MEMORY ACCESSES DO NOT OVERLAP IN TIME
    - STORES ARE ATOMIC SO THAT ALL COPIES ARE UPDATED INSTANTANEOUSLY
    - STORE/LOAD ORDERS ARE ENFORCED BY ACCESSES TO OTHER MEMORY LOCATIONS (E.G., SYNCHRONIZATION PRIMITIVES)
- **Notations**



$L^1(A)a_1$

thread number

load

memory address

value

$S^1(A)a_2$

thread number

store

memory address

value

LET'S LOOK AT STORE ATOMICITY

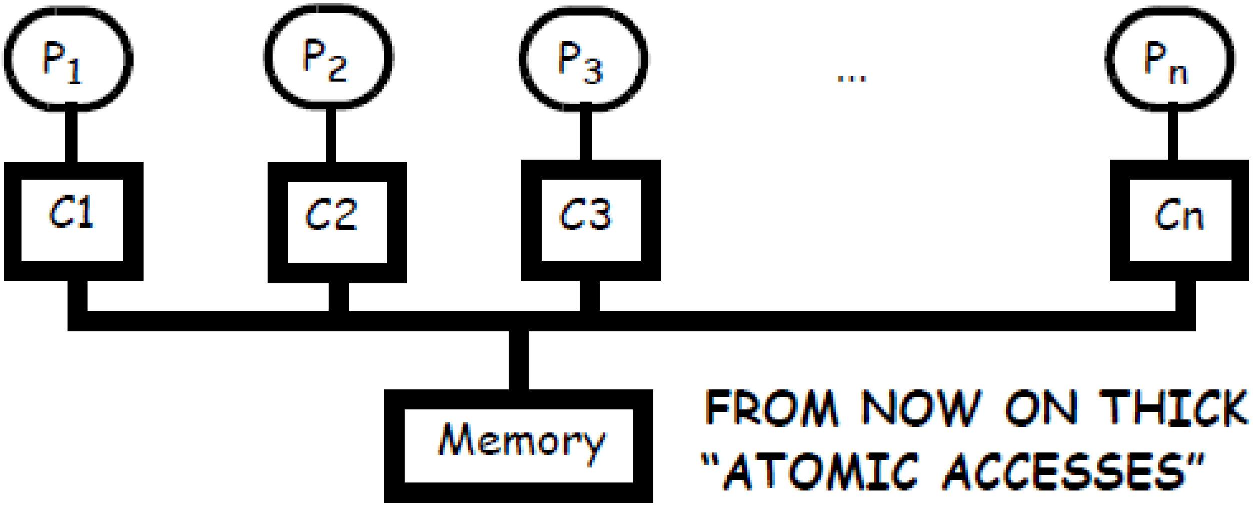
ATOMIC MEMORY ACCESSES

- **Strict coherence is applicable if all memory accesses are atomic or do not overlap in time**
  - Example: MSI invalidate

CLK	T1:	T2:	Comments
t1:	$S^1(A)a_1$		Data not in C2 and
t2:	$L^1(A)a_1$		dirty in C1
t3:	$S^1(A)a_2$		
t4:	$L^1(A)a_2$		
t5:	-----------------------	$L^2(A)a_2$-----------	APT: Read Miss in C2;
t6:	$L^1(A)a_2$		A becomes Shared in C1
			and C2; both threads
t7:		$L^2(A)a_2$	can read A= a_2
t8:	$L^1(A)a_2$		No one can write
t9:	-------------------------	$S^2(A)a_3$---------	APT:C1 is invalidated
t10:		$L^2(A)a_3$	and C2 becomes Dirty
t11:	$S^1(A)a_4$---------------------------------		APT:Store miss in C1;
t13:	$S^1(A)a_5$		C2 is invalidated

MEMORY ACCESS ATOMICITY

- A long time ago, processors were not pipelined, were connected by a single, circuit-switched bus, no store buffer



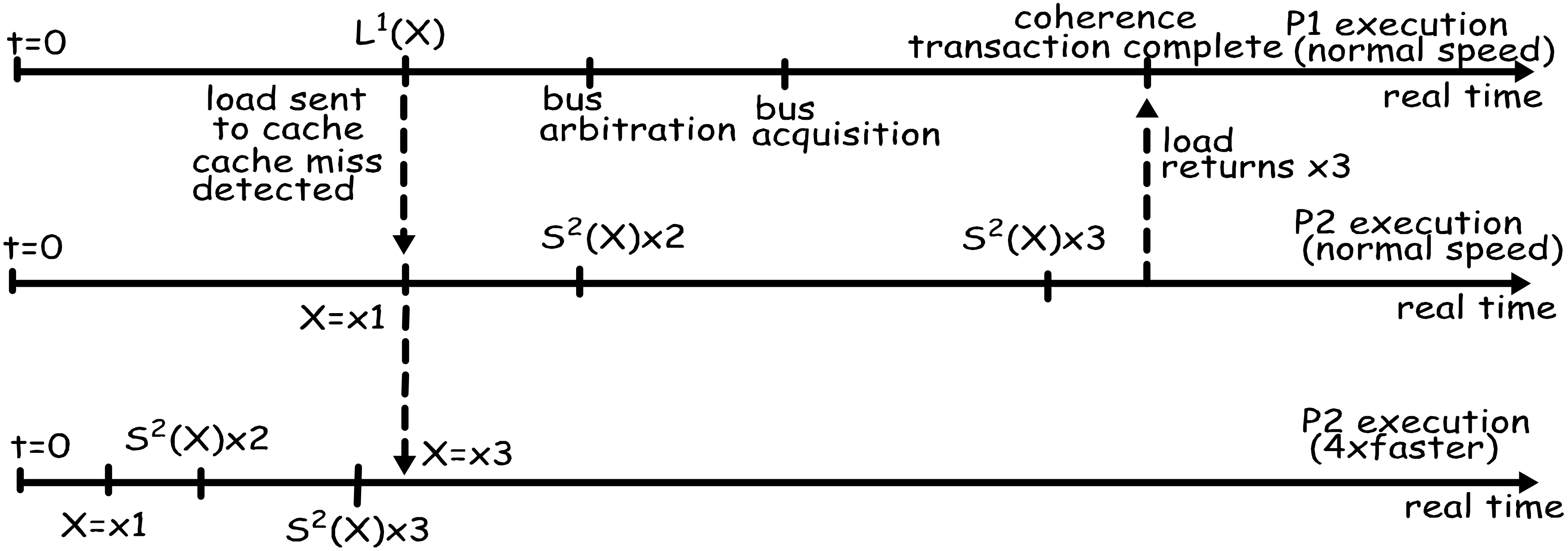
FROM NOW ON THICK LINES MEANS "ATOMIC ACCESSES"

- On a coherence transaction, processor blocks, cache gets access to the bus and complete the transaction in remote caches atomically
  - COHERENCE TRANSACTION DID NOT OVERLAP IN TIME
  - THUS THE PROTOCOL WORKED EXACTLY AS ITS FSM
- THE COHERENCE TRANSACTION IS PERFORMED ATOMICALLY WHEN THE BUS IS RELEASED

Today we must deal with non-atomic transactions

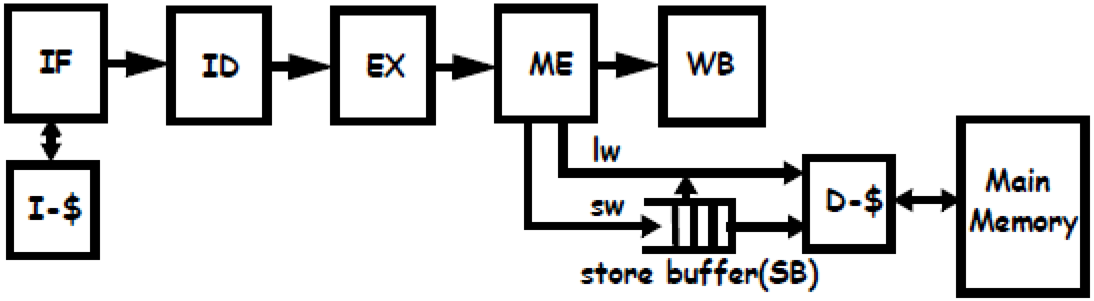
MEMORY ACCESS ATOMICITY

- Assume processor P1 has no copy of X in its cache
- and processor P2 has a Modified (unique) copy of X in its cache in the cycle the load of P1 reaches its cache



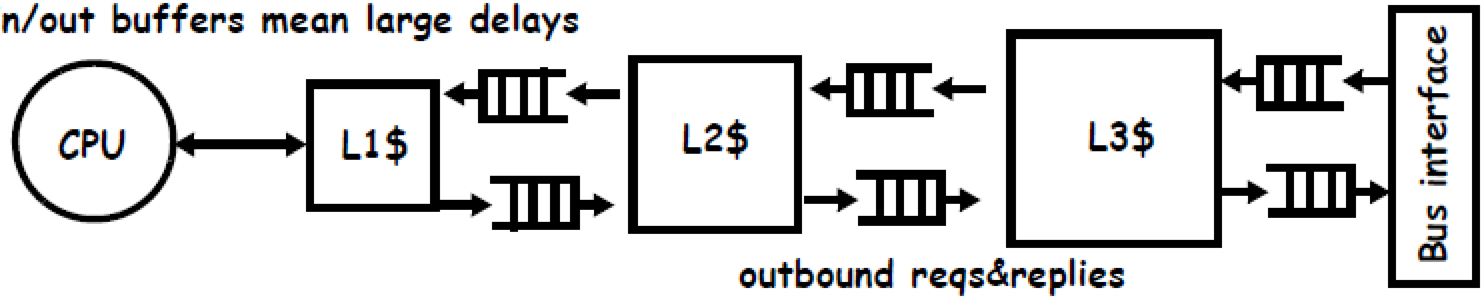
Shared memory program cannot detect the difference between the load returning x1 or the load returning x3, since x3 would be returned if loads were executed instantaneously but P2 was 4 times faster.

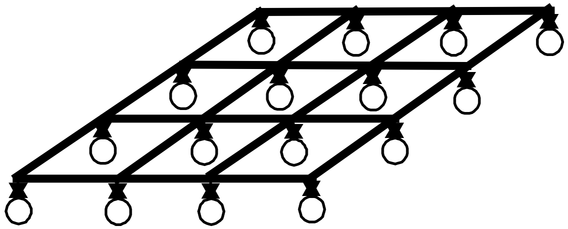
TODAY COHERENCE TRANSACTIONS ARE NON-ATOMIC



Store Buffer
Possibility of
forwarding

in/out buffers mean large delays





point-to-point circuit-switched interconnect

many routes between two points

- adaptive routing

ordered vs. unordered networks

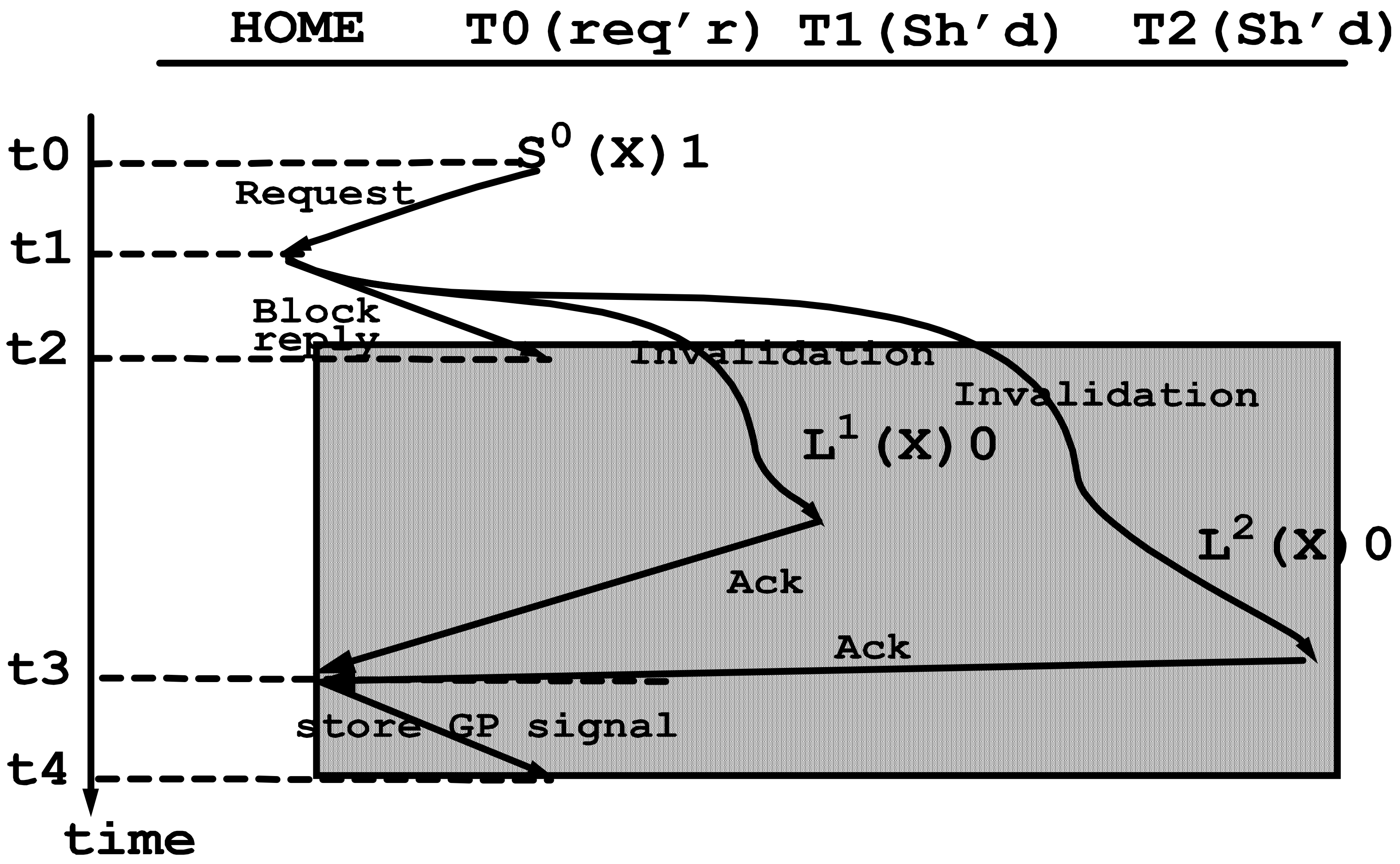
no order + very unpredictable transmission times

MEMORY ACCESS ATOMICITY--SUFFICIENT CONDITION

- **COHERENCE TRANSACTIONS CANNOT HAPPEN INSTANTANEOUSLY**
  - MUST MAKE THEM LOOK ATOMIC TO THE SOFTWARE
  - WELL-KNOWN PROBLEM: DATABASE SYSTEMS, CRITICAL SECTION
- **CONDITION:MAKE SURE THAT ONLY ONE VALUE IS ACCESSIBLE AT ANY ONE TIME**
- **DEFINITIONS**
  - A load is performed at the point in time when its value is bound (atomic) and cannot be recalled
  - A store is performed with respect to thread i at the point in time when a load of thread i cannot return a value prior to the store
  - A store is globally performed when it is performed with respect to all threads
  - A load is globally performed when it is performed and the store providing the value is also globally performed
- **Enforce the following SUFFICIENT conditions (not necessary)**
  - A global order of stores to the same address exists
  - A load must be globally performed before its value can be used
    - value is bound
    - store is globally performed

This second condition means that no thread can observe a new value while any other thread can still observe the old value

STORE ATOMICITY IN cc-NUMAs



WRITE MISS IN THREAD 0 IN A cc-NUMA WITH MSI INVALIDATE

STORE ATOMICITY AND COHERENCE

- **The directory locks entry from t1 to t3**
  - Should home send the block copy to T0 at t1?
  - Should home send it at t3 instead?
- **For store atomicity, T0 cannot return values from its own stores until t4 (nor give them away)**
  - New copy becomes available atomically at t3
  - Store is globally performed at t3
- **For PLAIN coherence, the block can be released to T0 at t2 (as demonstrated below)**

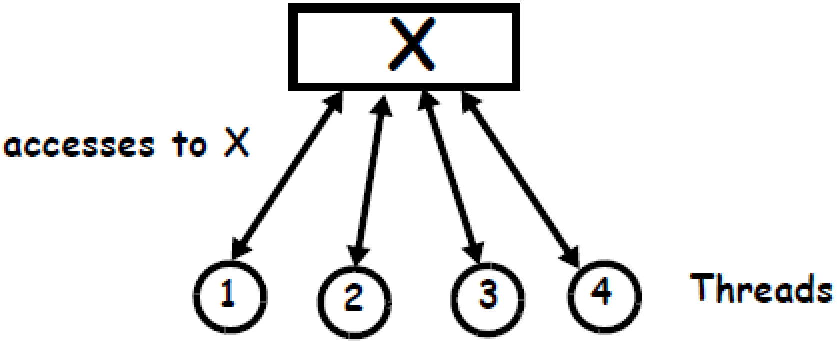
EXERCISE: Consider now MSI Update

- **Threads must all observe the same value**
- **While a new value is propagated, no thread can read the new value (including the writing thread) if other threads can still read the old value**
- **Invalidate protocols facilitate store atomicity**
  - The protocol prevents any other thread from reading the new value until t3, before the requester is allowed to read it (at t4)
- **MSI update (in which invalidates are replaced by updates) needs a second wave of acks**
  - Home must give authorization after receiving all acks

THE CONDITIONS ON STORE ATOMICITY ARE SUFFICIENT, NOT NECESSARY

PLAIN COHERENCE
A WEAKER FORM OF COHERENCE

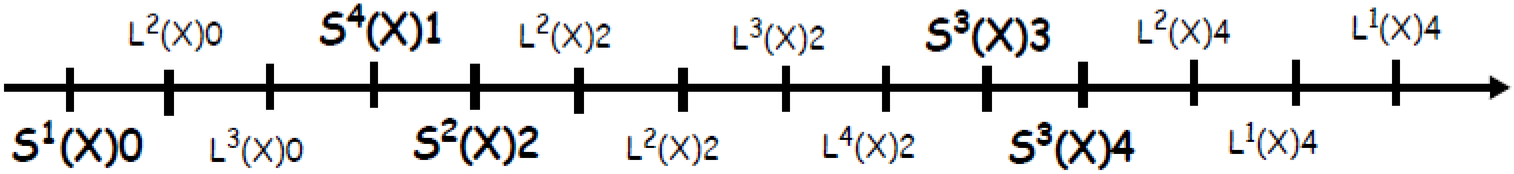
- **ESTABLISH AN ORDER OF ALL ACCESSES TO THE SAME ADDRESS**
- **THEN APPLY THE FUNDAMENTAL DEFINITION OF COHERENCE**
  - HERE "LATEST" IS NOT LATEST IN THE TEMPORAL SENSE
  - IT IS "LATEST" IN THE ORDER OF ALL ACCESSES
- **FORMAL MODEL:**



- **RULES OF THE MODEL:**
  - SINGLE COPY OF EACH DATA
  - ACCESSES ONE BY ONE IN THREAD ORDER TO EACH ADDRESS
- ***A system is coherent if its memory accesses to each address can be executed correctly in thread order in a system with one single copy of each memory address***

COHERENCE: DEFINITION

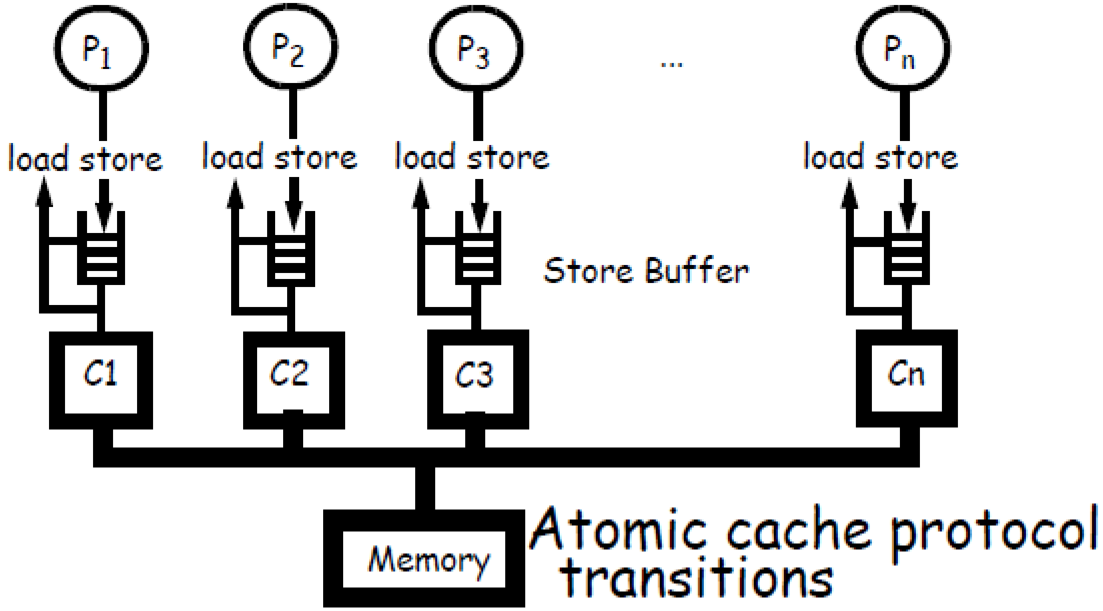
- "A system is coherent iff, for every execution and for any memory location, it is possible to construct a total order of all memory operations to the location such that:
  - memory operations of each thread to the location occur in thread order
  - the value returned by a load is the value of the latest Store to the location in the serial order"



- NOTE: 1) *ACCESSES BY EACH THREAD MUST APPEAR IN THREAD ORDER*
  2) NOT NECESSARILY THE TEMPORAL ORDER
- Since all accesses to every location are in thread order and every load returns the value of the latest store in the order, I can schedule accesses to one address one by one on the formal model and get the same values returned by all loads

FORWARDING STORE BUFFER

- An enlightening example of hardware which is PLAIN coherent but NOT store atomic is that of a store buffer that can forward to loads



- STORES ARE INSERTED IN THE STORE BUFFER AND ISSUED TO CACHE LATER
- LOADS ARE SATISFIED BY STORE BUFFER (IF SAME ADDRESS), OTHERWISE GO TO MEMORY

IMPORTANT: STORE BUFFERS ARE NOT PART OF CACHE COHERENCE

FORWARDING STORE BUFFERS

	T1	T2	T3	CACHE STATES			Comments
				C1	C2	C3	
t0			$L^3(A)a_0$	NIC	NIC	SHA	Miss in C3; APT1
t1	$S^1(A)a_1$			NIC	NIC	SHA	
t2			$S^3(A)a_2$	NIC	NIC	SHA	
t3	$L^1(A)a_1$			NIC	NIC	SHA	
t4		$S^2(A)a_3$		NIC	NIC	SHA	
t5			$L^3(A)a_2$	NIC	NIC	SHA	
t6	$S^1(A)a_4$			NIC	NIC	SHA	
t7		$L^2(A)a_3$		NIC	NIC	SHA	
t8	$L^1(A)a_4$			NIC	NIC	SHA	
t9		$L^2(A)a_3$		NIC	NIC	SHA	
t10	$WB^1(A)a_4$			MOD	NIC	INV	Miss in C1; APT2
t11	$L^1(A)a_4$			MOD	NIC	INV	Hit in C1
t12		$S^2(A)a_5$		DTY	NIC	INV	
t13		$WB^2(A)a_5$		INV	DTY	INV	Miss in C2; APT3
t14	$L^1(A)a_5$			SHA	SHA	INV	Miss in C1; APT4
t15			$L^3(A)a_2$	SHA	SHA	INV	
t16			$WB^3(A)a_2$	INV	INV	MOD	Miss in C3; APT5
t17	$L^1(A)a_2$			SHA	INV	SHA	Miss in C1; APT6
t18			$L^3(A)a_2$	SHA	INV	SHA	Hit in C3
t19		$L^2(A)a_2$		SHA	SHA	SHA	Miss in C2; APT7
t20		$S^2(A)a_6$		SHA	SHA	SHA	
t21		$L^2(A)a_6$		SHA	SHA	SHA	
t22		$WB^2(A)a_6$		INV	MOD	INV	Upgrade in C2; APT8

FORWARDING STORE BUFFERS

- **Aggressive Store Buffer management**
  - Stores overwrite previous values to same address (one single value in SB per address)
  - Stores forwards values to loads
- **We first show the order of accesses to caches**
- **INIT ->$L^3(A)a_0$->$WB^1(A)a_4$->$L^1(A)a_4$->$WB^2(A)a_5$->$L^1(A)a_5$-> $WB^3(A)a_2$->$L^1(A)a_2$-> $L^3(A)a_2$->$L^2(A)a_2$->$WB^2(A)a_6$**

- **We then expand all WBs by local lw/sw's**
- **$L^3(A)a_0$ $S^1(A)a_1$ $L^1(A)a_1$ $S^1(A)a_4$ $L^1(A)a_4$ $L^1(A)a_4$ $S^2(A)a_3$ $L^2(A)a_3$ $L^2(A)a_3$ $S^2(A)a_5$ $L^1(A)a_5$ $S^3(A)a_2$ $L^3(A)a_2$ $L^3(A)a_2$ $L^1(A)a_2$ $L^3(A)a_2$ $L^2(A)a_2$ $S^2(A)a_6$ $L^2(A)a_6$**

Temporal order: a_0-> a_1-> a_2-> a_3-> a_4-> a_5-> a_6

Coherence order: a_0-> a_1-> a_4-> a_3-> a_5-> a_2-> a_6

P1 observes a_1-> a_4-> a_5-> a_2

P2 observes a_3-> a_5-> a_2-> a_6

P3 observes a_0-> a_2

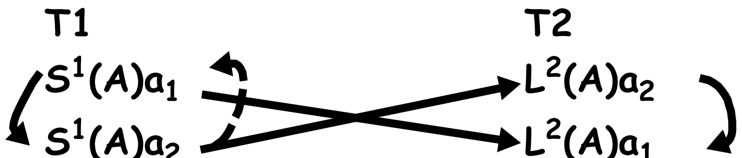
GENERALIZATIONS

- **"PRIVACY PRINCIPLE": A THREAD MAY ACCESS ITS OWN PRIVATE VALUES WHICH ARE NOT PROPAGATED TO OTHER THREADS WITHOUT VIOLATING COHERENCE**
    - REASON IS NO OTHER THREAD CAN OBSERVE THE VALUES, SO IT'S EASY TO INSERT THE ACCESSES TO THEM IN A GLOBAL ORDER

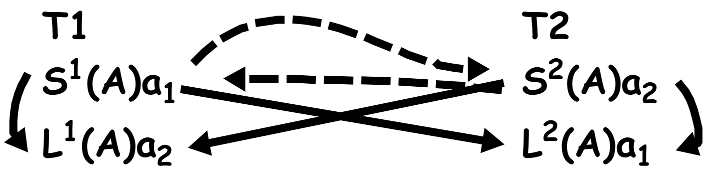
- **THIS RESULT CAN BE GENERALIZED AS FOLLOWS**
    - Other store buffer managements as recombining is the most aggressive one.
    - In lock-up free caches a block can be allocated on a store miss, filled by a store while the miss is pending and the stored value can be read by the thread
    - Threads in multithreaded cores can read each other's values in L1, while a miss is pending
    - Cluster of processors running many threads may share pending values in shared buffers (in hierarchical cache systems) or in shared L2 lockup free caches
    - A thread may modify and use the values returned by a directory protocol even before invalidations are executed (early ack).

IS THERE ANY NON-COHERENT EXECUTION?YES!

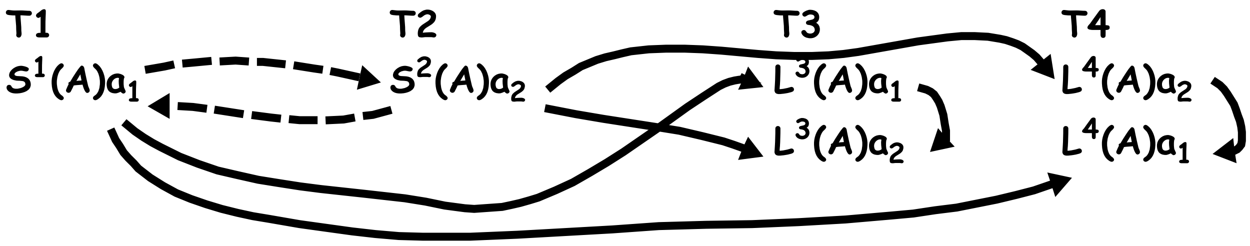
- A THREAD OBSERVES TWO STORES OF ANOTHER THREAD OUT OF ORDER



- THREADS OBSERVE EACH OTHER'S STORES INSTEAD OF OWN



- TWO STORES IN DIFFERENT THREADS ARE NOT ORDERED BUT ARE OBSERVED IN DIFFERENT ORDER BY TWO SEPARATE THREADS



IN THESE THREE CASES, STORES CANNOT BE ORDERED SO THAT THE EXECUTION IS COHERENT

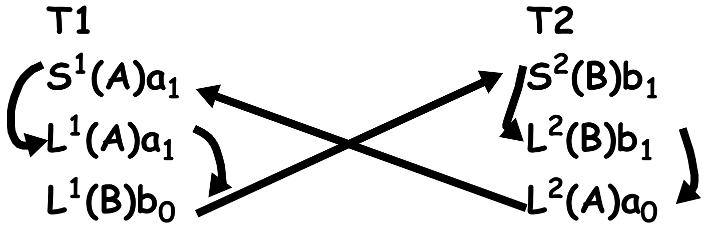
STORES TO THE SAME VARIABLE CANNOT BE EXECUTED AND/OR OBSERVED IN DIFFERENT ORDERS

THE IMPORTANCE OF PLAIN COHERENCE

- **COHERENCE SEEMS TO BE A VERY WEAK PROPERTY.**
- **HOWEVER, COHERENCE IS EXTREMELY USEFUL FOR THE FOLLOWING REASONS:**
  - THERE MUST BE A TOTAL ORDER OF STORES TO THE SAME LOCATION SO THAT TWO THREADS CANNOT OBSERVE THE STORES IN DIFFERENT ORDERS (STORES TO THE SAME LOCATION ARE ORDERED)
  - IT FACILITATES THE IMPLEMENTATION OF MEMORY CONSISTENCY MODELS, BY PROPAGATING VALUES TIMELY AND EFFICIENTLY.
  - IT MAKES SURE THAT, IF A COMPUTATION STOPS SUDDENLY (E.G., A CONTEXT SWITCH) THE MEMORY SYSTEM CONVERGES TO A CONSISTENT STATE FOR ALL DATA, AFTER ALL INSTRUCTIONS IN PROGRESS FINISH AND THE NETWORK AND ALL BUFFERS ARE DRAINED.
  - IT TAKES CARE OF THREAD MIGRATION

THE PROBLEM WITH PLAIN COHERENCE

- COHERENCE IS NOT COMPOSABLE WITH OTHER POSSIBLE ORDERS
  - LOAD-LOAD OR LOAD-STORE ON DIFFERENT LOCATIONS
  - INTRA-THREAD DEPENDENCIES
  - SYNCHRONIZATION (LOCK, BARRIER)
- EXAMPLE INIT $A=a_0;B=b_0;$



T1

$S^1(A)a_1$

$L^1(A)a_1$

$L^1(B)b_0$

T2

$S^2(B)b_1$

$L^2(B)b_1$

$L^2(A)a_0$

- IF THE LOADS IN BOTH THREADS MUST BE ORDERED, THEN IT IS NOT POSSIBLE TO FIND A GLOBAL ORDER OF ALL ACCESSES WHILE MAINTAINING COHERENCE
- WHEN A GLOBAL ORDER DOES NOT EXISTS RESONING ABOUT EXECUTIONS IS MUCH MORE COMPLEX
- THIS IS BECAUSE THE COHERENCE ORDER IS NOT NECESSARILY THE SAME AS THE TEMPORAL ORDER

NOT THE CASE WITH STORE ATOMICITY

COHERENCE IS NOT SUFFICIENT

- Point-to-point Synchronization
  - ASSUME A AND flag ARE BOTH 0 INITIALLY

```
P1                              P2
...                                 ...
A:=1;                         while(flag==0)do nothing;
flag:=1;                      print A;
...                           ...
```

- Communication
  - ASSUME A AND B ARE BOTH 0 INITIALLY

```
P1                            P2
...                           ...
A:=1;                         print B;
B:=2;                         print A;
...                           ...
```

- Dekker's Algorithm (critical section)
  - ASSUME A AND B ARE BOTH 0 INITIALLY

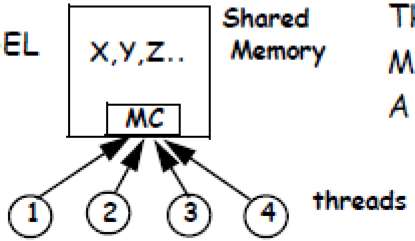
```
P1                            P2
...                           ...
$S^1$(A):A:=1                 $S^2$(B):B:=1
 $L^1$(B):while(B==1);        $L^2$(A): while(A==1);
<critical section>            <critical section>
A:=0                          B:=0
```

- PROGRAMMER'S INTUITION HERE IS THAT ACCESSES FROM DIFFERENT PROCESSES ARE "INTERLEAVED" IN PROCESS ORDER
  - DIFFERENT FROM COHERENCE, WHICH APPLIES TO A SINGLE LOCATION

SEQUENTIAL CONSISTENCY

FORMAL MODEL FOR SEQUENTIAL CONSISTENCY

SHARED MEMORY MODEL FOR EVERY MEMORY LOCATIONS X,Y,Z



THREADS EXECUTE MEMORY ACCESSES ONE AT A TIME IN PROCESS ORDER

- **PROGRAM ORDER OF ALL THREADs IS RESPECTED AND ALL MEMORY ACCESSES ARE ATOMIC BECAUSE OF THE MEMORY CONTROLLER**
  - "A MULTIPROCESSOR IS SEQUENTIALLY CONSISTENT IF THE <u>RESULT</u> OF <u>ANY</u> EXECUTION IS THE SAME <u>AS IF</u> THE MEMORY OPERATIONS OF ALL THE PROCESSORS WERE EXECUTED IN <u>SOME</u> SEQUENTIAL ORDER, AND THE OPERATIONS OF EACH INDIVIDUAL PROCESSOR APPEAR IN THE SEQUENCE IN THE ORDER SPECIFIED BY ITS PROGRAM"
- **"..<u>ANY</u> EXECUTION, <u>AS IF</u>.... IN <u>SOME</u> SEQUENTIAL ORDER". LET'S LOOK AT "AS IF"**

  A AND B ARE 0 INITIALLY

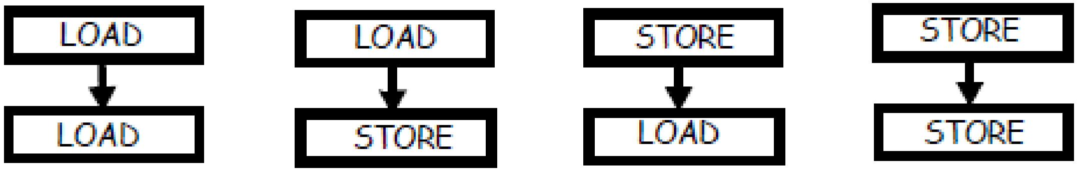
```
P1              P2
A:=1;           PRINT B;
B:=2;           PRINT A;
```

  - POSSIBLE PRINTED OUTCOMES UNDER SC: (A,B) = (0,0), (1,0),(1,2)
  - IMPOSSIBLE OUTCOME UNDER SC: (0,2)
  - LOOK AT EXECUTION B:=2 => A:=1 => PRINT A => PRINT B
    - SAME RESULTS AS A:=1 => B:=2 => PRINT B => PRINT A
    - IT IS SC, EVEN IF THE TEMPORAL ORDER VIOLATES PROCESS ORDER

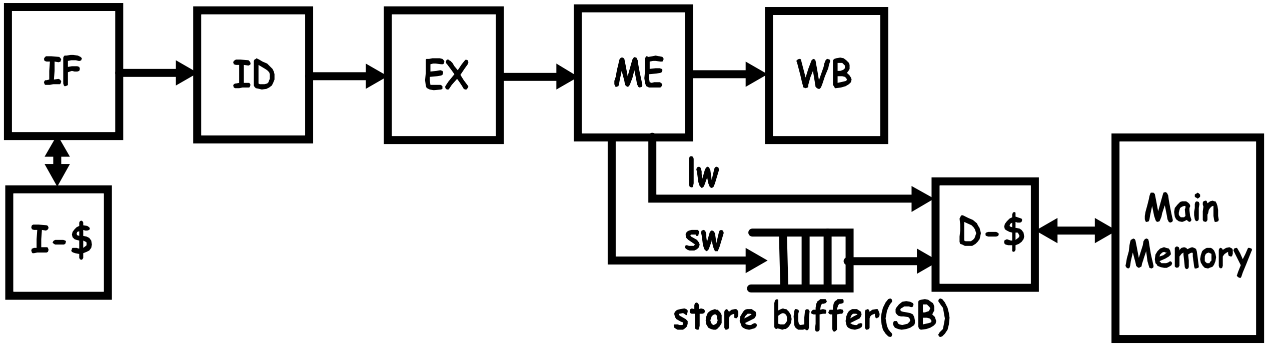
AS FOR COHERENCE, SC ORDER IS NOT NECESSARILY TEMPORAL ORDER

SEQUENTIAL CONSISTENCY

- LOOK AT MEMORY ACCESSES ORDERS THAT MUST BE GLOBALLY PERFORMED BY EACH PROCESSOR OR ENFORCED IN THE TOTAL ORDER OF ALL ACCESSES



- IN SC ALL ORDERS MUST BE ENFORCED
- ASSUME MEMORY IS ATOMIC: ONLY GP LOADs CAN RETURN VALUES
- WHAT DOES THIS MEAN FOR IO PROCESSORS?



- LOADs ARE BLOCKING, SO LOAD-LOAD AND LOAD-STORE ORDERS ARE ENFORCED
- STOREs ARE NON-BLOCKING (THEY MOVE TO SB)
  - STORE-LOAD: LOADs MUST BLOCK IN ME UNTIL SB IS EMPTY
  - STORE-STORE: STOREs MUST BE GPed ONE BY ONE FROM SB

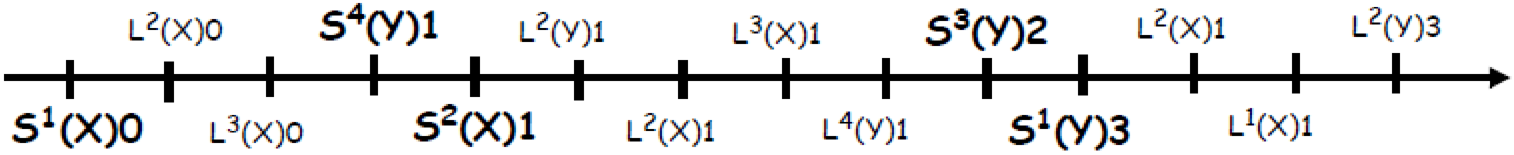
CONDITIONS FOR SC

- WE NEED TO FIND A TOTAL COHERENT ORDER OF ALL MEMORY ACCESSES IN WHICH ACCESSES OF EACH THREAD ARE IN THREAD ORDER FOR EVERY EXECUTION (LIKE COHERENCE EXCEPT THAT NOW IT'S FOR ALL MEMORY LOCATIONS, NOT JUST ONE)
  - SIMPLY SCHEDULE EACH ACCESS ONE AT A TIME ON THE FORMAL MODEL

SUFFICIENT CONDITION

  - EVERY PROCESSOR GLOBALLY PERFORMS ITS MEMORY ACCESSES IN THREAD ORDER

- IN THE SC ORDER:
  - ACCESSES TO ALL LOCATIONS FROM EACH THREAD MUST BE IN THREAD ORDER (T.O.)
  - SOURCE (STORE) OF VALUE MUST PRECEDE THE LOAD OF THE VALUE



  - PLUS IMPLIED RULES (on the same location), TO ENFORCE COHERENCE:
    - THERE CAN BE NO STORE BETWEEN THE STORE SOURCING THE VALUE AND THE LOAD READING THE VALUE.

SEQUENTIAL CONSISTENCY

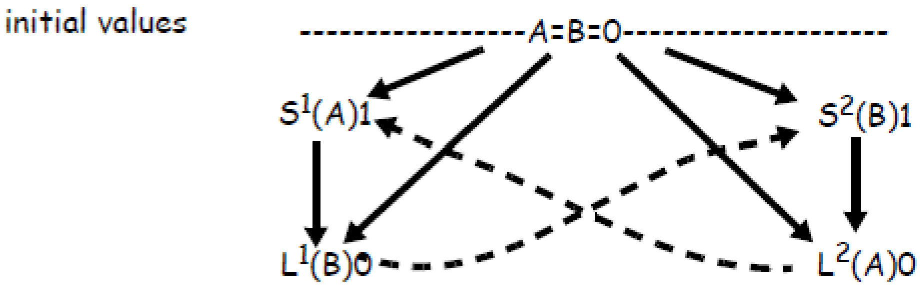
- **PROBLEM WITH STORE BUFFER: WITH STORE BUFFERS, LOADS CAN BE GPed BEFORE PREVIOUS STORES**
  - IN SC, A LOAD MUST BE STALLED IF PRIOR STORES ARE NOT GPed
  - EFFECTIVE FOR LONG BURSTS OF STORES
    - BUT STORES MUST BE PROPAGATED ONE BY ONE BEFORE THE NEXT LOAD ANYWAY

- **LOOK AT DEKKER AGAIN**

A and B are both 0 and cached as Shared initially

T1	T2
$S^1(A)1$	$S^2(B)1$
$L^1(B)0$	$L^2(A)0$

  - THIS IS NOT AN SC EXECUTION



  - EXECUTION GRAPH HAS CYCLES → EXECUTION CANNOT BE ORDERED
    - With store buffers the outcome can be (0,0)
    - T1 executes A:=1, which stays in the SB
    - T2 executes B:=1, which stays in the SB
    - Under MSI invalidate T1 and T2 both read 0

CONCLUSION ON SC

- **PROBLEM WITH STORE BUFFERS**
  - SC CANNOT REALLY TAKE ADVANTAGE OF STORE BUFFERS

- **PROBLEM WITH COMPILERS**
  - TAKE AGAIN PT-TO-PT SYNCHRONIZATION

```
P1              P2
...             ...
A:=1;           while(flag==0)do nothing;
flag:=1;        print A;
...             ...
```

  - SINCE THERE IS NO DEPENDENCY BETWEEN flag AND A, THE COMPILER MAY REORDER THE TWO INSTRUCTIONS IN P1
  - MIGHT ALSO REMOVE THE LOOP ON flag IN P2

NEXT IDEA: CHANGE THE RULES

MEMORY CONSISTENCY MODELS

MEMORY CONSISTENCY MODELS

- **ARE THERE OTHER PROGRAMMER'S INTUITIONS?**
  - SUCH AS THOSE PROVIDED BY SYNCHRONIZATION?
  - FOR EXAMPLE, DEKKER'S ALGORITHM IS NOT USED IN PRACTICE
    - gets complex for more than 2 processors and to solve the deadlock
  - WHENEVER SHARED VARIABLES ARE READ/WRITE THEY SHOULD BE PROTECTED BY LOCKS

```
BAR IS 0 INITIALLY
P1                      P2
A:=1;                   BARRIER(BAR,2);
B:=2;                   R1:=A;
BARRIER(BAR,2);         R2:=B;
```

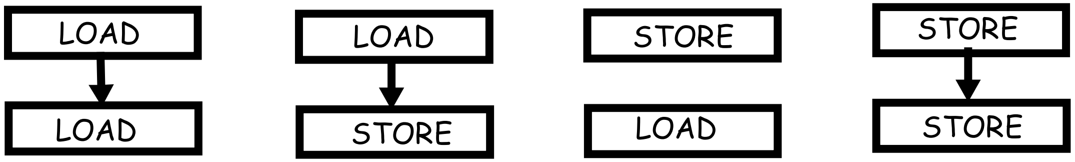
Because of the barrier, the result is always SC and P2 returns (1,2)

- **NEXT: WHY SHOULD WE BOTHER ABOUT THE PROGRAMMER ANYWAY?**
  - WHY COULDN'T WE DESIGN MEMORY ACCESS RULES THAT ARE HARDWARE FRIENDLY AND THEN CONSTRAIN THE PROGRAMMER??

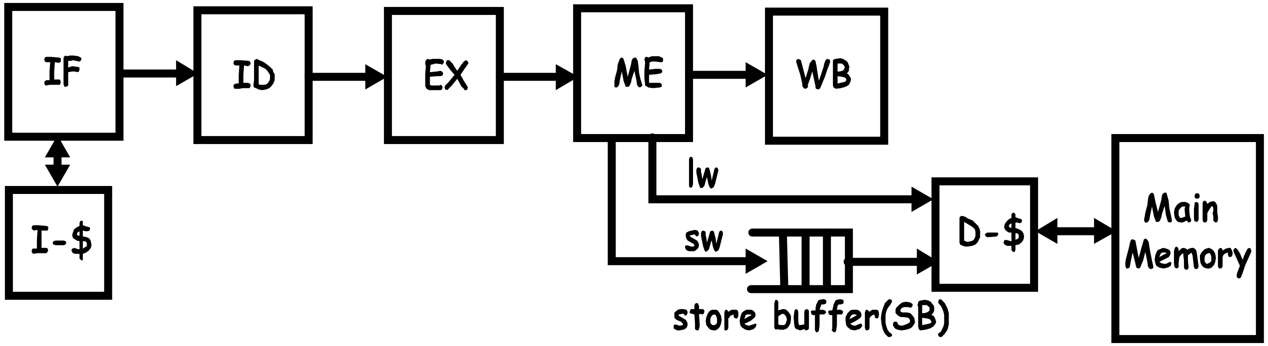
- **IN ANY CASE WE NEED A MEMORY ACCESS ORDERING MODEL ON WHICH PROGRAMMERS AND MACHINE ARCHITECTS CAN AGREE**
  - CALLED THE MEMORY CONSISTENCY MODEL
- **THIS MODEL MUST BE PART OF THE ISA DEFINITION, SINCE IT IS AT THE INTERFACE BETWEEN SOFTWARE AND HARDWARE**
  - TODAY'S INSTRUCTION SET MANUALS INCLUDE THE MEMORY CONSISTENCY MODEL AS PART OF THE ISA DEFINITION

RELAXED MEMORY CONSISTENCY MODELS

- WE CAN RELAX SOME OF THE ACCESS ORDERS OF EACH THREAD
- THE MAJOR RELAXATION IS THE STORE-TO-LOAD ORDER: LOADS CAN BYPASS PRIOR STORES



  - STORES FROM THE SAME PROCESSOR MUST BE OBSERVED BY ALL OTHER PROCESSORS IN THREAD ORDER (BECAUSE OF STORE-STORE AND LOAD-LOAD ORDERS)
- WHAT DOES THIS MEAN FOR IO PROCESSORS?
  - ASSUME MEMORY IS ATOMIC: LOADS RETURN VALUES FROM MEMORY ONLY WHEN THE VALUES ARE GPed



  - LOAD-LOAD, LOAD-STORE, STORE-STORE: SAME AS FOR SC
  - STORE-LOAD: LOADs DON'T WAIT FOR SB EMPTY;

RELAXING STORE-TO-LOAD ORDERS

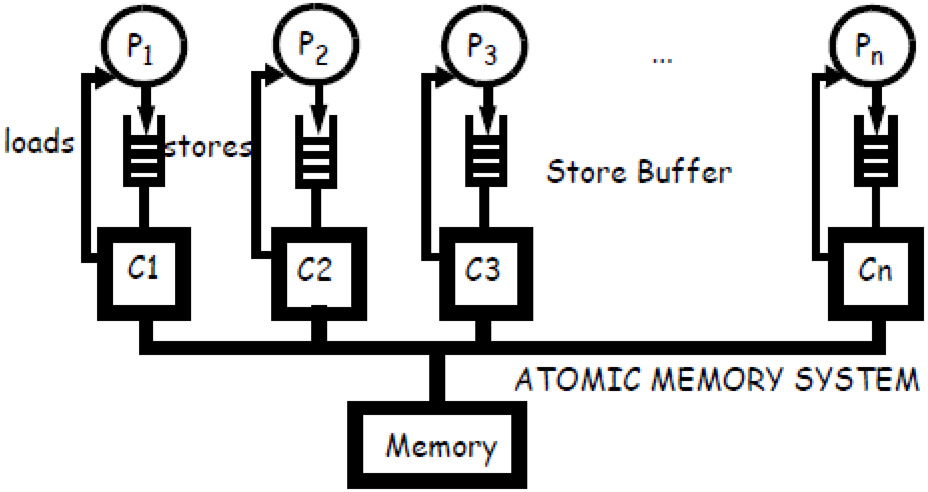
- **EXAMPLE: SUN MICRO TOTAL STORE ORDER (TSO)**
  - DEKKER'S ALGORITHM DOES NOT WORK
  - POINT-TO-POINT COMMUNICATION STILL WORKS

- **COMBINING OF STORES IN THE STORE BUFFER CAN ONLY BE DONE IF THE STORES ARE NOT SEPARATED BY STORES TO OTHER ADDRESSES**

- **ACCESSES TO CACHE ARE SUBMITTED OUT OF PROGRAM ORDER**

- **VALUES MAY OR MAY NOT BE FORWARDED FROM SB TO LOADS**
  - NO FORWARDING FROM SB → STORE ATOMIC
  - FORWARDING FROM SB → PLAIN COHERENT (CASE OF TSO)
    - THIS MEANS THAT SOME LOADS IN TSO RETURN VALUES EVEN IF THEY ARE NOT GPed

NO FORWARDING FROM SB



IMPORTANT NOTE: THE STORE BUFFER IN THE MODEL DOES NOT HAVE TO BE A PHYSICAL STORE BUFFER. THE SB IN THE MODEL REPRESENTS THE PRIVATE STORE PIPELINE OF THE CORE OR THREAD.(EXAMPLE: LOCKUP-FREE CACHE)

- ATOMIC STORES. LOADS ONLY RETURN GPed VALUES
- VALID EXECUTION (NOT VALID IN SC)

INIT: A=B=0

T1	T2
$S^1(A)1$	$S^2(B)1$
$L^1(B)0$	$L^2(A)0$

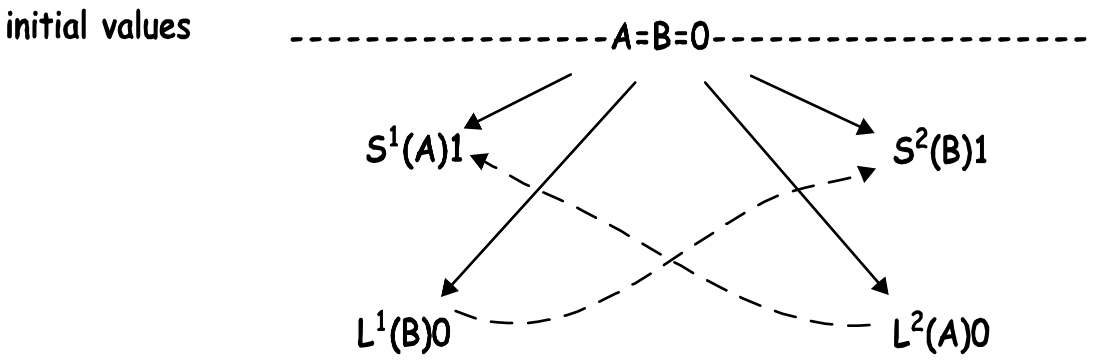
THEREFORE DEKKER'S ALGORITHM DOES NOT WORK

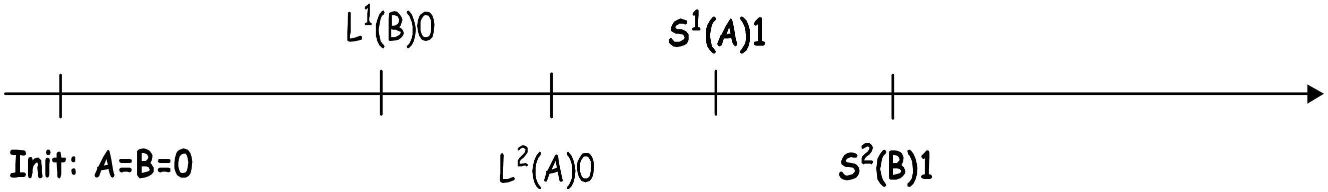
NO FORWARDING FROM SB

- VALID EXECUTION

INIT: A=B=0

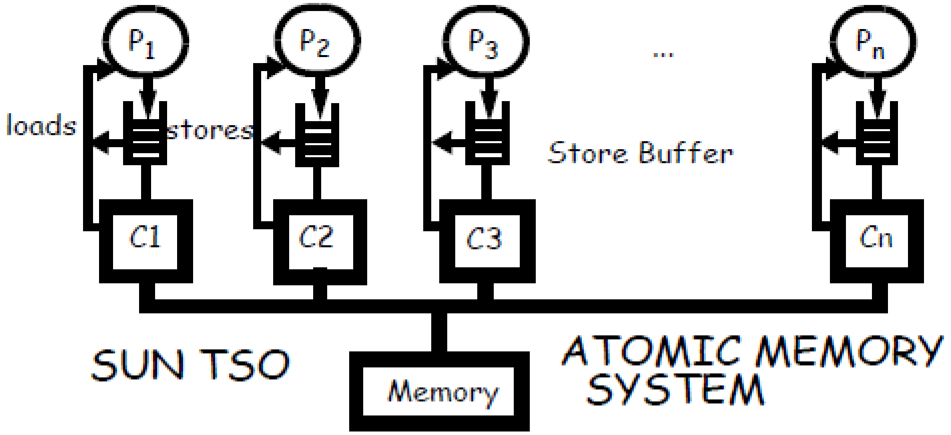
T1	T2
$S^1(A)1$	$S^2(B)1$
$L^1(B)0$	$L^2(A)0$





- NO ARROW BETWEEN STORE AND LOAD IN EACH THREAD
- NO GLOBAL ORDER BETWEEN THE STORE AND LOAD IN EACH THREAD
- STILL STORE ATOMIC OVERALL → GLOBAL ORDER IS COHERENT

FORWARDING FROM SB (TSO)



NOTE AGAIN THAT THIS MODEL IS AN ABSTRACT VIEW OF THE MEMORY BEHAVIOR

- THE STORE BUFFER IN THE MODEL IS THE "STORE PIPELINE" OF MEMORY ACCESSES BY EACH PROCESSOR
- AS LONG AS A STORE CAN ONLY BE OBSERVED BY ITS OWN THREAD IT DOES NOT NEED TO PROPAGATE TO OTHERS
- AS SOON AS IT CAN BE OBSERVED BY OTHER THREADS, IT MUST BECOME VISIBLE TO ALL, ATOMICALLY
- FOR EXAMPLE, IF L1 IS PRIVATE TO A THREAD AND IS LOCKUP FREE, THEN THE THREAD CAN RETURN VALUES FROM L1 THAT ARE NOT GPed
- BUT IF L1 IS SHARED BY SEVERAL THREADS, L1 MAY NOT RETURN A NON-GP VALUE.

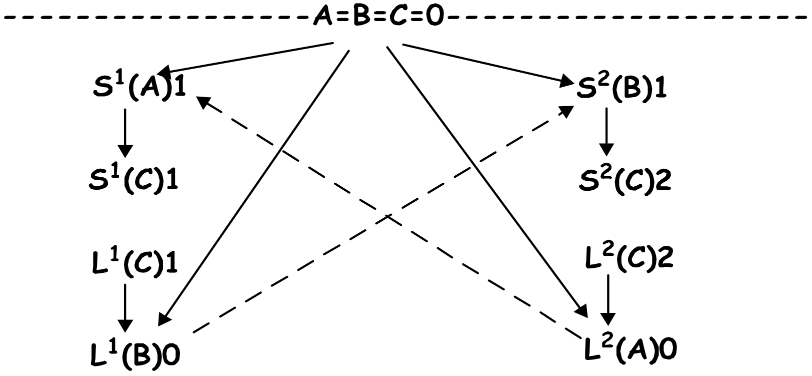
FORWARDING FROM SB (TSO)

- CODE CORRECT UNDER FORWARDING BUT NOT WITHOUT FORWARDING

INIT: A=B=C=0

T1	T2
$S^1(A)1$	$S^2(B)1$
$S^1(C)1$	$S^2(C)2$
$L^1(C)1$	$L^2(C)2$
$L^1(B)0$	$L^2(A)0$

- IN THIS CODE IS THERE STILL A POSSIBLE COHERENT ORDER OF ALL ACCESSES? ANSWER: NO
- THE EXECUTION IS NOT STORE ATOMIC, BUT IT IS STILL PLAIN COHERENT (CONSIDER EACH ADDRESS SEPARATELY)
-



-----------------A=B=C=0--------------------

$S^1(A)1$

$S^1(C)1$

$L^1(C)1$

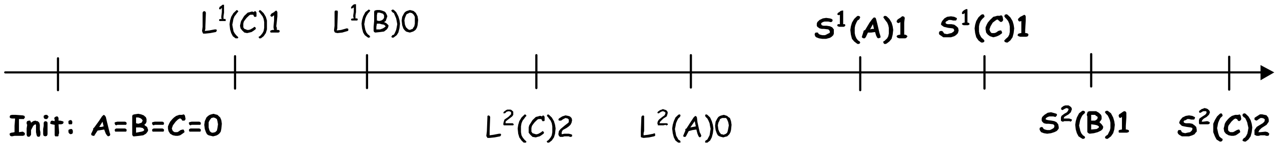
$L^1(B)0$

$S^2(B)1$

$S^2(C)2$

$L^2(C)2$

$L^2(A)0$



Init: A=B=C=0

$L^1(C)1$

$L^1(B)0$

$L^2(C)2$

$L^2(A)0$

$S^1(A)1$

$S^1(C)1$

$S^2(B)1$

$S^2(C)2$

WITH FORWARDING (OR UNDER SC) THERE WOULD BE AN ARROW BETWEEN STORE AND LOAD IN EACH THREAD AND THIS EXECUTION WOULD NOT BE POSSIBLE

SUN MICRO RELAXED MEMORY ORDER (RMO)

- **IN RMO ONLY INTRA-THREAD MEMORY DEPENDENCY ORDER IS ENFORCED**
  - AS IN ALL UNIPROCESSORS
- **NO IMPLICIT ORDER BETWEEN THREADS**

- **MEMBAR INSTRUCTIONS SPECIFY ORDERS BETWEEN THREADS EXPLICITELY**
  - 4 BITS ARE USED TO SPECIFY UP TO 4 ORDERS
  - LAOD-LOAD ➔ FENCE FORCING ALL PRECEDING LOADS TO BE GPed BEFORE ANY LOAD MAY BE ISSUED
  - LOAD-STORE ➔FORCES ALL PRECEDING LOADS TO BE GP BEFORE ANY STORE
  - STORE-STORE ➔ FORCES ALL PRECEDING STORES TO BE GP BEFORE ANY STORE
  - STORE-LOAD ➔ FORCES ALL PRECEDING STORES TO BE GP BEFORE ANY LOAD
- **MEMBARS ARE INSERTED BY THE COMPILER OR PROGRAMMER**
  - MEMBARs ARE EXTRA INSTRUCTIONS

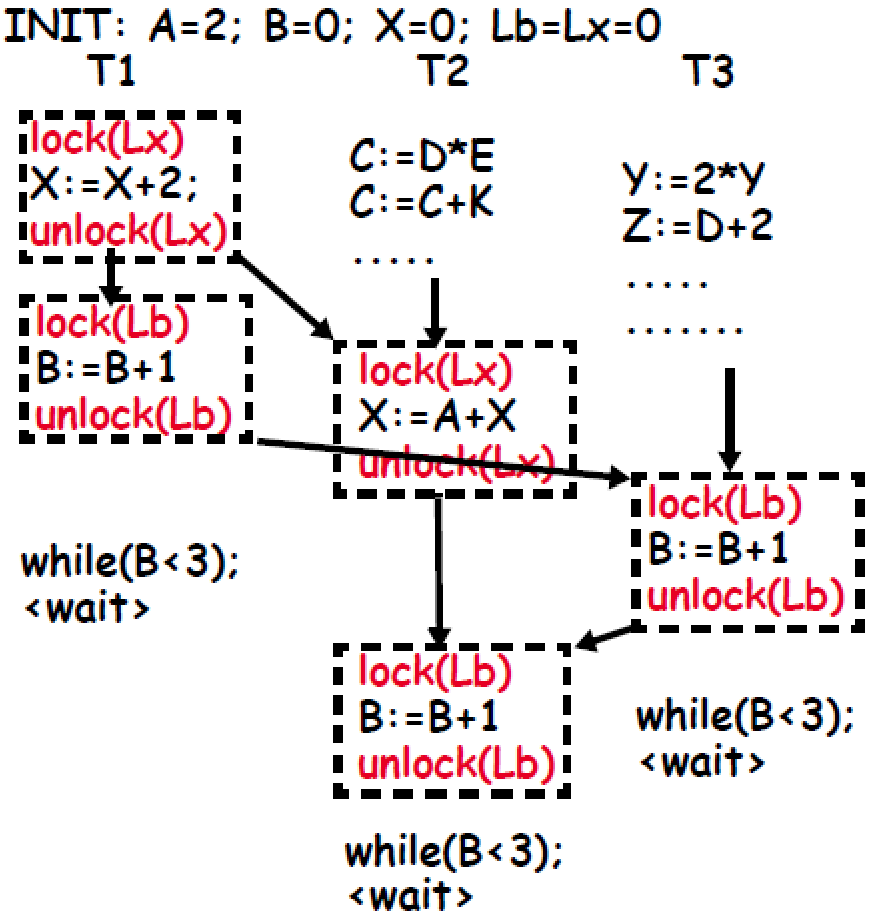
```
T1                 T2                  T3
A:=1;              while(A==0);        R2:=B;
                   B:=1;               R3:=A;
```

- **NOT SEQUENTIALLY CONSISTENT (YIELDS NON-SC OUTCOMES)**

```
T1                 T2                  T3
A:=1;              while(A==0);        R2:=B;
                   MEMBAR 0100         MEMBAR 100
                   B:=1;               R3:=A;
```

- **WITH MEMBARS: SEQUENTIALLY CONSISTENT (YIELDS SC OUTCOMES ONLY)**

MCM USING SYNCHRONIZATION



INIT: A=2; B=0; X=0; Lb=Lx=0

```
T1                  T2              T3

lock(Lx)            C:=D*E          Y:=2*Y
X:=X+2;             C:=C+K          Z:=D+2
unlock(Lx)          .....           .....
                                    .......
lock(Lb)            lock(Lx)
B:=B+1              X:=A+X
unlock(Lb)          unlock(Lx)
                                    lock(Lb)
while(B<3);                         B:=B+1
<wait>                              unlock(Lb)

                    lock(Lb)
                    B:=B+1          while(B<3);
                    unlock(Lb)      <wait>

                    while(B<3);
                    <wait>
```

Locks use special instructions

- RMW (for lock)
- special stores (for unlocks)

Treat locks as FENCES for all memory accesses

Also called "SYNC" in PowerPC's memory model

lock and unlock act as fences

- perform all preceeding accesses in t.o. before attempting a SYNC
- perform SYNC beform attempting any following accesses in t.o.

In between two SYNC Ops, order of memory Ops is arbitrary

WEAK ORDERING

- MULTITHREADED EXECUTION USES LOCKING MECHANISMS TO AVOID RACE CONDITIONS.
- EXECUTIONS INCLUDE VARIOUS PHASES:
  - ACCESSES TO PRIVATE OR READ-ONLY SHARED DATA, OR
  - ACCESSES TO SHARED MODIFIABLE DATA, PROTECTED BY LOCKS AND BARRIERS.
- IN EACH PHASE THE THREAD HAS EXCLUSIVE ACCESS TO ALL ITS DATA, MEANING
  - NO OTHER THREAD CAN WRITE TO THEM, OR
  - NO OTHER THREAD CAN READ THE DATA IT MODIFIES
- ACCESSES TO SYNCHRONIZATION DATA (INCLUDING ALL LOCKS AND SHARED DATA IN SYNCHRONIZATION PROTOCOLS) ARE TREATED DIFFERENTLY BY THE HARDWARE FROM ACCESSES TO OTHER SHARED AND PRIVATE DATA.
  - THEY ACT AS FENCES ON ALL ACCESSES
  - MUST GLOBALLY PERFORM ALL ACCESS PRECEDING SYNC ACCESS IN T.O.
  - MUST GLOBALLY PERFORM SYNC ACCESS BEFORE ALL FOLLOWING ACCESSES IN T.O.

WEAK ORDERING

- **ACCESSES TO OTHER (NON-SYNC) SHARED AND PRIVATE DATA MUST ENFORCE UNIPROCESSOR DEPENDENCIES ON SAME ADDRESS**
  - OTHERWISE, NO REQUIREMENT
  - EVEN COHERENCE IS NOT A REQUIREMENT

- **VARIABLES THAT ARE USED FOR SYNCHRONIZATION MUST BE DECLARED AS SUCH (e.g., flag, A and B below) OR SPECIFIC STATEMENTS MUST BE LABELLED OR MARKED**

  - SO THAT EXECUTION ON THESE VARIABLES ARE SAFE.

```
A=flag=0 initially
T1                                T2
A:=1;                             while(flag==0)do nothing;
flag:=1;                          print A;
...                               ...
```

Flag MUST BE DECLARED AS SYNC VARIABLE

```
A=B=0 initially
T1                                T2
A:=1                              B:=1
while(B==1);                      while(A==1);
<critical section>                <critical section>
A:=0                              B:=0
```

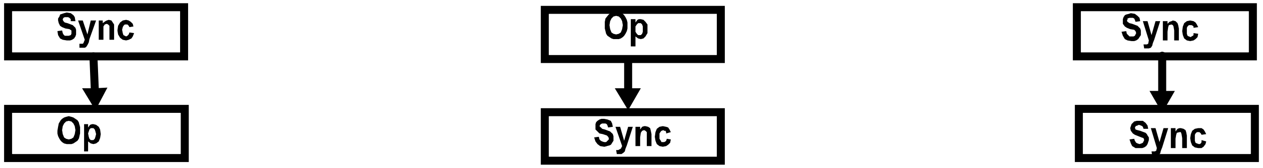
A AND B MUST BE DECLARED AS SYNC VARIABLES

WEAK ORDERING

- **A RMW ACCESS TO A MEMORY LOCATION IS GLOBALLY PERFORMED ONCE BOTH THE LOAD AND STORE IN THE RMW ACCESS ARE GLOBALLY PERFORMED.**
  - ATOMICITY MUST ALSO BE ENFORCED BETWEEN THE TWO ACCESSES (EASIER IN WRITE-INVALIDATE PROTOCOLS--TREAT THE T&S AS A STORE IN THE PROTOCOL)

- **SYNC OPERATION MUST BE RECOGNIZABLE BY THE HARDWARE AT THE ISA LEVEL**
  - RMW (T&S)
  - SPECIAL LOADS AND STORES FOR SYNC VARIABLE ACCESSES

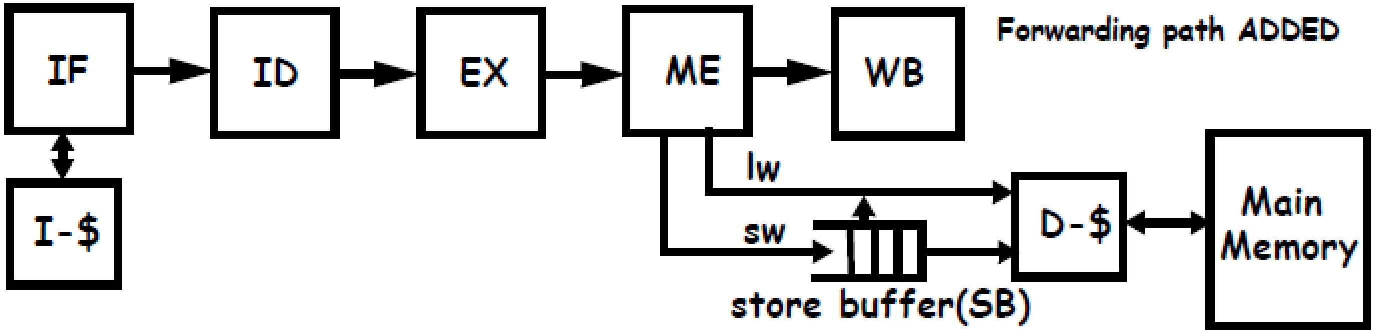
- **ORDERS TO ENFORCE:**



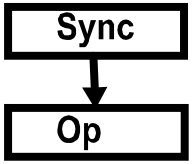
- **Op = regular load or store**
- **Sync = any synchronization access, e.g., swap, T&S, special load/store**

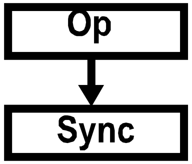
WEAK ORDERING

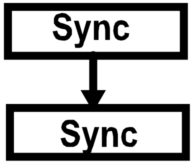
- WHAT DOES IT MEAN FOR IO PROCESSORS?
  - NOTE: HERE LOADS CAN RETURN VALUES EVEN IF THEY ARE NOT GPed



- REGULAR STORES IN THE STORE BUFFER CAN BE EXECUTED IN ANY ORDER, IN PARALLEL
- REGULAR LOADS NEVER WAIT FOR STORES AND CAN BE FORWARDED TO
- WHEN A SYNC ACCESS IS EXECUTED, IT IS TREATED DIFFERENTLY:
  - IT BLOCKS IN THE MEMORY STAGE UNTIL ALL STORES IN THE STORE BUFFER ARE GLOBALLY PERFORMED WHICH ENFORCES OP-TO-SYNC.
  - SYNC-TO-OP AND SYNC-TO-SYNC ORDERS ARE AUTOMATICALLY ENFORCED BY IO PROCESSOR

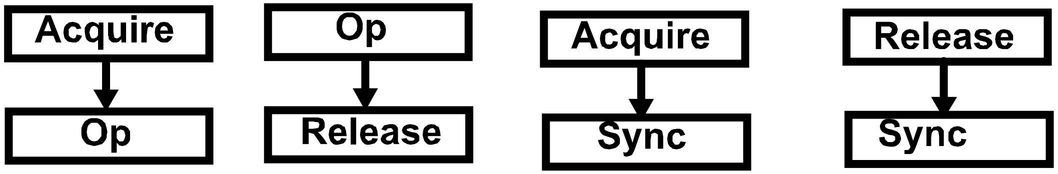






RELEASE CONSISTENCY

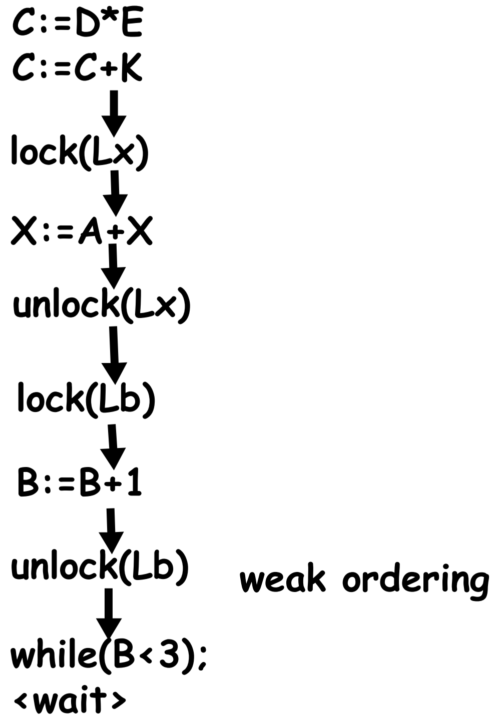
- **A REFINEMENT OF WEAK ORDERING**
  - DISTINGUISHES BETWEEN ACQUIRES AND RELEASES OF LOCK
  - ACQUIRES MUST BE BE GLOBALLY PERFORMED BEFORE STARTING ANY FOLLOWING MEMORY OPS
  - ALL MEMORY OPS MUST BE GLOBALLY PERFORMED BEFORE A RELEASE CAN START
  - MORE "RELAXED" THAN WEAK ORDERING
- **PROGRAMMERS MUST MARK SYNCHRONIZATIONS AS "ACQUIRES" OR "RELEASES"**
  - ACQUIRES AND RELEASES MUST BE SEQUENTIALLY CONSISTENT
- **ORDERS TO ENFORCE GLOBALLY**



  - Op = regular load or store
  - Sync = Acquire or Release

RELEASE CONSISTENCY

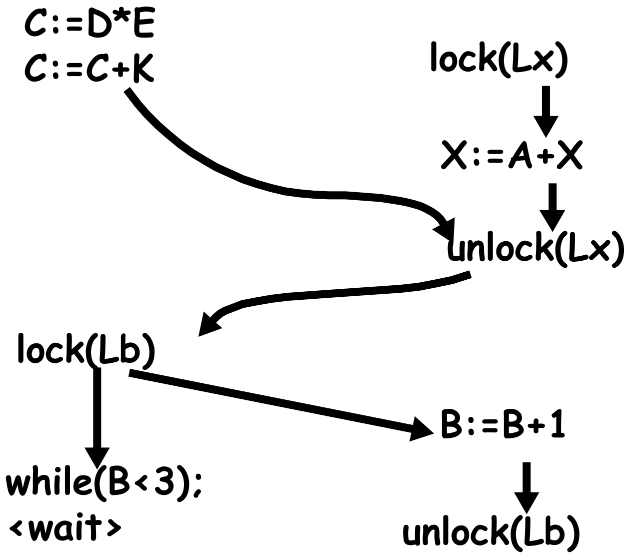
- CONSIDER THE CODE OF T2 IN THE WO EXAMPLE



C:=D*E
C:=C+K
lock(Lx)
X:=A+X
unlock(Lx)
lock(Lb)
B:=B+1
unlock(Lb)
while(B<3);
<wait>

weak ordering

WEAK ORDERING



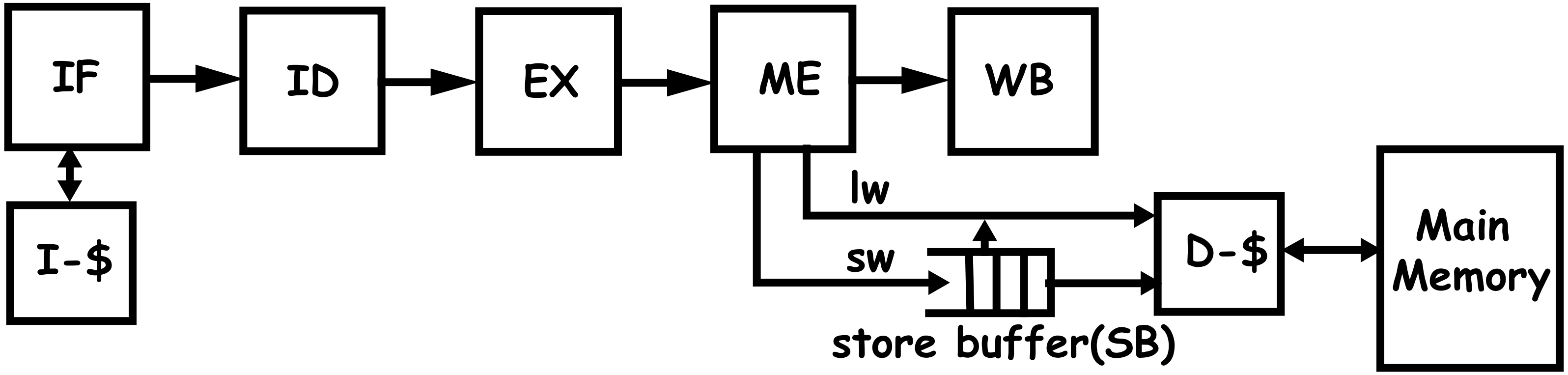
C:=D*E
C:=C+K
lock(Lx)
X:=A+X
unlock(Lx)
lock(Lb)
while(B<3);
<wait>
B:=B+1
unlock(Lb)

RELEASE CONSISTENCY

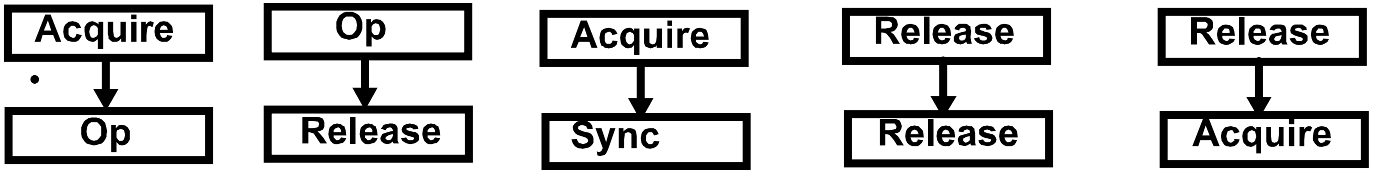
- A LOT MORE CONCURRENCY IN RELEASE vs WEAK
  - THE CODE BEFORE UNLOCK(Lx) CAN BE EXECUTED IN PARALLEL
  - THE CODE AFTER LOCK(Lb) CAN ALL BE EXECUTED IN PARALLEL
- THE PROBLEM IS TO BE ABLE TO TAKE ADVANTAGE OF THIS ADDITIONAL CONCURRENCY WITHIN EACH THREAD

RELEASE CONSISTENCY

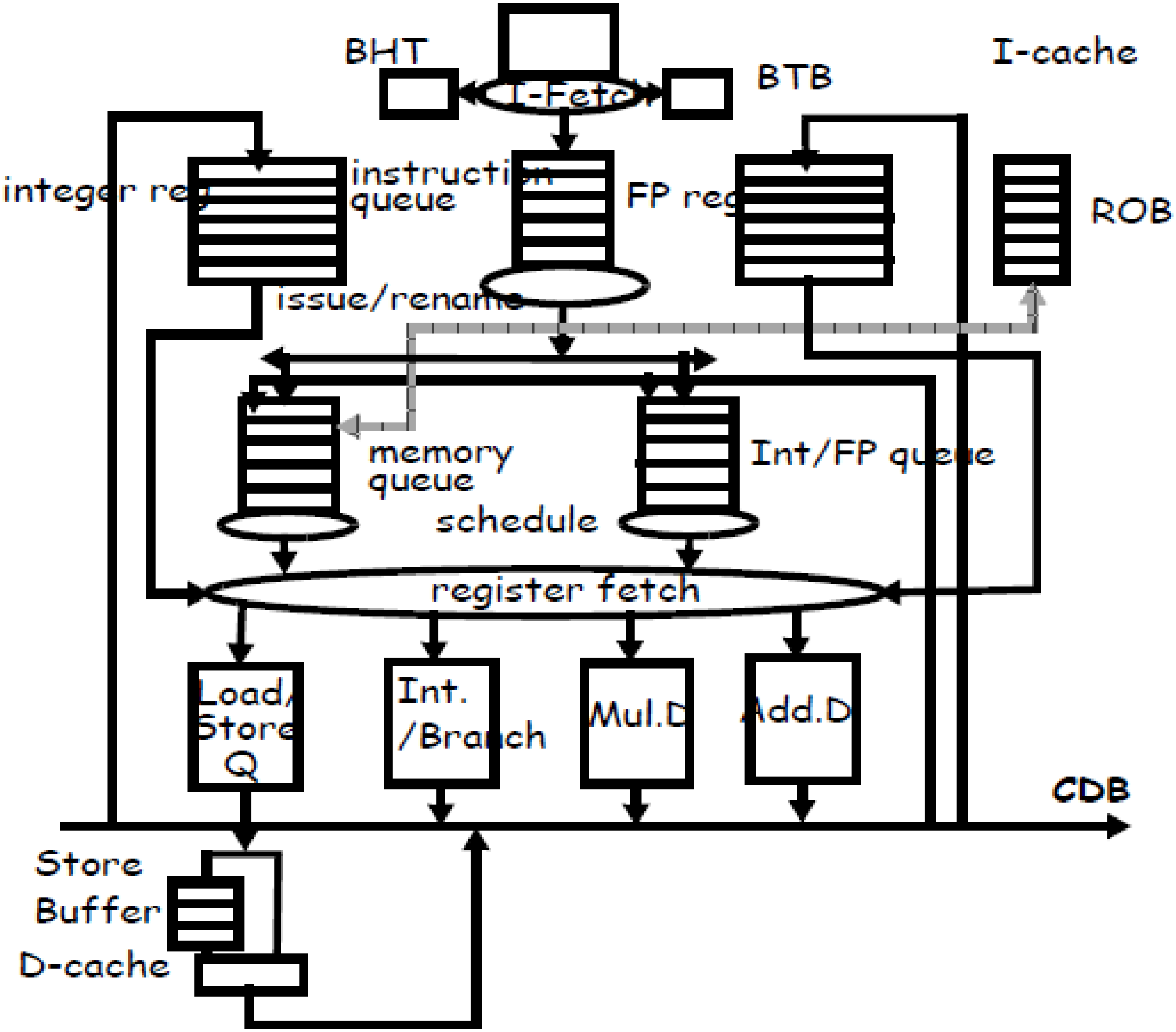
- WHAT DOES IT MEAN FOR IO PROCESSOR?



- NO ORDERING RULES AMONG REGULAR LOADs/STOREs
  - REGULAR STORES IN THE STORE BUFFER CAN BE EXECUTED IN ANY ORDER, IN PARALLEL
  - REGULAR LOADS NEVER WAIT FOR REGULAR STORES IN SB AND CAN BE FORWARDED TO
  - LOADS RETURN VALUE EVEN IF THE VALUE IS NOT GP.
- ACQUIRES ARE BLOCKING: THUS ACQUIRE-OP AND ACQUIRE-SYNC ORDER IS AUTOMATICALLY ENFORCED
- A RELEASE IS INSERTED IN THE STORE BUFFER WITH REGULAR STORES.
  - OP-RELEASE AND RELEASE-RELEASE ORDERS: RELEASES MUST WAIT IN SB UNTIL ALL PRIOR STORES AND RELEASES ARE GP (LOADS PRIOR TO THE RELEASE HAVE RETIRED)
  - RELEASE-ACQUIRE ORDER: AN ACQUIRE WAITS UNTIL ALL PRIOR RELEASES IN SB HAVE BEEN GPed



OoO PROCESSORS



ALL HAZARDS ON SAME MEMORY ADDRESS (RAW, WAW and WAR) ARE SOLVED

- IN LOAD/STORE QUEUE
- HELPS COHERENCE;
- DOES NOTHING FOR THE MCM --DIFFERENT ADDRESSES

LOADs/STOREs HANDLING IN SPECULATIVE OoO PROCESSORS

- **WHEN THE LOAD ADDRESS IS KNOWN, THE LOAD CAN ISSUE TO CACHE**
  - PROVIDED NO STORE WITH THE SAME OR AN UNKNOWN ADDRESS IS AHEAD IN THE L/S QUEUE (CONSERVATIVE MEMORY DISAMBIGUATION)
  - CAN EVEN BE ISSUED BEFORE ADDRESSES OF PREVIOUS STORES ARE KNOWN (SPECULATIVE MEMORY DISAMBIGUATION)

- **THE VALUE OF A LOAD ISSUED TO CACHE IS RETURNED AND USED SPECULATIVELY BEFORE THE LOAD RETIRES**
  - THE LOAD VALUE IS NOT BOUND UNTIL THE LOAD RETIRES (IT IS NOT PERFORMED, JUST SPECULATIVELY PERFORMED)
  - THE VALUE CAN BE RECALLED FOR ALL KINDS OF REASONS (MISSPREDICTED BRANCH, EXCEPTION, ETC…)

- **STORES CANNOT UPDATE CACHE UNTIL THEY REACH THE TOP OF THE ROB**
  - AS STORES REACH THE TOP OF THE REORDER BUFFER AND OF THE L/S QUEUE THEY COMMIT BY MOVING TO THE STORE BUFFER

CONSERVATIVE MCM ENFORCEMENT

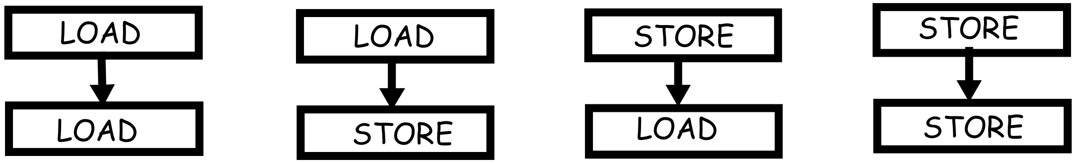
- **THE ONLY TIME THAT IT IS KNOWN FOR SURE THAT A MEMORY ACCESS IS PERFORMED IS WHEN IT REACHES THE TOP OF ROB**
  - BEFORE THAT IT IS SPECULATIVE
- **COULD WAIT TO EXECUTE AND PERFORM A LOAD IN CACHE UNTIL IT REACHES THE TOP OF ROB**
  - DOWNSIDE: NO MEMORY LATENCY TOLERANCE DUE TO OoO EXECUTION
- **IN THIS CONSERVATIVE SCHEME, LOADs AND STOREs CAN STILL BE *PREFETCHED* IN CACHE (NON-BINDING PREFETCHES)**
  - IF THE LOAD OR STORE DATA IS IN THE RIGHT STATE IN CACHE AT THE TOP OF THE ROB, PERFORMING THE ACCESS TAKES ONE CACHE CYCLE.
  - STILL, LOAD VALUES CAN NOT BE USED SPECULATIVELY

NEXT IDEA: EXPLOIT SPECULATIVE EXECUTION TO SPECULATIVELY VIOLATE MEMORY ORDERS

WORKS BECAUSE ORDER VIOLATIONS ARE RARE

SPECULATIVE VIOLATIONS OF SEQUENTIAL CONSISTENCY

- **LET'S LOOK AGAIN AT ORDERS TO ENFORCE GLOBALLY**



- **BECAUSE STOREs ARE SENT TO THE STORE BUFFER ONLY WHEN THEY RETIRE, THE LOAD-STORE AND STORE-STORE ORDERS ARE AUTOMATICALLY SATISFIED IN THE PROCESSOR**
  - PROVIDED STOREs ARE GLOBALLY PERFORMED IN ORDER FROM THE STORE BUFFER

- **REMAINING ORDERS ARE LOAD-LOAD AND STORE-LOAD**
  - THE SECOND ACCESS IN THESE ORDERS IS A LOAD

EXPLOIT SPECULATION

SPECULATIVE VIOLATIONS OF SEQUENTIAL CONSISTENCY

- **ASSUME THAT A MEMORY ACCESS (LOAD OR STORE) TO X PRECEDES A LOAD OF Y AND THAT THE ORDER MUST BE ENFORCED BY THE MODEL, I.E.,**

  LOAD X/STORE X, followed by
  LOAD Y

  - PERFORM LOAD Y SPECULATIVELY BEFORE THE ACCESS TO X
  - USE THE VALUE RETURNED BY LOAD Y SPECULATIVELY.
  - LATER, WHEN ACCESS TO X RETIRES AT THE TOP OF ROB, CHECK TO SEE WHETHER THE VALUE OF Y HAS CHANGED AND ROLL BACK IF IT HAS
  - OR (MORE SIMPLY) CHECK THE VALUE OF Y WHEN LOAD Y REACHES THE TOP OF ROB

- **TWO MECHANISMS ARE NEEDED: VALIDATION AND ROLLBACK**
- **VALIDATION OF LOAD VALUES AT THE TOP OF ROB**
  - CHECK THE VALUE OF Y BY RE-PERFORMING THE LOAD IN CACHE (USES EXTRA CACHE BANDWIDTH AND POSSIBLY CYCLES)
  - OR: MONITOR EVENTS THAT COULD CHANGE THE VALUE OF Y BETWEEN THE TIME LOAD Y PERFORMS SPECULATIVELY UNTIL LOAD Y COMMITS
    - ""EVENTS" INCLUDE INVALIDATES, UPDATES OR CACHE REPLACEMENTS IN THE NODE'S CACHE
    - SNOOP THE LOAD Q

"LOAD VALUE RECALL": ROB ROLLBACK, AS FOR MISPREDICTED BRANCHES

SPECULATIVE VIOLATIONS OF SEQUENTIAL CONSISTENCY

- LOAD-LOAD ORDER: USE LOAD VALUE RECALL

- STORE-LOAD ORDER: USE LOAD VALUE RECALL + STALL LOADS AT TOP OF ROB UNTIL ALL PREVIOUS STOREs HAVE BEEN GLOBALLY PERFORMED

- LOAD-STORE ORDER: STOREs RETIRE AT TOP OF ROB, BY THAT TIME ALL PREVIOUS LOADS HAVE RETIRED

- STORE-STORE ORDER: STOREs RETIRE AT TOP OF ROB + GLOBALLY PERFORM STOREs FROM STORE BUFFER ONE BY ONE IN T.O.

- PERFORMANCE ISSUES
  - A LOAD MAY REACH THE TOP OF THE ROB BUT CANNOT PERFORM AND IS STALLED BECAUSE OF PRIOR STORES IN THE STORE BUFFER
  - THIS BACKS UP THE ROB (AND OTHER INTERNAL BUFFERS) AND EVENTUALLY STALLS THE PROCESSOR

NEXT IDEA: RELAX STORE-LOAD ORDER

SPECULATIVE TSO VIOLATIONS

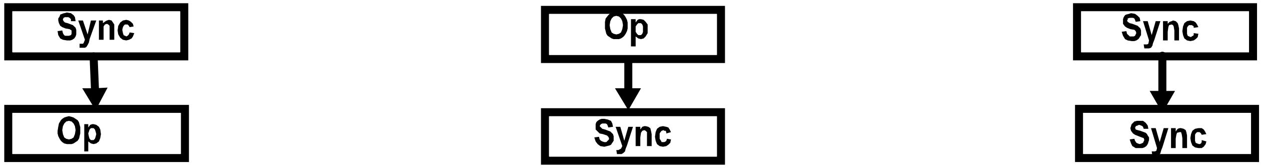
- LOAD-LOAD ORDER: USE LOAD VALUE RECALL
- STORE-LOAD ORDER: NO NEED TO ENFORCE. THUS VALUE OF LOAD SHOULD NOT BE RECALLED IF ALL PRECEDING PENDING ACCESSES IN THE LOAD/STORE Q ARE STOREs (POSSIBLE OPTIMIZATION) + LOADS DON'T WAIT ON STORES IN SB
- LOAD-STORE ORDER: STOREs RETIRE AT TOP OF ROB
- STORE-STORE ORDER: STOREs RETIRE AT TOP OF ROB + STOREs ARE GLOBALLY PERFORMED FROM STORE BUFFER IN T.O.
- PERFORMANCE ISSUES:
  - IF A LONG LATENCY STORE BACKS UP THE STORE BUFFER THEN STORES CANNOT RETIRE, WHICH MAY BACK UP THE ROB AND OTHER Q's AND STALL DISPATCH

NEXT IDEA: RELAX FURTHER WITH WEAK ORDERING OR RELEASE CONSISTENCY

SPECULATIVE EXECUTION OF RMW ACCESSES

- **RMW INSTRUCTIONS ARE MADE OF A LOAD FOLLOWED BY A STORE**
  - MUST EXECUTE AS A GROUP AND BE ATOMIC
- **BECAUSE OF THE LOAD, THE VALUE OF THE LOCK IS RETURNED SPECULATIVELY**
  - BASED ON THE SPECULATIVE LOCK VALUE THE CRITICAL SECTION IS ENTERED OR THE LOCK IS RETRIED SPECULATIVELY
  - THIS ACTIVITY IS ALL SPECULATIVE IN THE ROB
  - THE STORE IN THE RMW DOES NOT EXECUTE IN CACHE UNTIL IT REACHES THE TOP OF ROB
  - LOADs IN RMW ACCESSES MUST REMAIN SUBJECT TO LOAD VALUE RECALL UNTIL THE RMW IS RETIRED
  - WILL BE ROLLED BACK IF VIOLATION IS DETECTED
- **BECAUSE OF THE STORE, THE RMW ACCESS DOES NOT UPDATE MEMORY (AND CANNOT PERFORM) UNTIL THE TOP OF THE ROB**
  - IF THE STORE CAN EXECUTE AT THE TOP OF THE ROB, THEN NO OTHER THREAD GOT THE LOCK DURING THE TIME THE RMW WAS SPECULATIVE
  - THE LOAD AND THE STORE CAN BE PERFORMED ATOMICALLY IN CACHE WITH A WRITE-INVALIDATE PROTOCOL

SPECULATIVE VIOLATIONS OF WEAK ORDERING

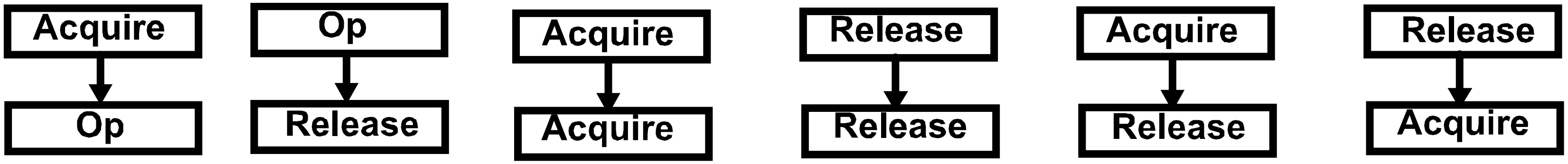


- **OP-TO-SYNC:**
  - ANY ACCESS TO A SYNC VARIABLE MUST BE GLOBALLY PERFORMED AT THE TOP OF THE ROB, AFTER ALL PREVIOUS ACCESSES HAVE RETIRED AND AFTER ALL STORES IN THE STORE BUFFER HAVE BEEN GLOBALLY PERFORMED

- **SYNC-TO-OP/SYNC-TO-SYNC:**
  - NO OP OR SYNC ACCESS FOLLOWING AN ACCESS TO A SYNC VARIABLE CAN PERFORM UNTIL THE SYNC ACCESS HAS BEEN GLOBALLY PERFORMED.
    - THIS IS AUTOMATICALLY ENFORCED FOR STORES AND FOR SYNC ACCESSES SINCE THEY CANNOT BE GLOBALLY PERFORMED UNTIL THEY REACH THE TOP OF THE ROB.
    - NO SYNC ACCESS MAY COMMIT AND PERFORM UNTIL THE STORE BUFFER IS EMPTY.
    - A LOAD OF A SYNC VARIABLE (INCLUDING THE LOAD IN A RMW ACCESS) CAN BE SPECULATIVELY PERFORMED PROVIDED IT IS SUBJECT TO LOAD VALUE RECALL.
- **REGULAR LOADS ARE NOT SUBJECT TO LOAD VALUE RECALL**

SPECULATIVE VIOLATIONS OF RELEASE CONSISTENCY

- IN RC, A RELEASE MAY NOT PERFORM UNTIL ALL PREVIOUS ACCESSES ARE GLOBALLY PERFORMED AND ACCESSES FOLLOWING AN ACQUIRE MUST BE DELAYED UNTIL THE ACQUIRE IS GLOBALLY PERFORMED.



- RC REQUIRES THAT ACQUIRES MUST BE PERFORMED BEFORE ANY FOLLOWING ACCESS CAN PERFORM.
  - HOWEVER ALL LOADS, INCLUDING RMW LOADS, MAY BE **SPECULATIVELY** PERFORMED.
  - REGULAR LOADS ARE NOT SUBJECT TO RECALL,
  - ONLY SYNC LOADS IN ACQUIRES ARE SUBJECT TO RECALL
- RELEASES ARE PUT IN THE STORE BUFFER AS "SPECIAL" STORES (UNLESS THEY ARE RMW INSTRUCTIONS)
  - BY THAT TIME ALL PREVIOUS LOADS HAVE BEEN PERFORMED
  - RELEASE MUST WAIT IN SB UNTIL ALL PREVIOUS STORES HAVE BEEN PERFORMED
- ALL STORES IN THE STORE BUFFER (BESIDES RELEASES) CAN PERFORM IN ANY ORDER
- ACQUIRES MUST WAIT UNTIL ALL PRIOR RELEASES IN SB ARE PERFORMED