LECTURE 14

Memory Consistency

Miquel Pericàs
EDA284/DIT361 - VT 2020

Overview of Lectures 12-14

Goals:
1. Study the correct and reliable communication of values in shared-memory multiprocessors
2. How programmers can enforce order in modern CMPs

- **Synchronization (L12) 26/2**
- **Strict coherence (L13) 3/3**
- **Plain coherence (L13) 3/3**
- **Sequential consistency (L14, part 1) 3/3**
- **Memory consistency models (L14, part 2) 6/3**

What's cooking

1. **Lectures**
    - Today - Final lecture**: Memory Consistency Models**
    - Tuesday (10/3): **The European Processor Initiative** (Guest Lecturers: Sonia Rani Gupta, Bhavishya Goel)
    - Wednesday (11/3): **Q&A session**
2. **Project Status**
    - March 10th deadline for assessment phase
3. **Practice Sessions**
    - Tuesday (10/3): **Message Passing, Synchronization**
        - problems will be published soon

Memory Consistency Models: Outline

- **Sequential Consistency (today)**
- **Relaxed Memory Consistency Models (Friday)**
    - Not relying on Synchronization (E.G. Store-load Relaxation)
    - Relying on Synchronization (E.G. Weak Ordering)

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;
 …                  …
```

- **<u>Expectation</u>:**
  - S^1(A)1 should reach P2 before S^1(flag)1
  - with coherence only, print A=0 is a valid outcome

Coherence is not sufficient

- **Communication**

```
  assume A and B are both 0 initially
   P1              P2
    …              …
   A:=1;           print B;
   B:=2;           print A;
    …               …
```

- **Expectation**
  - if print B prints 2, then print A must print 1
  - again, with coherence only (2, 0) is a valid outcome

Coherence is not sufficient

- **Dekker's Algorithm (critical section)**

```
 assume A and B are both 0 initially
P1                      P2

 …                       …
S¹(A):A:=1              S²(B):B:=1
L¹(B):while(B==1);      L²(A):while(A==1);
<critical section>       <critical section>
A:=0                     B:=0
```

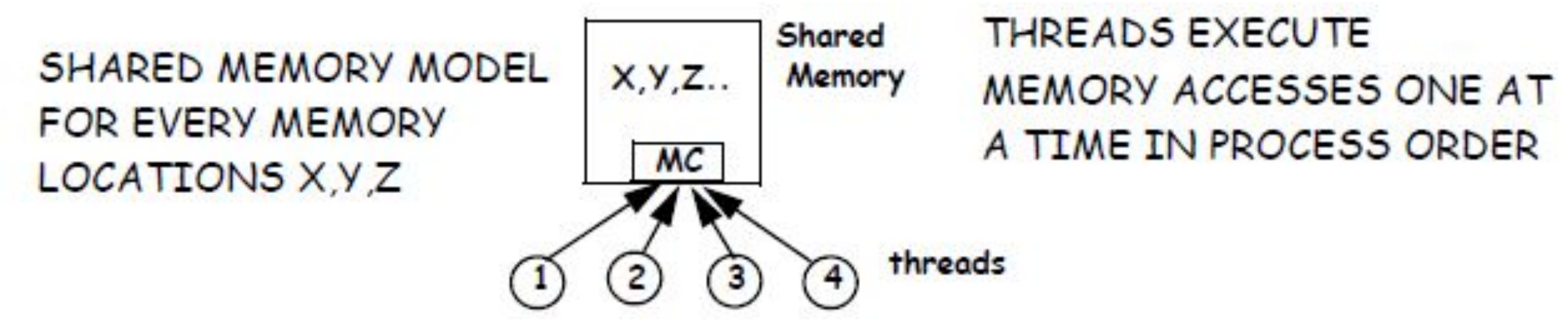
- **Expectation**
  - Only one thread executes the critical section, or deadlock if both threads execute the while statement at the same time.
  - Even with store atomicity $L^1(B)0$ and $L^2(A)0$ may happen, which results in both threads entering the critical section

Coherence is not sufficient

- **<u>Programmer's intuition</u> is that accesses from different processes are "interleaved" in process order**
  - different from coherence, which applies to a single location
  - need to formalize this model

SEQUENTIAL CONSISTENCY

Formal model for Sequential Consistency (SC)



- **Program order of all threads is respected and all memory accesses are atomic**

A multiprocessor is sequentially consistent if "the result of any execution is the same as if the memory operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in the sequence in the order specified by its program"

Leslie Lamport, *How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs*, IEEE Transactions on Computers, September 1979

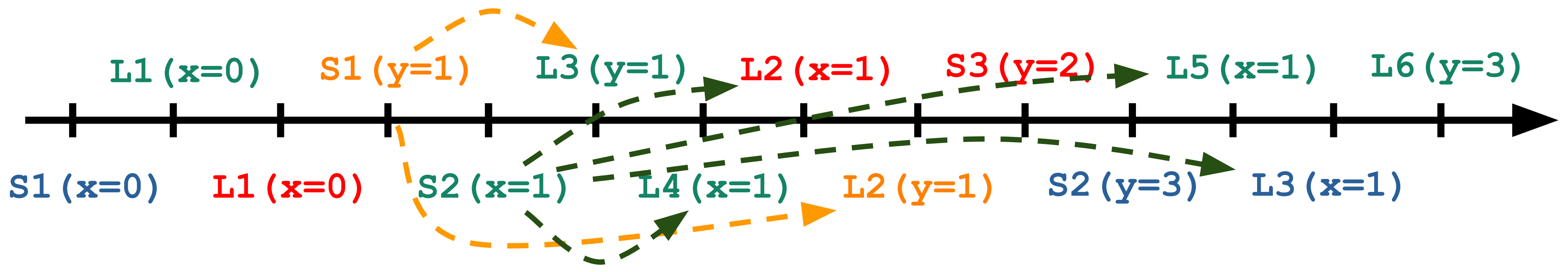
Sequential Consistency (SC)

What it means: all loads and stores from all threads can be laid out on a causality line so that:

1) all intra-thread orders are respected
2) only one access to shared memory is executed at a time

```
Thread 1: S1(x=0) → S2(y=3) → L3(x=1)
Thread 2: L1(x=0) → S2(x=1) → L3(y=1) → L4(x=1) → L5(x=1) → L6(y=3)
Thread 3: L1(x=0) → L2(x=1) → S3(y=2)
Thread 4: S1(y=1) → L2(y=1)
```

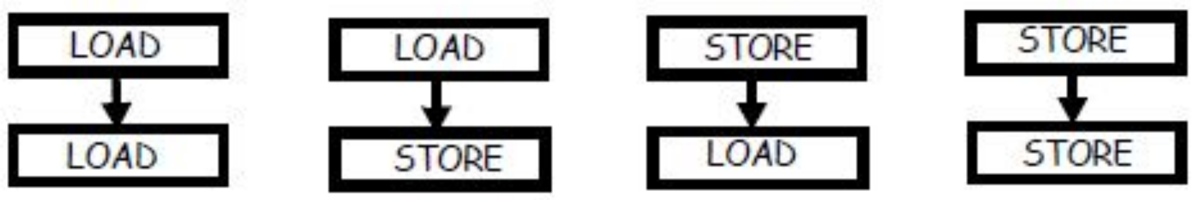
Causality timeline of loads (L) and stores (S)



Like plain coherence, except that it applies to all memory locations

Sequential Consistency

- Thread orders that must be enforced in the global order of all memory accesses

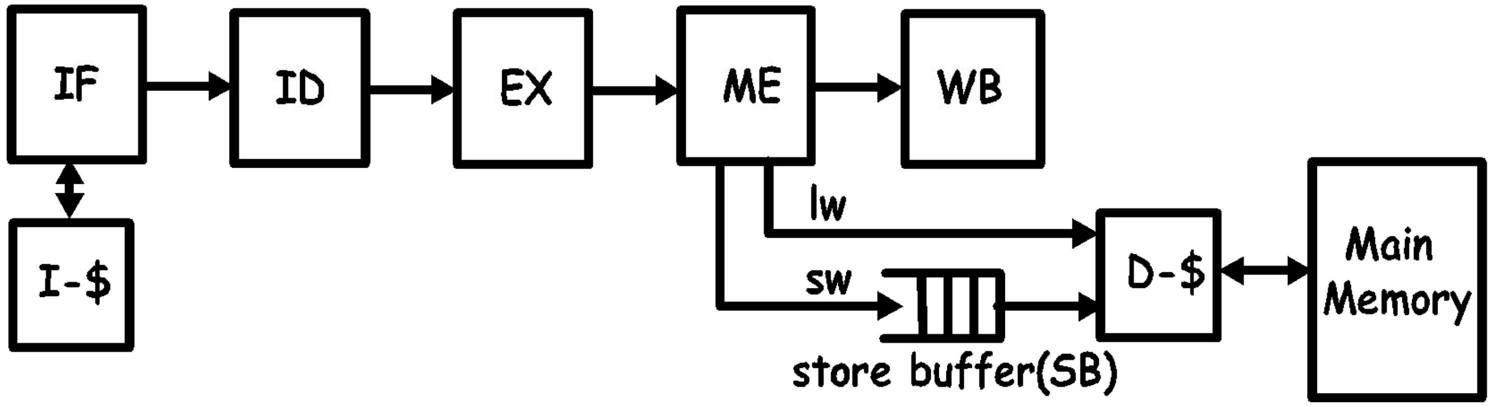


- **<u>In SC all intra-thread orders must be globally enforced</u>**

- <u>Sufficient condition</u> for Sequential Consistency:
  - Global order of stores to same address is enforced
  - A thread cannot issue an access to memory until <u>all its previous memory accesses</u> (for all memory locations) have been globally performed.
    - Hence, processor needs to wait before starting every single access to any memory location!

Sequential Consistency

- What does this mean for in-order 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 (no forwarding!)
  - **STORE-STORE**: stores must be GPed one by one from SB (no write combining, unless there is no intervening store to a different location between two stores that combine)

Sequential Consistency and store buffers

- **Problem with store buffer: with forwarding store buffers, loads can be performed before previous stores**
  - in SC, a LOAD must be stalled if prior STOREs are not GPed. No forwarding
  - Store Buffer is effective for long bursts of STOREs (write combining)
    - But STOREs must be propagated one by one before the next load anyway

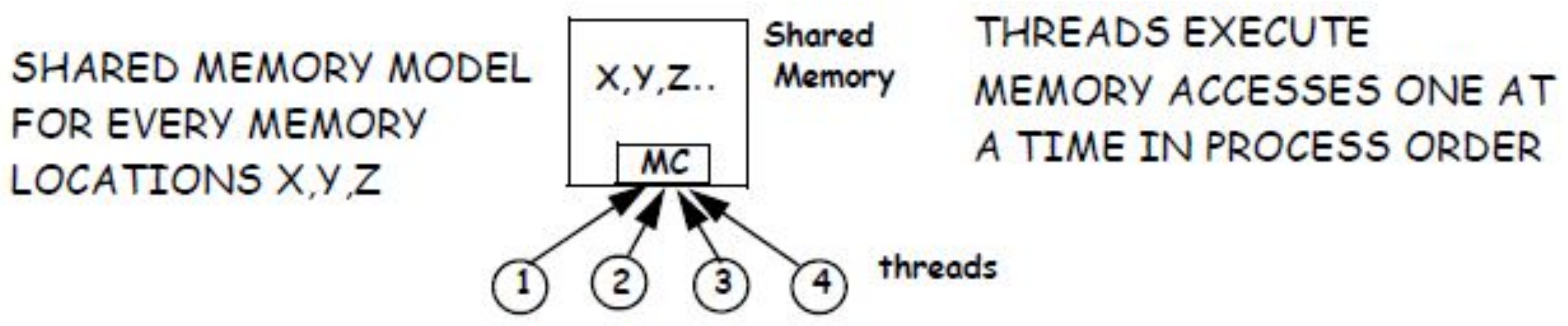
- **LOOK AT DEKKER AGAIN**

```
A and B are both 0 and cached as Shared initially
T1                      T2
S¹(A)1                  S²(B)1
L¹(B)?                  L²(A)?
```

What outcomes are possible under Sequential Consistency?

Dekker: expected outcomes

- Programmer Intuition:


SHARED MEMORY MODEL FOR EVERY MEMORY LOCATIONS X,Y,Z
X,Y,Z..
MC
Shared Memory
THREADS EXECUTE MEMORY ACCESSES ONE AT A TIME IN PROCESS ORDER
1
2
3
4
threads

Processor instructions:

```
Initially X,Y := 0
T1.1: ST X,1           T2.1: ST Y,1
T1.2: LD EAX1,Y        T2.2: LD EAX2,X
```

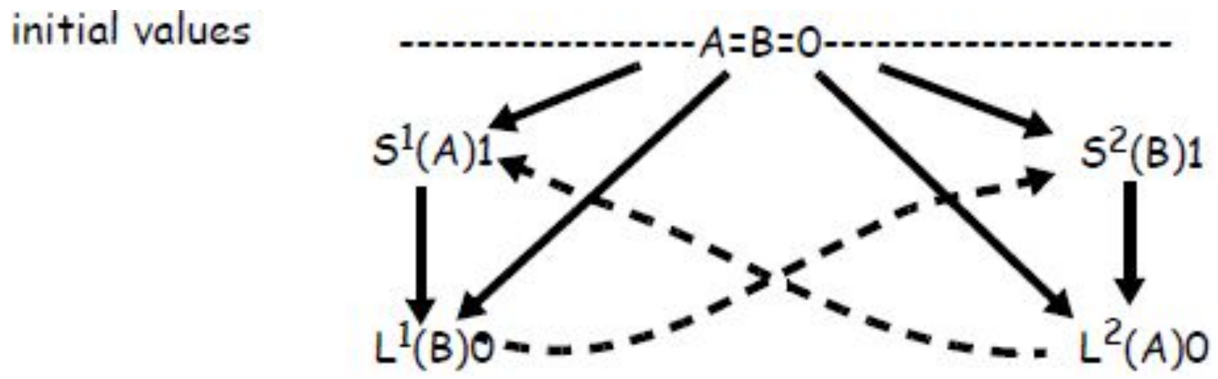
EAX1, EAX2 outcome

- **Quiz**: out of the four outcomes [EAX1,EAX2]= [1,1], [0,1], [1,0], [0,0], which are possible?

Dekker and SC

- look at dekker again, can SC result in the [0,0] outcome?

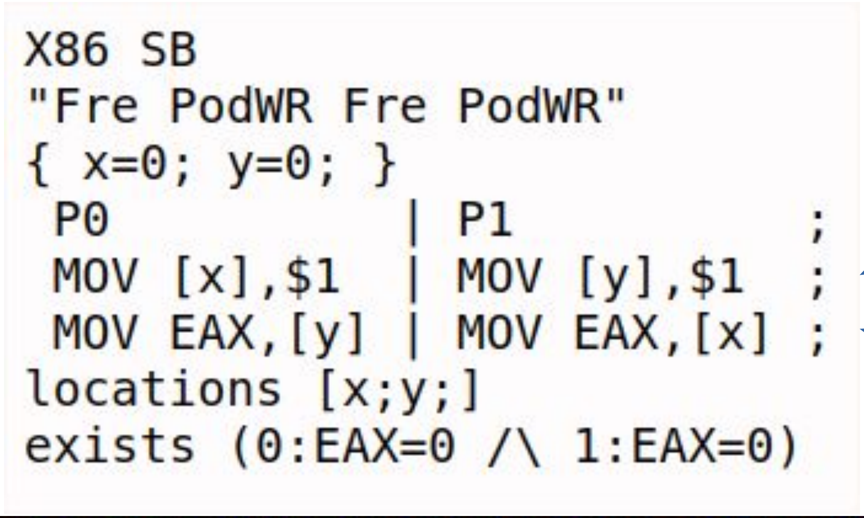
```
    A and B are both 0 and cached as Shared initially
    T1                    T2
    S¹(A)1                S²(B)1
    L¹(B)0                L²(A)0
```



- Execution graph has cycles → **execution cannot be ordered, hence this is not possible under SC!**
- Dekker's algorithm is correct under SC, so modern hardware should implement SC, right?

Ok, let's see this on my laptop

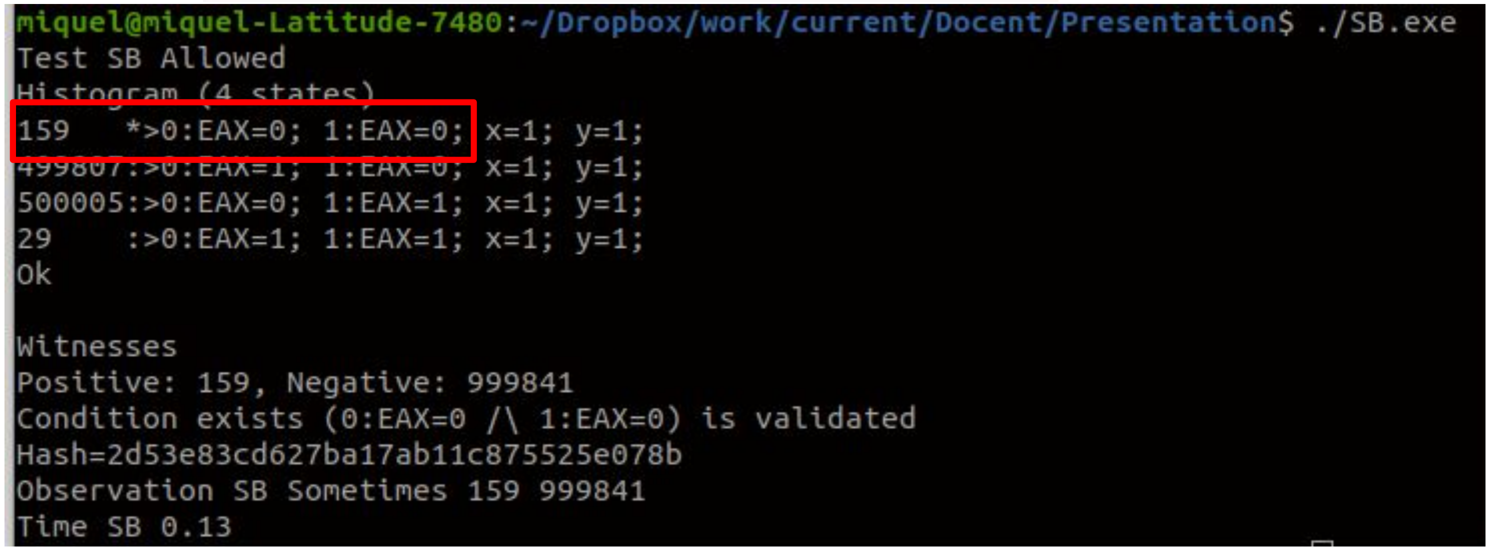
with http://diy.inria.fr/



```
X86 SB
"Fre PodWR Fre PodWR"
{ x=0; y=0; }
 P0          | P1          ;
 MOV [x],$1  | MOV [y],$1  ;
 MOV EAX,[y] | MOV EAX,[x] ;
locations [x;y;]
exists (0:EAX=0 /\ 1:EAX=0)
```

ST Y, 1

LD EAX, X



```
miquel@miquel-Latitude-7480:~/Dropbox/work/current/Docent/Presentation$ ./SB.exe
Test SB Allowed
Histogram (4 states)
159    *>0:EAX=0; 1:EAX=0; x=1; y=1;
499807:>0:EAX=1; 1:EAX=0; x=1; y=1;
500005:>0:EAX=0; 1:EAX=1; x=1; y=1;
29     :>0:EAX=1; 1:EAX=1; x=1; y=1;
Ok

Witnesses
Positive: 159, Negative: 999841
Condition exists (0:EAX=0 /\ 1:EAX=0) is validated
Hash=2d53e83cd627ba17ab11c875525e078b
Observation SB Sometimes 159 999841
Time SB 0.13
```

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

on Intel systems, L^1(B)0 & L^2(A)0 is possible!
i.e. both threads will enter the critical section.
Is Intel HW broken, or does it allow such outcomes?

Dekker, SC and Store Buffers

- Intel HW allows loads to bypass stores in the Store Buffer
- With store buffers and no GP store-load order, 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
- Problem: SC is very restrictive and cannot take really advantage of store buffers
- But store buffers are critical for performance!

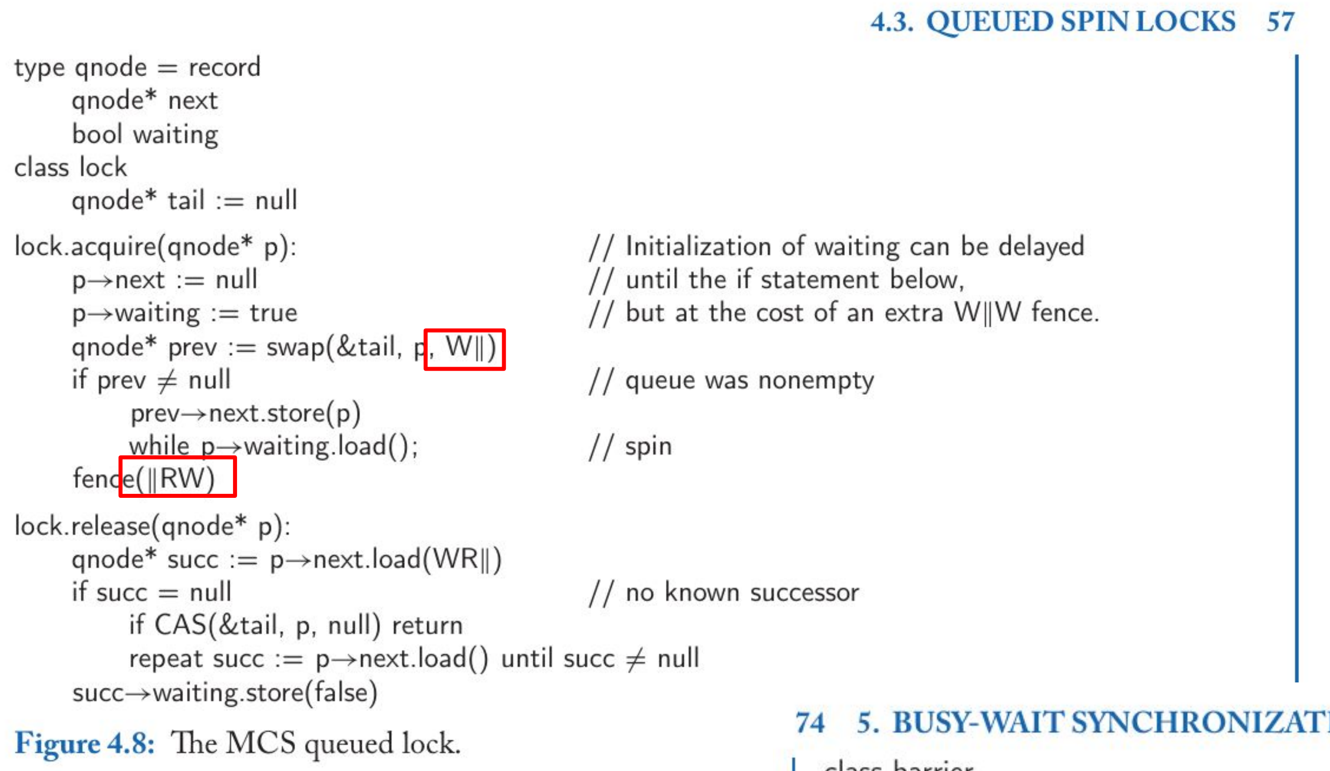
    - How do we solve the *performance* vs *correctness* question?

need to change the rules:

MEMORY CONSISTENCY MODELS

(next lecture)

How to implement locks and lock-free algorithms?

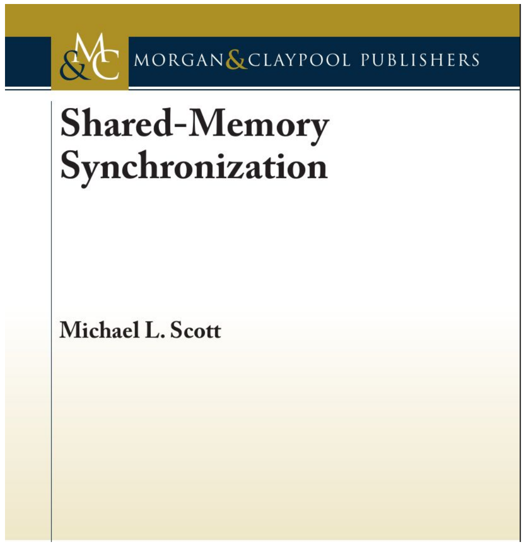


```
type qnode = record
    qnode* next
    bool waiting
class lock
    qnode* tail := null

lock.acquire(qnode* p):                 // Initialization of waiting can be delayed
    p→next := null                      // until the if statement below,
    p→waiting := true                   // but at the cost of an extra W‖W fence.
    qnode* prev := swap(&tail, p, W‖)
    if prev ≠ null                      // queue was nonempty
        prev→next.store(p)
        while p→waiting.load();         // spin
    fence(‖RW)

lock.release(qnode* p):
    qnode* succ := p→next.load(WR‖)
    if succ = null                      // no known successor
        if CAS(&tail, p, null) return
        repeat succ := p→next.load() until succ ≠ null
    succ→waiting.store(false)
```

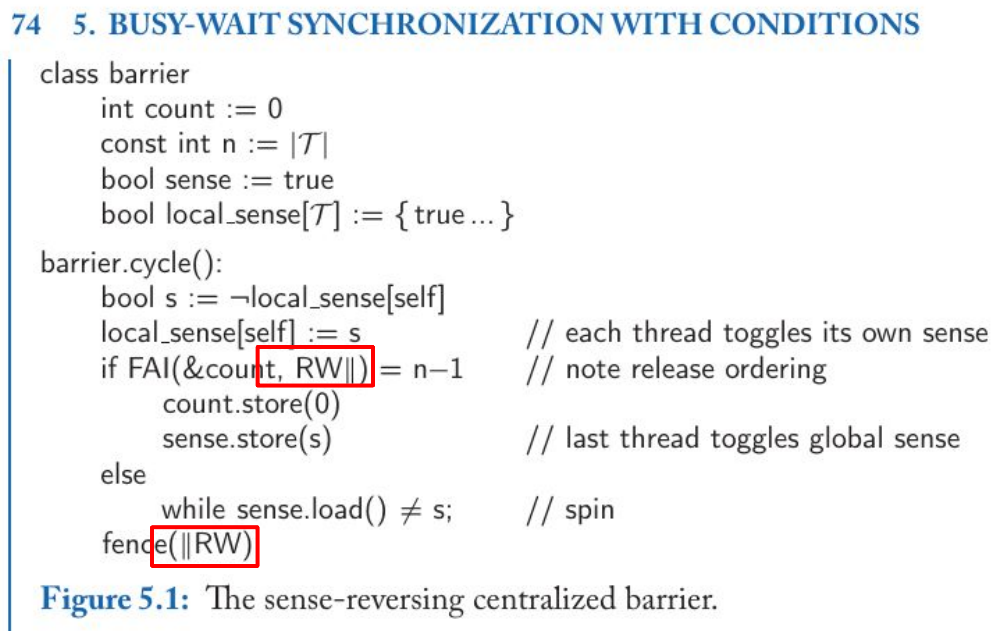
Figure 4.8: The MCS queued lock.



MORGAN & CLAYPOOL PUBLISHERS

Shared-Memory Synchronization

Michael L. Scott



```
class barrier
    int count := 0
    const int n := |T|
    bool sense := true
    bool local_sense[T] := { true ... }

barrier.cycle():
    bool s := ¬local_sense[self]
    local_sense[self] := s              // each thread toggles its own sense
    if FAI(&count, RW‖) = n−1           // note release ordering
        count.store(0)
        sense.store(s)                  // last thread toggles global sense
    else
        while sense.load() ≠ s;         // spin
    fence(‖RW)
```

Figure 5.1: The sense-reversing centralized barrier.

What is the meaning of these annotations?

How do we use them in our programs?

Memory Consistency Models: Outline

- ~~Sequential Consistency~~
- **Relaxed Memory Consistency Models (today)**
  - Not relying on Synchronization (E.G. Store-load Relaxation)
  - Relying on Synchronization (E.G. Weak Ordering)

Memory Consistency Models

- **SC is very restrictive. Are there other programmer's intuitions?**
  - such as those provided by synchronization?
  - Whenever shared variables are read/write they should be protected by synchronization methods (locks, barriers, ...):

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

- Thee barrier enforces the order such that P2 can only observe the SC-compliant case (A,B) = (1,2).

Accesses to sync variables can be treated differently from accesses to regular variables by the hardware, e.g. via special loads and stores or RMW atomics.

In this case, the pursuit of sequential consistency seems pointless. Does it allow to build more efficient HW?

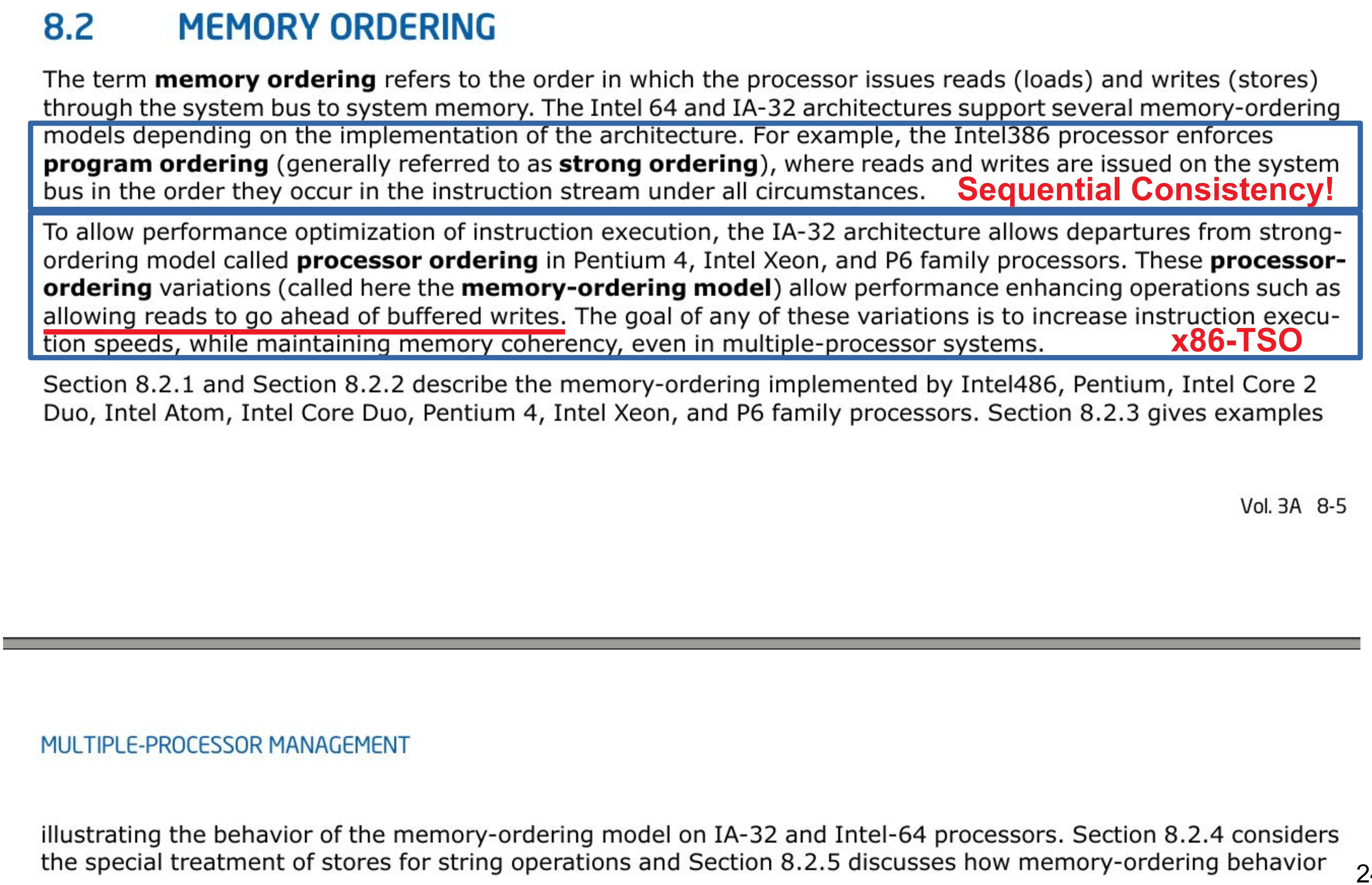
Let's consider the opposite approach?

- **Why bother about the programmer anyway?**
  - How about designing memory access rules that are hardware friendly and then let the programmer take care of it??

Memory Consistency Model

- **In any case we need a memory access ordering model on which programmers, compilers 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

From the Intel64 developers manual (Part 3.a)



8.2 MEMORY ORDERING

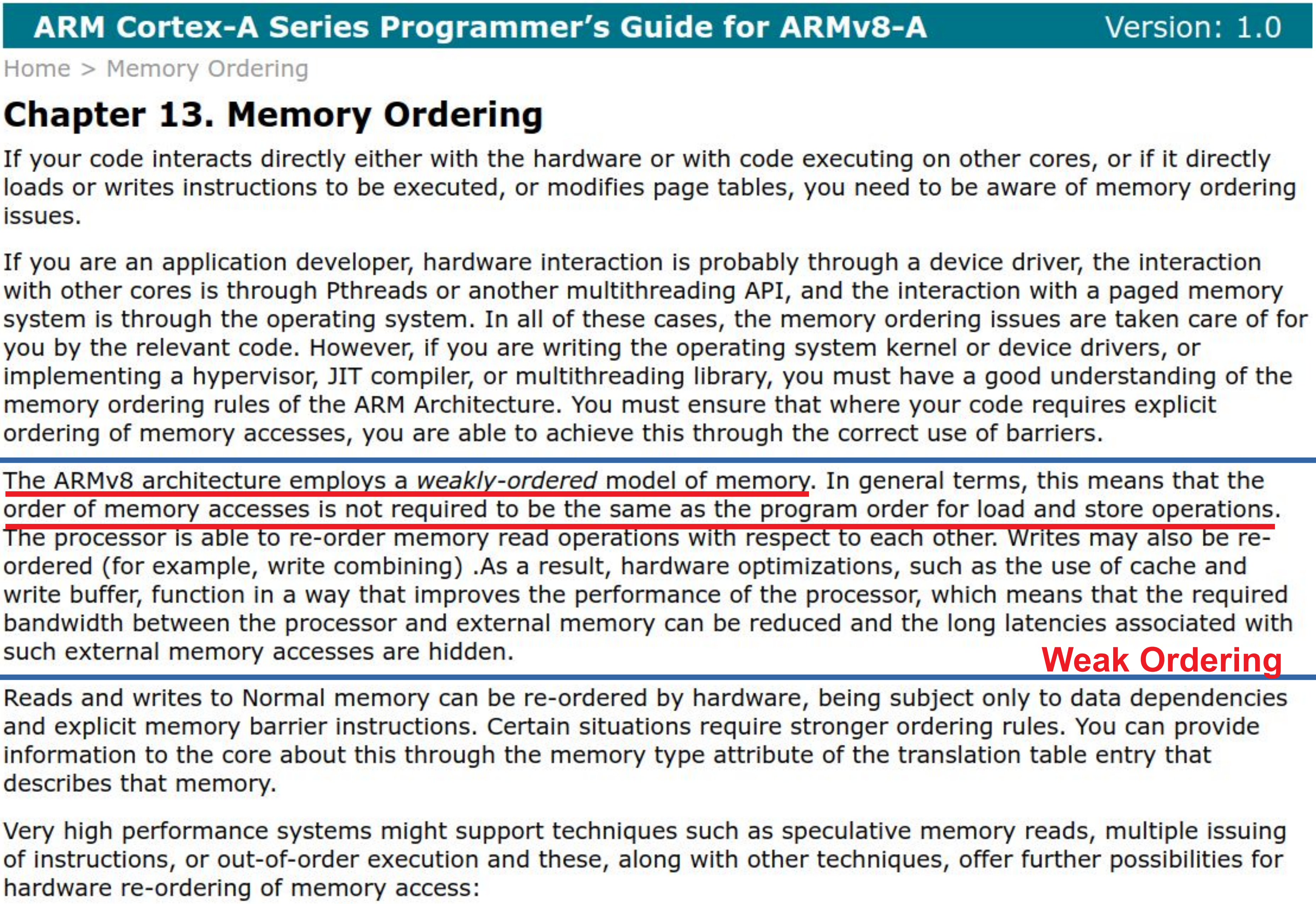
The term **memory ordering** refers to the order in which the processor issues reads (loads) and writes (stores) through the system bus to system memory. The Intel 64 and IA-32 architectures support several memory-ordering models depending on the implementation of the architecture. For example, the Intel386 processor enforces **program ordering** (generally referred to as **strong ordering**), where reads and writes are issued on the system bus in the order they occur in the instruction stream under all circumstances. **Sequential Consistency!**

To allow performance optimization of instruction execution, the IA-32 architecture allows departures from strong-ordering model called **processor ordering** in Pentium 4, Intel Xeon, and P6 family processors. These **processor-ordering** variations (called here the **memory-ordering model**) allow performance enhancing operations such as allowing reads to go ahead of buffered writes. The goal of any of these variations is to increase instruction execution speeds, while maintaining memory coherency, even in multiple-processor systems. **x86-TSO**

Section 8.2.1 and Section 8.2.2 describe the memory-ordering implemented by Intel486, Pentium, Intel Core 2 Duo, Intel Atom, Intel Core Duo, Pentium 4, Intel Xeon, and P6 family processors. Section 8.2.3 gives examples

illustrating the behavior of the memory-ordering model on IA-32 and Intel-64 processors. Section 8.2.4 considers the special treatment of stores for string operations and Section 8.2.5 discusses how memory-ordering behavior

From the ARMv8-A manual



ARM Cortex-A Series Programmer's Guide for ARMv8-A Version: 1.0

Home > Memory Ordering

Chapter 13. Memory Ordering

If your code interacts directly either with the hardware or with code executing on other cores, or if it directly loads or writes instructions to be executed, or modifies page tables, you need to be aware of memory ordering issues.

If you are an application developer, hardware interaction is probably through a device driver, the interaction with other cores is through Pthreads or another multithreading API, and the interaction with a paged memory system is through the operating system. In all of these cases, the memory ordering issues are taken care of for you by the relevant code. However, if you are writing the operating system kernel or device drivers, or implementing a hypervisor, JIT compiler, or multithreading library, you must have a good understanding of the memory ordering rules of the ARM Architecture. You must ensure that where your code requires explicit ordering of memory accesses, you are able to achieve this through the correct use of barriers.

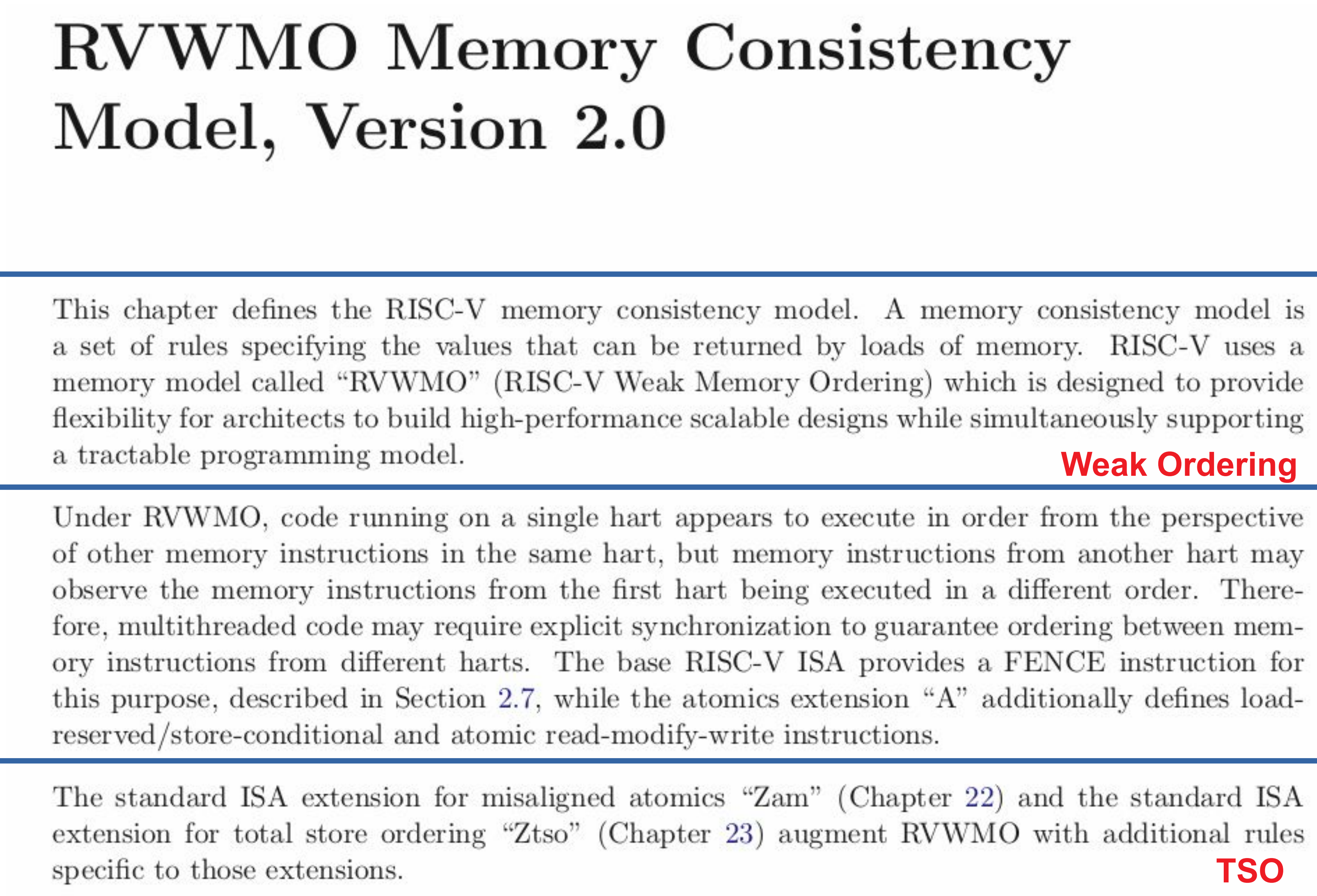
The ARMv8 architecture employs a *weakly-ordered* model of memory. In general terms, this means that the order of memory accesses is not required to be the same as the program order for load and store operations. The processor is able to re-order memory read operations with respect to each other. Writes may also be re-ordered (for example, write combining) .As a result, hardware optimizations, such as the use of cache and write buffer, function in a way that improves the performance of the processor, which means that the required bandwidth between the processor and external memory can be reduced and the long latencies associated with such external memory accesses are hidden.

Weak Ordering

Reads and writes to Normal memory can be re-ordered by hardware, being subject only to data dependencies and explicit memory barrier instructions. Certain situations require stronger ordering rules. You can provide information to the core about this through the memory type attribute of the translation table entry that describes that memory.

Very high performance systems might support techniques such as speculative memory reads, multiple issuing of instructions, or out-of-order execution and these, along with other techniques, offer further possibilities for hardware re-ordering of memory access:

From the RISC-V spec



RVWMO Memory Consistency Model, Version 2.0

This chapter defines the RISC-V memory consistency model. A memory consistency model is a set of rules specifying the values that can be returned by loads of memory. RISC-V uses a memory model called “RVWMO” (RISC-V Weak Memory Ordering) which is designed to provide flexibility for architects to build high-performance scalable designs while simultaneously supporting a tractable programming model.

Weak Ordering

Under RVWMO, code running on a single hart appears to execute in order from the perspective of other memory instructions in the same hart, but memory instructions from another hart may observe the memory instructions from the first hart being executed in a different order. Therefore, multithreaded code may require explicit synchronization to guarantee ordering between memory instructions from different harts. The base RISC-V ISA provides a FENCE instruction for this purpose, described in Section 2.7, while the atomics extension “A” additionally defines load-reserved/store-conditional and atomic read-modify-write instructions.

The standard ISA extension for misaligned atomics “Zam” (Chapter 22) and the standard ISA extension for total store ordering “Ztso” (Chapter 23) augment RVWMO with additional rules specific to those extensions.

TSO

Memory Consistency Models

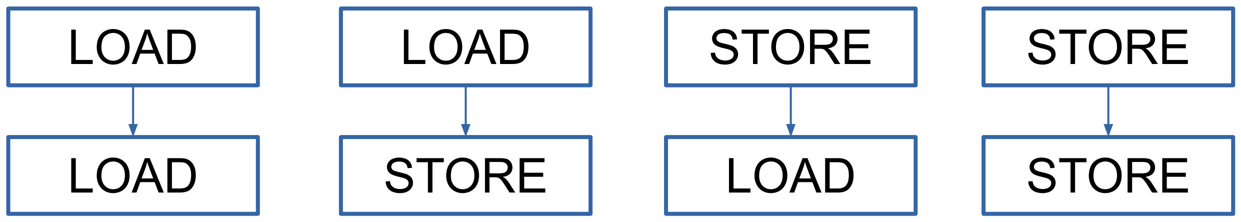
- there are two types of relaxed memory consistency models:
  - models not relying on synchronization
  - models relying on synchronization

Let's look first at models not relying on synchronization

Relaxed Memory Consistency Models

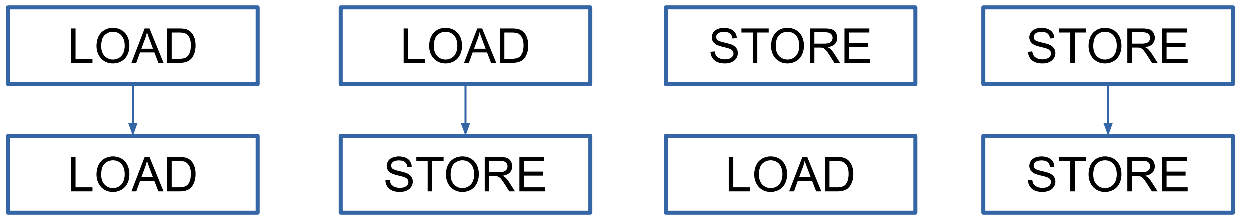
Sequential Consistency is a strict memory model. All intra-thread orders must be enforced in the global order

Sequential Consistency (SC)
[old Intel, MIPS...]

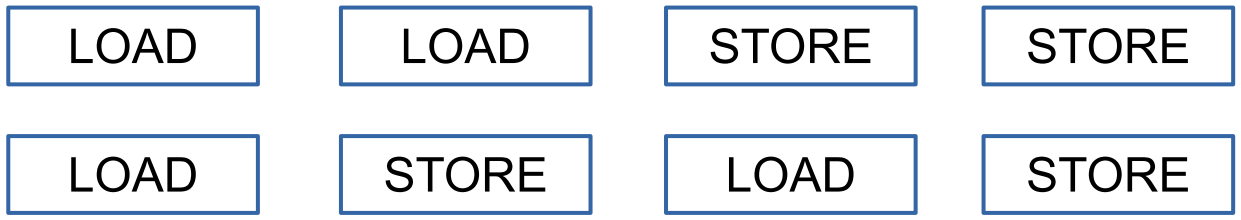


We can relax some of the access orders of each thread!

Total Store Order (TSO)
[Sun, new Intel]

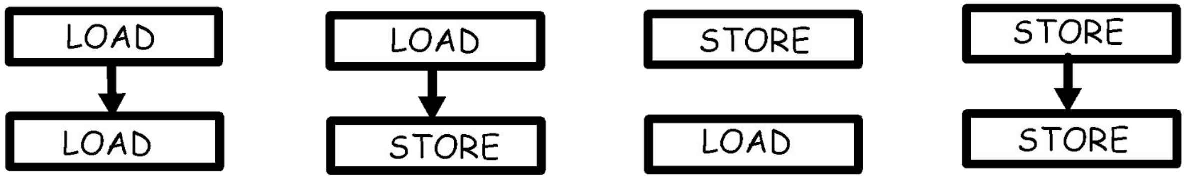


Relaxed Memory Order (RMO)
[Power,ARM,Alpha]



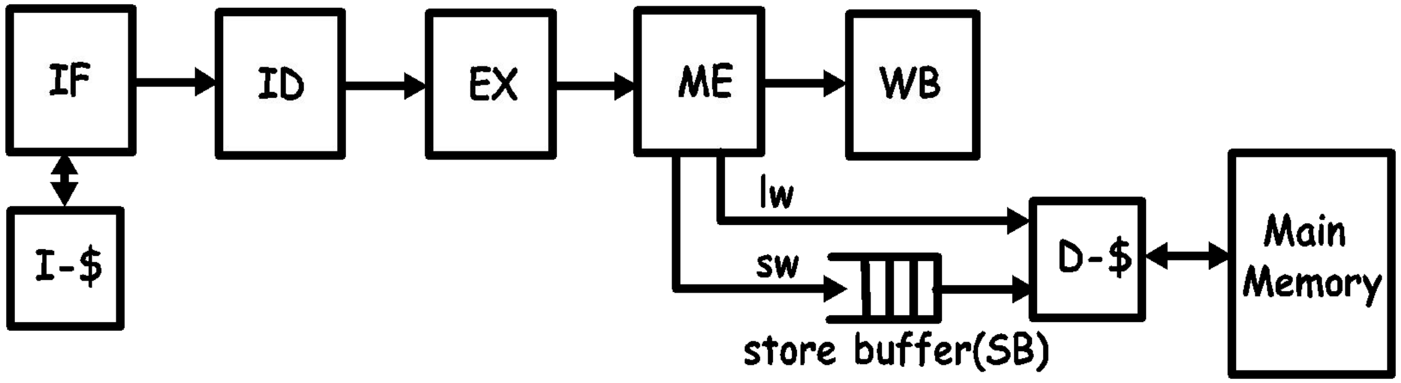
Relaxing store-to-load order

- 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 in-order processors?**
  - Assuming store atomicity: 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 to empty;

Relaxing Store-to-load Orders

- **Example: Sun Microsystems/x86 Total Store Order (TSO)**
  - Dekker's algorithm does not work (LOADs can bypass prior STOREs)
    - *remember: we saw this on my laptop!*
  - Point-to-point communication still works (because STORE-STORE and LOAD-LOAD are enforced)

- **Values may or may not be forwarded from SB to loads**
  - No forwarding from SB → strict coherent (store atomic)
    - Implementations: IBM 370
  - Forwarding from SB → plain coherent (case of TSO)
    - This means that some loads in TSO return values even if they are not GPed
    - Implementations: Sun TSO, Intel x86-TSO

RMO: Relaxed Memory Order (SUN, ARM, POWER, RISC-V)

- **In RMO only intra-thread memory dependency order is enforced**
  - as in all uniprocessors
- No implicit order between threads
- Enables more performance optimizations (most "hardware-friendly")

- **MEMBAR** (memory barrier) instructions specify orders explicitly. For example:
  - 4 bits are used to specify up to 4 orders
  - **LOAD-LOAD** → forces 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

RMO: Relaxed Memory Order (SUN, ARM, POWER)

- **MEMBARS are inserted by the compiler or programmer**
  - MEMBARs are extra instructions executed by single thread (unlike thread barrier synchronization)

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

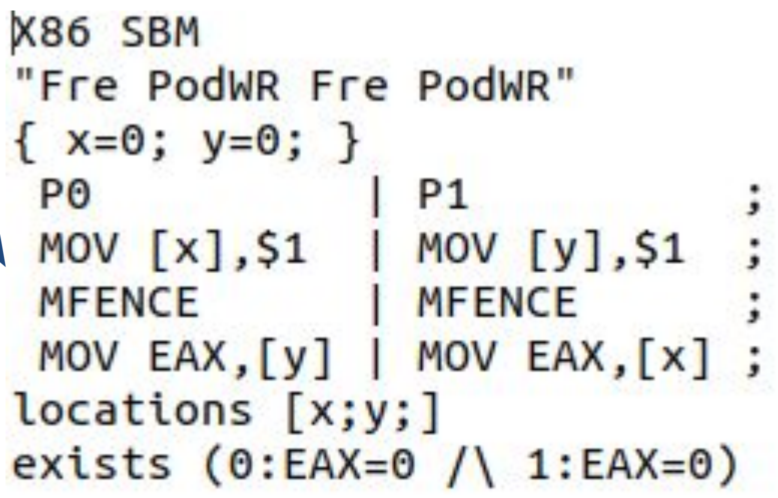
  - not sequentially consistent (yields non-SC outcomes)

```
T1              T2                 T3
A:=1;           while(A==0);       R2:=B;
                MEMBAR 0100        MEMBAR 1000
                B:=1;              R3:=A;
```

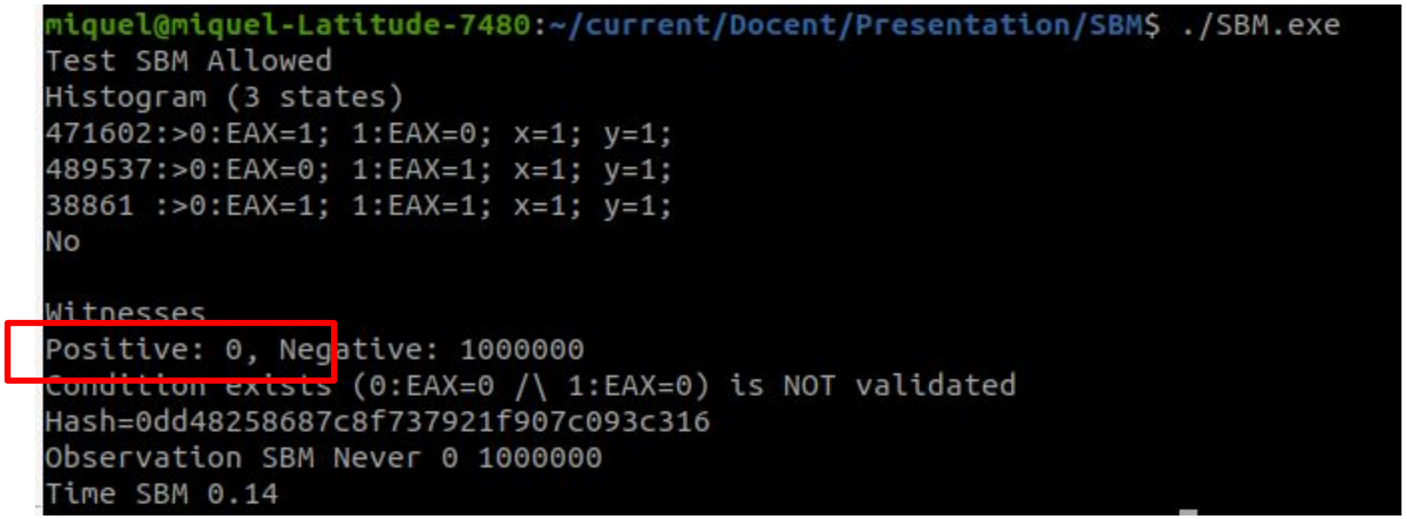
  - **with membars:** sequentially consistent (yields SC outcomes only)
    - `MEMBAR 0100: LOAD-STORE`
    - `MEMBAR 1000: LOAD-LOAD`

Also useful to enforce order in TSO

Can we solve the Dekker ordering problem by inserting a memory barrier (MFENCE)?
(note: Intel MFENCE instruction basically equivalent to `MEMBAR 1111`)



```
X86 SBM
"Fre PodWR Fre PodWR"
{ x=0; y=0; }
 P0            | P1            ;
 MOV [x],$1    | MOV [y],$1    ;
 MFENCE        | MFENCE        ;
 MOV EAX,[y]   | MOV EAX,[x]   ;
locations [x;y;]
exists (0:EAX=0 /\ 1:EAX=0)
```



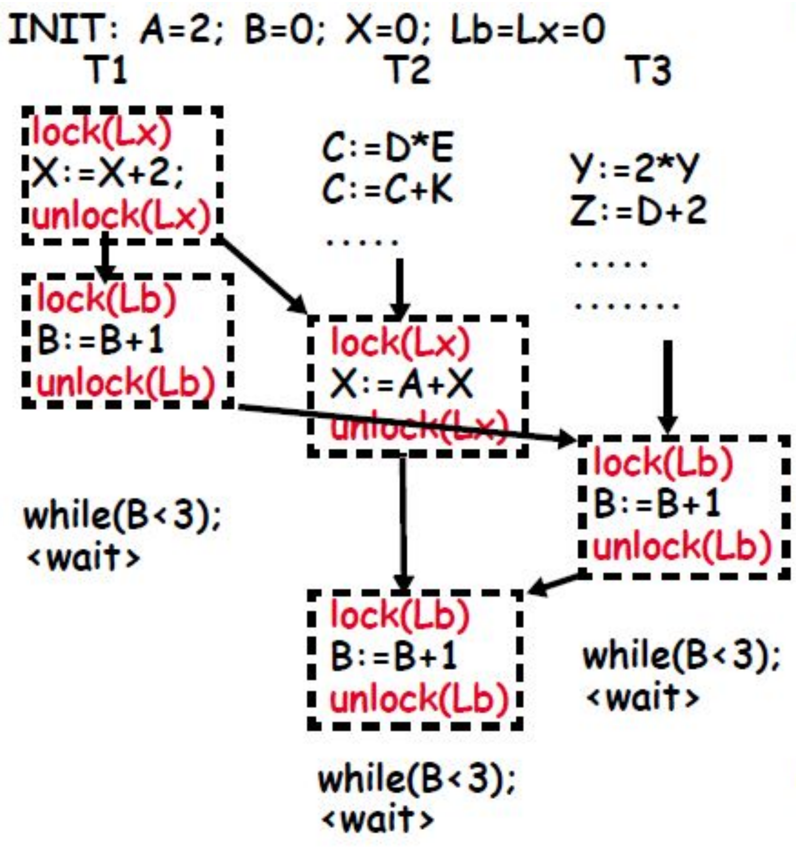
```
miquel@miquel-Latitude-7480:~/current/Docent/Presentation/SBM$ ./SBM.exe
Test SBM Allowed
Histogram (3 states)
471602:>0:EAX=1; 1:EAX=0; x=1; y=1;
489537:>0:EAX=0; 1:EAX=1; x=1; y=1;
38861 :>0:EAX=1; 1:EAX=1; x=1; y=1;
No

Witnesses
Positive: 0, Negative: 1000000
Condition exists (0:EAX=0 /\ 1:EAX=0) is NOT validated
Hash=0dd48258687c8f737921f907c093c316
Observation SBM Never 0 1000000
Time SBM 0.14
```

Yes:
Dekker works with TSO + **MFENCE**

MCM based on Synchronization: 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
  - <u>Hence, no need to enforce ordering within a phase</u>



MCM based on Synchronization: Weak Ordering

- Accesses to **synchronization data** (including all locks and shared data in synchronization protocols) must be treated differently by the hardware from accesses to other shared and private data.
    - They act as memory barriers on all accesses
    - Must globally perform all access preceding SYNC access in thread order (T.O.)
    - Must globally perform SYNC access before all following accesses in thread order (T.O.)

- Accesses to other (non-sync) shared and private data must enforce uniprocessor dependencies on same address

WEAK ORDERING

- Variables that are used for synchronization must be declared as such (e.g., flag, A and B below) or specific statements must be labeled or marked
  - so that execution on these variables is safe
  - to avoid compiler reordering

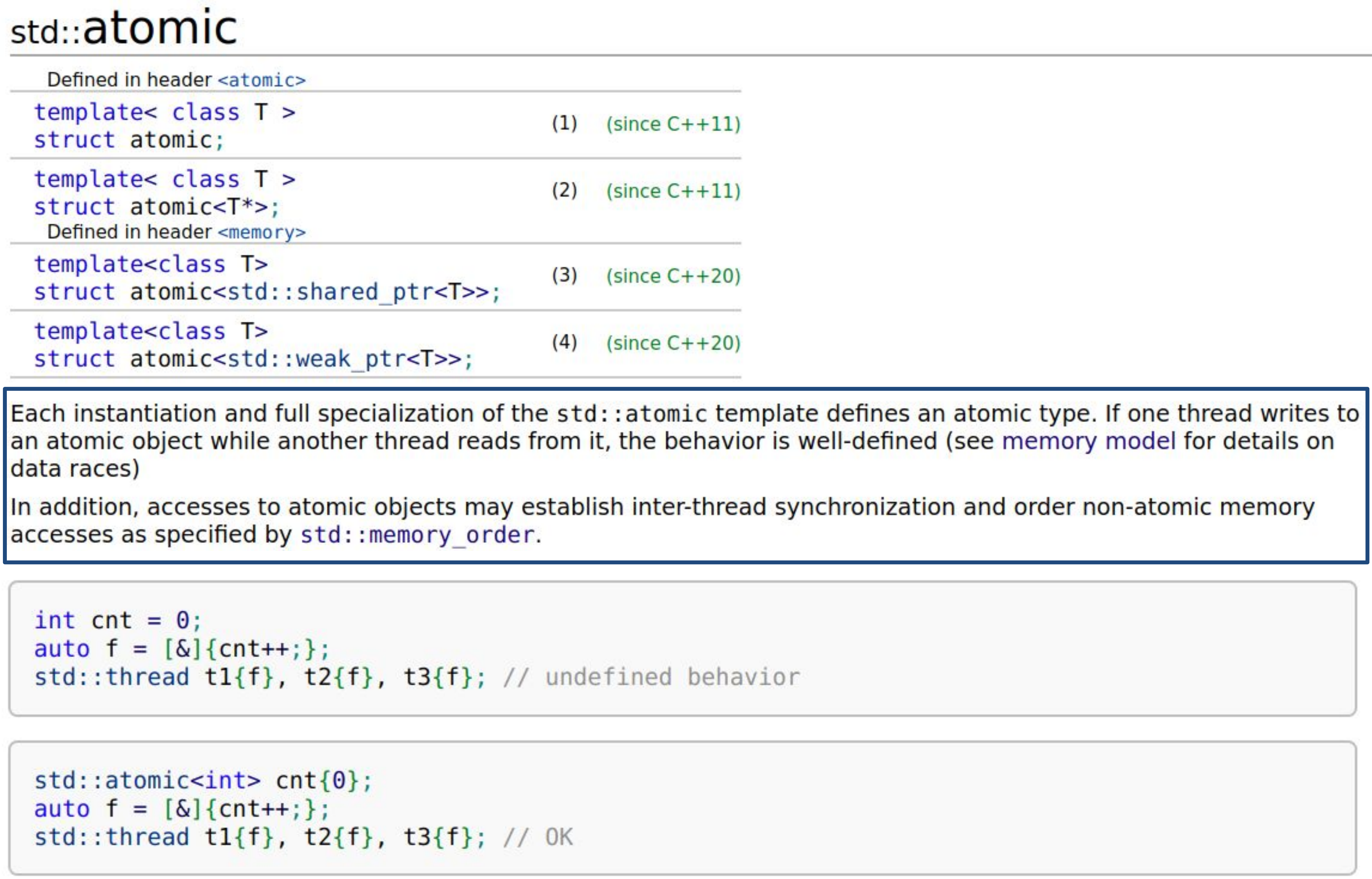
```
A=flag=0 initially
T1                             T2
A:=1;                          while(flag==0)do nothing;
flag:=1;                       print A;
 ...                            ...
```

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

Declaring synchronizing variables in C++11



std::atomic

Defined in header `<atomic>`		
`template< class T > struct atomic;`	(1)	(since C++11)
`template< class T > struct atomic<T*>;`	(2)	(since C++11)
Defined in header `<memory>`		
`template<class T> struct atomic<std::shared_ptr<T>>;`	(3)	(since C++20)
`template<class T> struct atomic<std::weak_ptr<T>>;`	(4)	(since C++20)

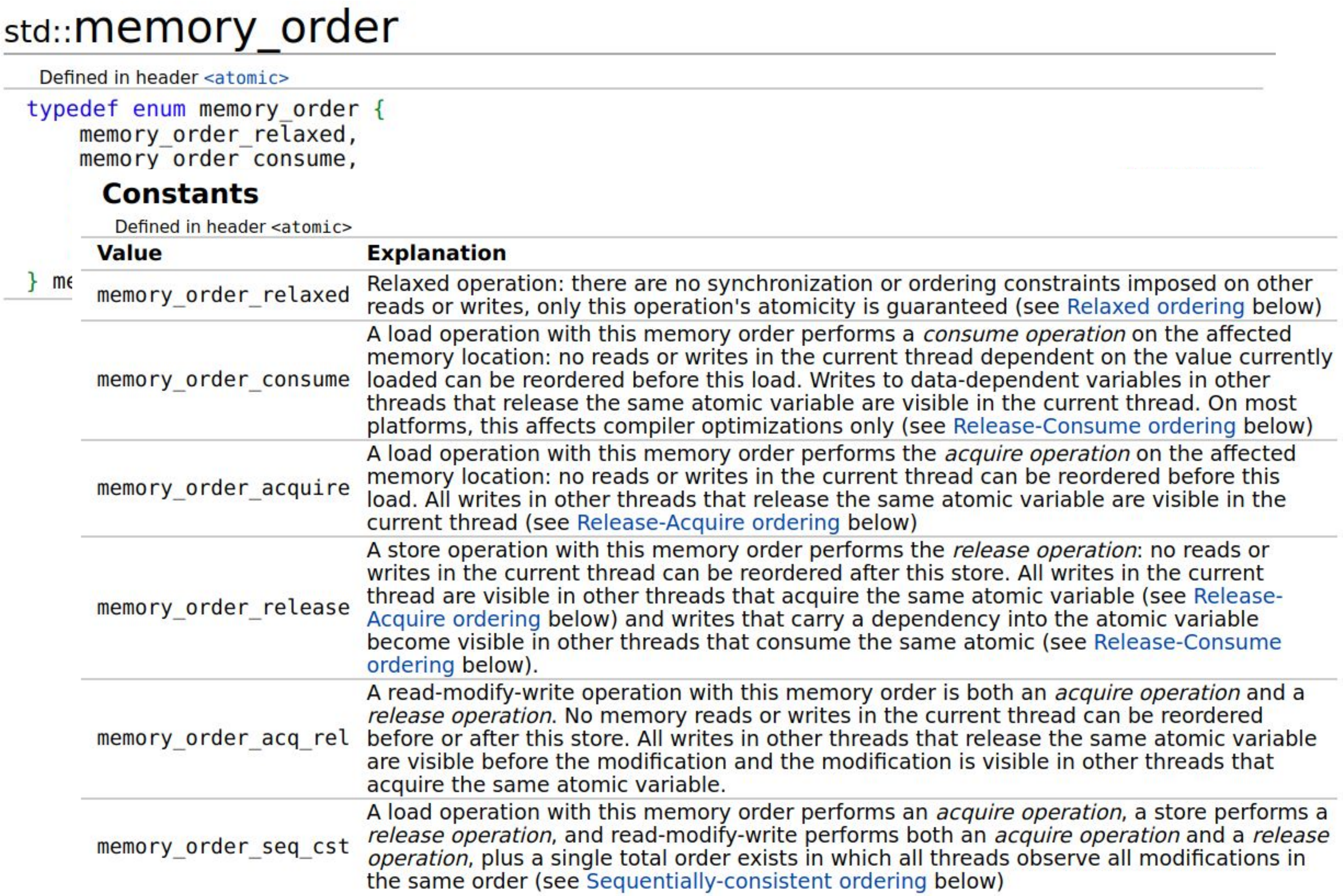
Each instantiation and full specialization of the `std::atomic` template defines an atomic type. If one thread writes to an atomic object while another thread reads from it, the behavior is well-defined (see memory model for details on data races)

In addition, accesses to atomic objects may establish inter-thread synchronization and order non-atomic memory accesses as specified by `std::memory_order`.

```
int cnt = 0;
auto f = [&]{cnt++;};
std::thread t1{f}, t2{f}, t3{f}; // undefined behavior
```

```
std::atomic<int> cnt{0};
auto f = [&]{cnt++;};
std::thread t1{f}, t2{f}, t3{f}; // OK
```

Memory Ordering in C++11



std::memory_order

Defined in header <atomic>

```
typedef enum memory_order {
    memory_order_relaxed,
    memory_order_consume,

} me
```

Constants

Defined in header <atomic>

Value	Explanation
`memory_order_relaxed`	Relaxed operation: there are no synchronization or ordering constraints imposed on other reads or writes, only this operation's atomicity is guaranteed (see Relaxed ordering below)
`memory_order_consume`	A load operation with this memory order performs a *consume operation* on the affected memory location: no reads or writes in the current thread dependent on the value currently loaded can be reordered before this load. Writes to data-dependent variables in other threads that release the same atomic variable are visible in the current thread. On most platforms, this affects compiler optimizations only (see Release-Consume ordering below)
`memory_order_acquire`	A load operation with this memory order performs the *acquire operation* on the affected memory location: no reads or writes in the current thread can be reordered before this load. All writes in other threads that release the same atomic variable are visible in the current thread (see Release-Acquire ordering below)
`memory_order_release`	A store operation with this memory order performs the *release operation*: no reads or writes in the current thread can be reordered after this store. All writes in the current thread are visible in other threads that acquire the same atomic variable (see Release-Acquire ordering below) and writes that carry a dependency into the atomic variable become visible in other threads that consume the same atomic (see Release-Consume ordering below).
`memory_order_acq_rel`	A read-modify-write operation with this memory order is both an *acquire operation* and a *release operation*. No memory reads or writes in the current thread can be reordered before or after this store. All writes in other threads that release the same atomic variable are visible before the modification and the modification is visible in other threads that acquire the same atomic variable.
`memory_order_seq_cst`	A load operation with this memory order performs an *acquire operation*, a store performs a *release operation*, and read-modify-write performs both an *acquire operation* and a *release operation*, plus a single total order exists in which all threads observe all modifications in the same order (see Sequentially-consistent ordering below)

Weak Ordering

- **A RMW atomic on a memory location is globally performed once both the LOAD and STORE in the RMW access are globally performed.**

- **SYNC operation must be recognizable by the hardware at the ISA level**
  - RMW (T&S, F&OP, CAS,...)
  - Special loads and stores for SYNC variable accesses

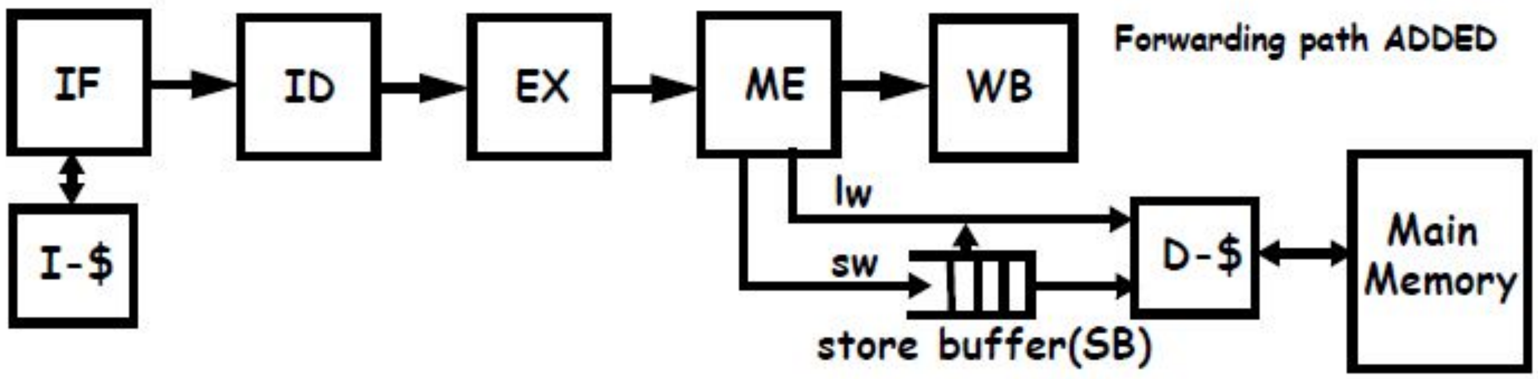
- **Orders to enforce:**

$$\begin{tikzcd}
\boxed{\text{Sync}} \arrow[d] & \boxed{\text{Op}} \arrow[d] & \boxed{\text{Sync}} \arrow[d] \\
\boxed{\text{Op}} & \boxed{\text{Sync}} & \boxed{\text{Sync}}
\end{tikzcd}$$

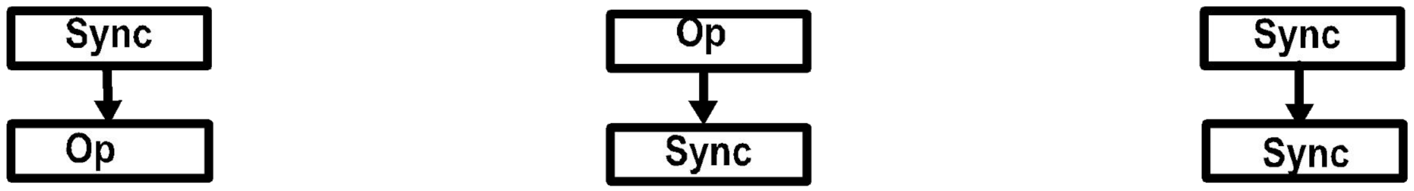
- OP = regular LOAD or STORE
- SYNC = any synchronization access, e.g., SWAP, T&S, special LOAD/STORE

WEAK ORDERING

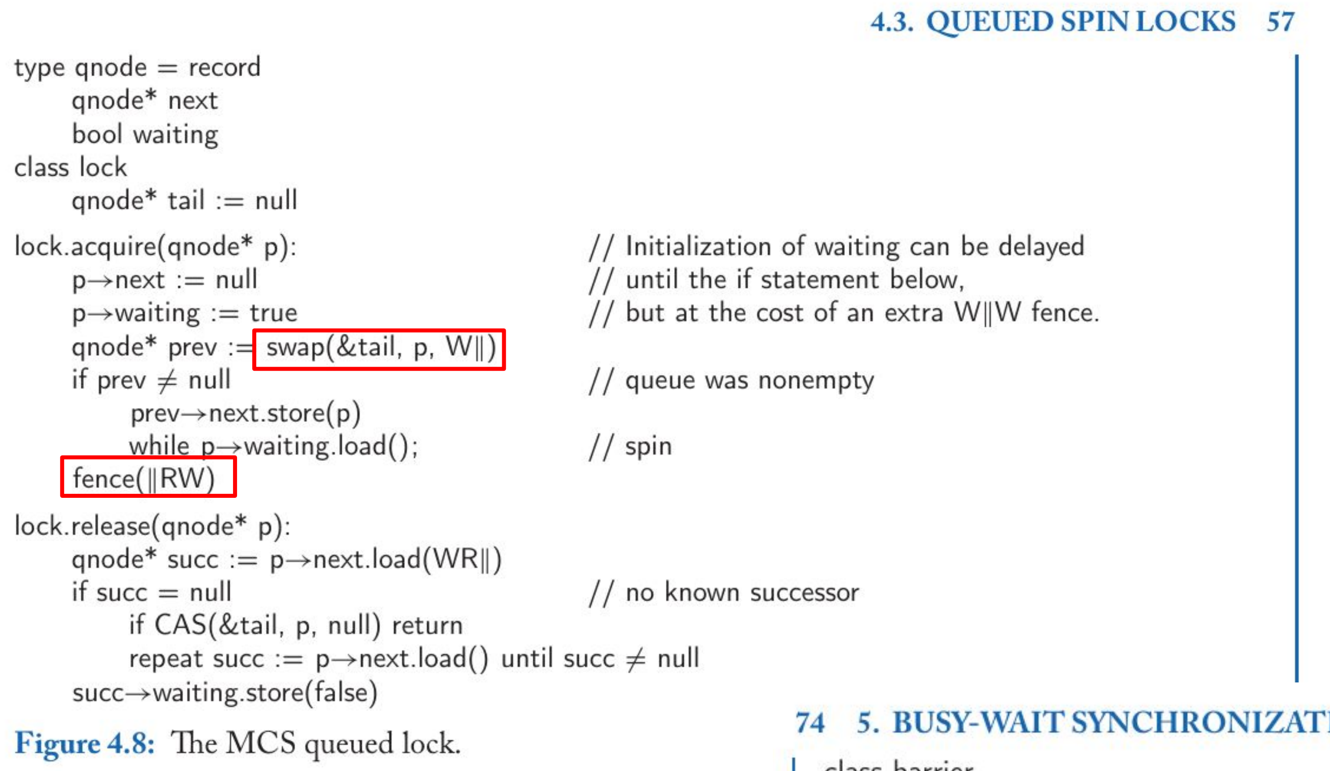
- **What does it mean for IN-ORDER processors?**
  - Note: here LOADs can return values even if they are not GPed



- **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 in-order processors (note: not the case for OoO -- need schemes to reconstruct order)



Orderings are critical for concurrent algorithms

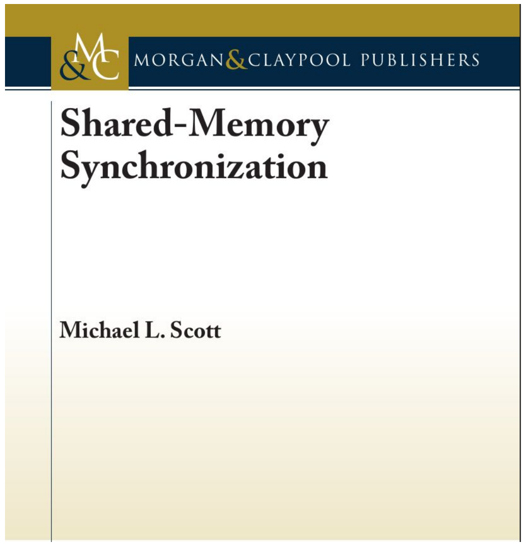


```
type qnode = record
    qnode* next
    bool waiting
class lock
    qnode* tail := null

lock.acquire(qnode* p):                      // Initialization of waiting can be delayed
    p→next := null                           // until the if statement below,
    p→waiting := true                        // but at the cost of an extra W‖W fence.
    qnode* prev := swap(&tail, p, W‖)
    if prev ≠ null                           // queue was nonempty
        prev→next.store(p)
        while p→waiting.load();              // spin
    fence(‖RW)

lock.release(qnode* p):
    qnode* succ := p→next.load(WR‖)
    if succ = null                           // no known successor
        if CAS(&tail, p, null) return
        repeat succ := p→next.load() until succ ≠ null
    succ→waiting.store(false)
```

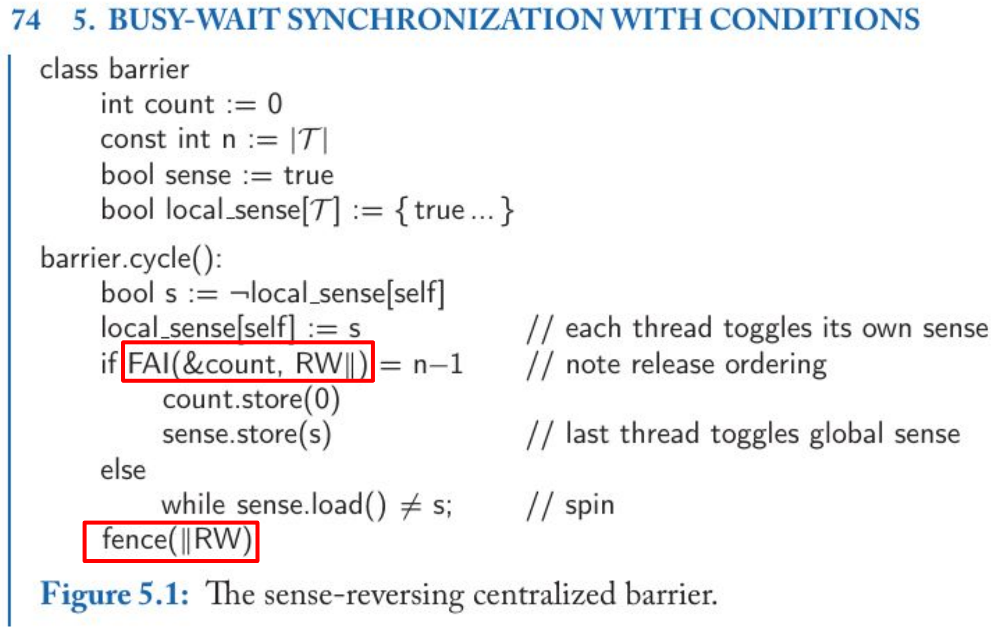
Figure 4.8: The MCS queued lock.



MORGAN & CLAYPOOL PUBLISHERS

Shared-Memory Synchronization

Michael L. Scott



```
class barrier
    int count := 0
    const int n := |T|
    bool sense := true
    bool local_sense[T] := { true ... }

barrier.cycle():
    bool s := ¬local_sense[self]
    local_sense[self] := s                   // each thread toggles its own sense
    if FAI(&count, RW‖) = n−1                // note release ordering
        count.store(0)
        sense.store(s)                       // last thread toggles global sense
    else
        while sense.load() ≠ s;              // spin
    fence(‖RW)
```

Figure 5.1: The sense-reversing centralized barrier.

<u>Specifies weakest semantics to use to correctly implement lock-free algorithms</u>:

- memory barriers (various orders)
- "relaxed" atomics (various orders)

Fully-SC atomics are almost never required, but use them (1) when ISA does not provide weaker instructions, or (2) when unsure :)

Summary

- **Sequential Consistency**
- **Memory Consistency Models**
    - Not relying on Synchronization (E.G. Store-load Relaxation)
    - Relying on Synchronization (E.G. Weak Ordering)