SOLUTIONS

Computer System Architecture
6.823 Quiz #3
May 1, 2020

Name: _____

90 Minutes
16 Pages

Notes:

- Not all questions are equally hard. Look over the whole quiz and budget your time carefully.
- Please state any assumptions you make, and show your work.
- Please write your answers by hand, on paper or a tablet.
- Please email all 16 pages of questions with your answers, including this cover page. Alternatively, you may email scans (or photographs) of separate sheets of paper. Emails should be sent to 6823-staff@csail.mit.edu
- Do not discuss a quiz's contents with students who have not yet taken the quiz.
- Please sign the following statement before starting the quiz. If you are emailing separate sheets of paper, copy the statement onto the first page and sign it.

I certify that I will start and finish the quiz on time, and that
I will not give or receive unauthorized help on this quiz.

Sign here: _____

Part A	_____	15 Points
Part B	_____	25 Points
Part C	_____	25 Points
Part D	_____	35 Points
TOTAL	_____	**100 Points**

Part A: Networks (15 points)

Question 1 (6 points)

Consider a **fully connected** network topology with N nodes, where each node is directly connected to all (N-1) other nodes.

(a) What is the total number of network links?

N*(N-1)/2

Note that we count each bi-directional link only once.

(b) What is the diameter of the network?

1

(c) What is the bisection bandwidth this network? You may assume N is even, and only consider bisections that divide the number of nodes into equal halves.

Consider the N/2 nodes on one side of the partition. Each of those nodes will have N/2 links connecting to nodes on the other side.

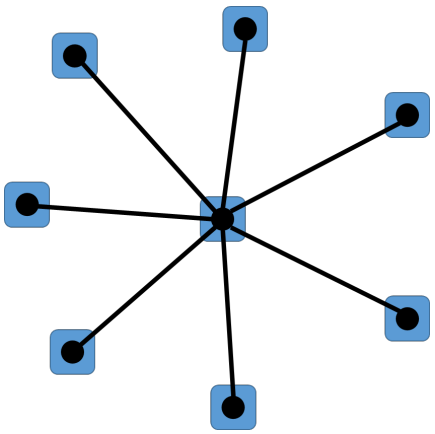
$N/2 * N/2 = N^2 / 4$

Alternative solution method: The bisection yields two fully-connected components of size N/2, so using the formula from part (a), the cut must have removed a number of links equal to the difference between a single network of size N and two networks of size N/2:

$N*(N-1)/2 - 2 * (N/2)*(N/2 - 1)/2 = N^2 / 4$

Question 2 (9 points)

Consider a **star** topology, where only a central node is connected to all (N-1) other nodes. The diagram to the right shows a star topology with N=8 nodes for illustration purposes. When answering the questions below, provide answers for N-node star networks (*not* for N=8).



(a) What is the total number of network links?

N-1

(b) What is the diameter of the network?

2

(c) What is the bisection bandwidth this network? You may assume N is even, and only consider bisections that divide the number of nodes into equal halves.

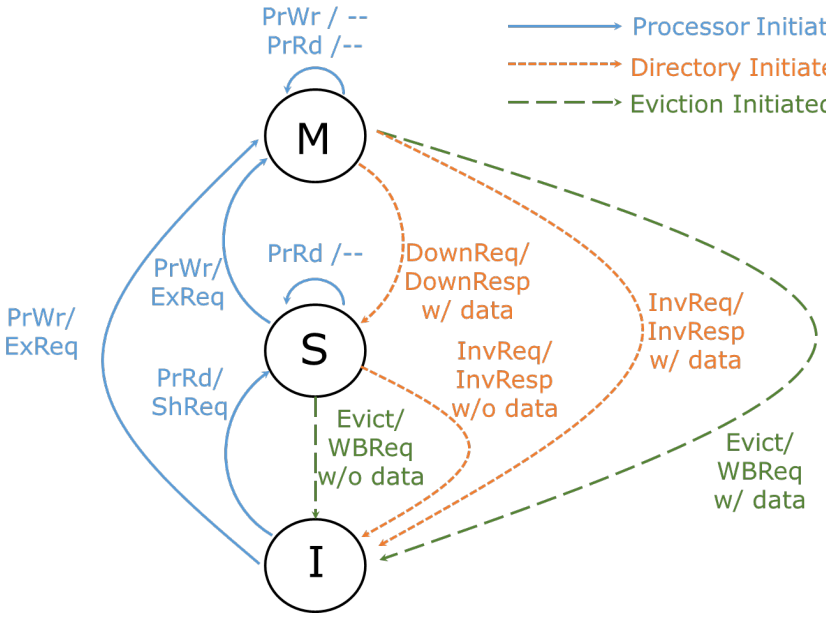
N/2

(d) Assume a dedicated buffer to receive flits at each end of each bidirectional link. If 180-degree turns are prohibited in this topology and messages are routed on minimal-length paths, can deadlock occur in this network?

No. The network has no cycles.

Part B: Cache Coherence (25 points)

Ben Bitdiddle wants to study design tradeoffs in a directory-based MSI coherence protocol. Ben starts by considering the directory-based MSI protocol presented in lecture. The protocol is described in the quiz handout and summarized in this cache-side state transition diagram:



In this protocol, evictions of a cache line in the S state require sending a WBReq (without data) to notify the directory, so the directory can remove the cache from the sharer set.

Ben thinks that he can reduce the number of messages sent on the network during cache evictions. Ben wants to **silently drop** cache lines when evicting cache lines in the S state, sending no message on the network. This means that a cache line can move from S to I without informing the directory.

Question 1 (4 points)

Consider a machine with two cores, where each core has a private cache that uses Ben's proposal for silent drops. Suppose a cache line A is in S state and it is in Core0's cache. Core 0's cache evicts line A, silently dropping it. The directory still has Core 0 in the sharer set for cache line A.

(a) Consider that after the silent drop by Core 0's cache, Core 0 performs a read of the evicted cache line A. To reobtain the cache line, Core 0's cache sends a ShReq to the directory. Assume there have been no writes to cache line A. What network message, if any, should the directory send to respond to the ShReq to make the protocol work?

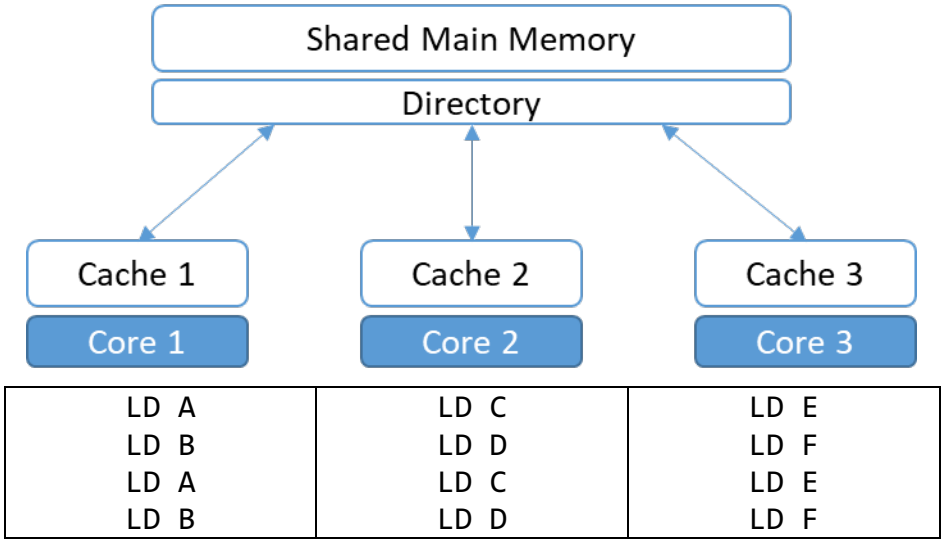
The directory should a ShResp with the data to Core 0's cache. (The directory stays in the shared state, and Core 0 remains a sharer.)

(b) Consider that after the silent drop by Core 0's cache, Core 1 performs a write to cache line A. Core 1's cache sends an ExReq to the directory. Since Core 0 is in the sharer set, the directory sends an InvReq to Core 0. What network message, if any, should Core 0's cache send when receiving the InvReq while the requested cache line is in the I state?

Core 0's cache should send an InvResp without data. (It is already in I so no state transition is necessary.)

Question 2 (6 points)

Consider the three-core system below. Each core has a private cache that *can only hold a single cache line*, and the caches start out empty. Each core runs a thread that performs four reads, alternating between reading two addresses. Each thread accesses different addresses on different cache lines. (Core 1's thread reads addresses `A` and `B`, Core 2's thread reads `C` and `D`, etc.) Due to evictions, all 12 accesses will be cache misses. *Assume the directory has unlimited capacity.*



(a) How many writeback requests are sent in the original MSI protocol from lecture?

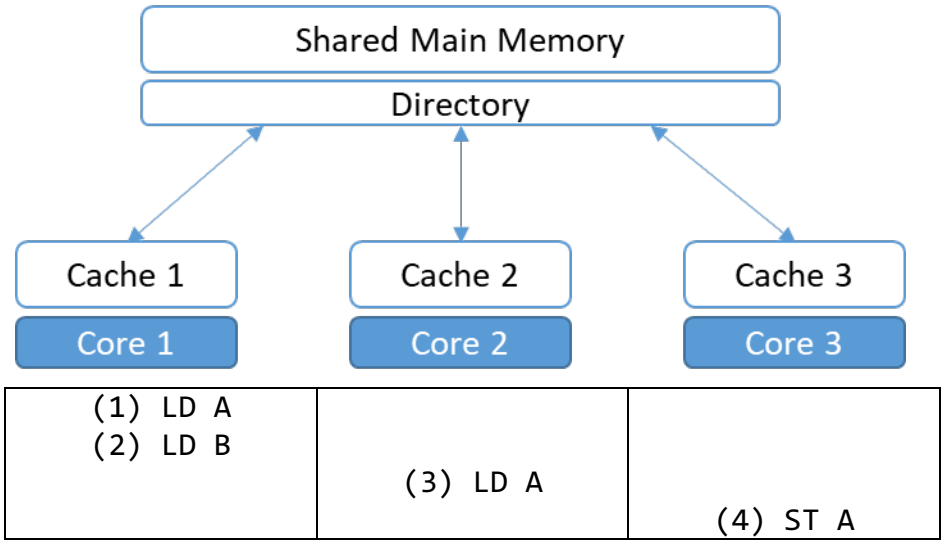
9 (Each core performs 3 evictions. The final line read by each core can stay in the cache.)

(b) How many writeback requests are sent with Ben's proposal for silent drops?

Zero

Question 3 (6 points)

Consider a different workload where the threads access shared data, as shown below. The number in parenthesis indicates the global order of the accesses (i.e. Core 1's `LD A` happens before `LD B`, which happens before Core 2's `LD A`, etc.). Each access completes before the next one begins. Again, assume all caches start empty and each cache can only hold a single line at a time.



(a) How many WBReq and InvReq messages are sent in the original MSI protocol from lecture? Count an invalidation of multiple caches as multiple requests. Do **not** count response messages.

1 writeback due to eviction + 1 invalidation (= 2 total)

(b) How many writeback requests and invalidation requests are sent with Ben's proposal for silent drops?

No writeback messages on evictions + 2 invalidations (= 2 total)

(Cache 3's store triggers invalidations to caches 1 and 2. This is similar to the scenario described earlier in problem 1(b).)

Question 4 (9 points)

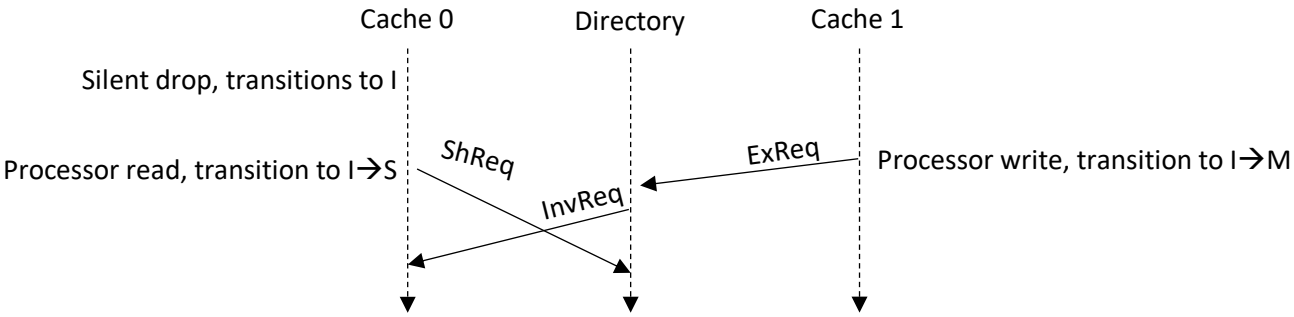
So far, we assumed each coherence transaction completes before the next transaction begins. Alyssa P. Hacker points out that Ben's silent drops make it harder to solve races when there are concurrent coherence requests. To see this, we will consider two scenarios. In each scenario, Core 0 silently drops a line from its private cache that it later needs to read, while Core 1 attempts to write to the **same cache line**. In each scenario, Core 0 receives an InvReq, and you must pick **one** of the three following answers:

A: Acknowledge the InvReq by sending an InvResp, remaining in the I→S transient state to wait for a later ShResp.

B: Buffer or NACK the InvReq, waiting for a ShResp to first serve its read before performing an invalidation.

C: Performing either of A or B will result in correct behavior.

(a) In this scenario, the directory receives Cache 1's ExReq before Cache 0's ShReq. While Cache 0 is waiting for a ShResp, it receives a InvReq from the directory.

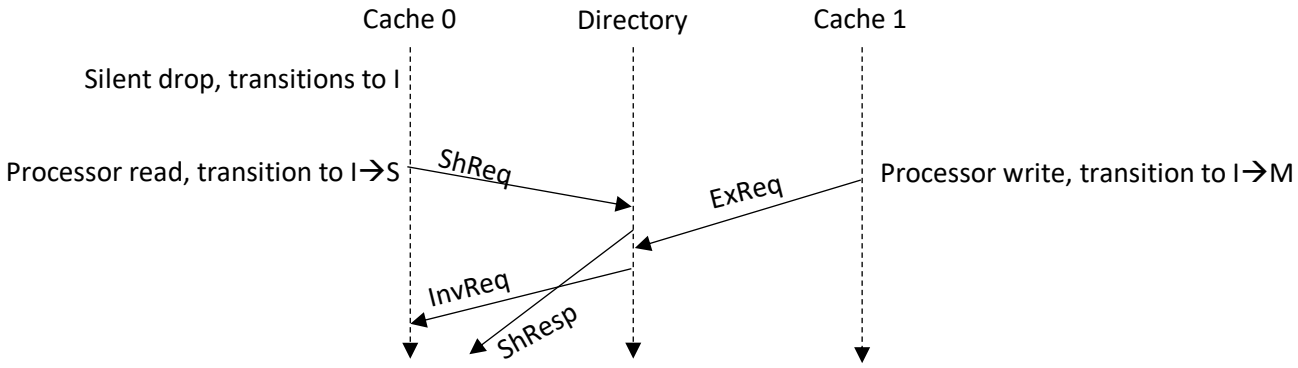


To maintain coherence, what action should Cache 0 take in response to the InvReq while in the I→S transient state?

A

(B causes deadlock as the directory has decided to serve the ExReq first, and will not send a ShResp in response to the ShReq until after it receives acknowledgement of the invalidation.)

(b) In this scenario, the directory receives Cache 0's ShReq before Cache 1's ExReq. The directory sends a ShResp to Core 0 followed by a InvReq. However, the ShResp is traveling slowly in the network, and Cache 0 receives the InvReq before the ShResp.



To maintain coherence, what action should Cache 0 take in response to the InvReq while in the I→S transient state?

B

(A violates coherence because Cache 0 may end up forever holding stale data from the ShResp. The directory and Cache 1 will think Cache 0 has invalidated the data, and may send no more invalidations.)

Part C: Memory Consistency (25 points)

Consider a shared-memory machine that executes the following two threads on two different cores. Assume that memory locations `a` and `b` contain initial value `0`.

T1	T2
T1.1: Store (a) ← 1 T1.2: Load r1 ← (a) T1.3: Load r3 ← (b)	T2.1: Store (b) ← 1 T2.2: Load r2 ← (b) T2.3: Load r4 ← (a)

Question 1 (5 points)

If the machine implements **sequential consistency**, what execution outcomes (i.e., values of r1, r2, r3, and r4) can this code produce?

Note: You can but *do not have to* express the result as (r1, r2, r3, r4) tuples.

r1 = 1
r2 = 1
r3, r4 can be any of (0,1) (1,0) or (1,1)

Question 2 (4 points)

If the machine implements the **Total Store Order (TSO)** consistency model, what execution outcomes (i.e., values of r1, r2, r3, and r4) can this code produce?

Note: You can but *do not have to* express the result as (r1, r2, r3, r4) tuples.

r1 = 1
r2 = 1
r3, r4 can be anything

Note that TSO allows the following execution:
T1.2 < T1.3 < T1.1 while requiring T1.2 to return the value stored by T.1.1 due to store-load forwarding. ("<" means happens-before.)

Question 3 (4 points)

If the machine implements a **relaxed consistency model, RMO**, which allows loads and stores to be reordered after later loads and stores, what execution outcomes (i.e., values of r1, r2, r3, and r4) can this code produce?

Note: You can but *do not have to* express the result as (r1, r2, r3, r4) tuples.

Same as TSO.

Store-load forwarding typically exists, so r1 and r2 are still 1s.
(0, 0) for (r3, r4) is valid because this simply requires store-load reordering, which is already allowed by TSO.

Question 4 (4 points)

The relaxed consistency model (RMO) has the following fine-grained barrier instructions:

- **MEMBAR$_{RR}$** guarantees that all reads that precede MEMBAR$_{RR}$ in program order will be performed before any read that follows the barrier.
- **MEMBAR$_{RW}$** guarantees that all reads that precede MEMBAR$_{RW}$ in program order will be performed before any write that follows the barrier.
- **MEMBAR$_{WR}$** guarantees that all writes that precede MEMBAR$_{WR}$ in program order will be performed before any read that follows the barrier.
- **MEMBAR$_{WW}$** guarantees that all writes that precede MEMBAR$_{WW}$ in program order will be performed before any write that follows the barrier.

Add barrier instructions to T1 and T2 so that the RMO machine produces the same outputs as the SC machine for this code. Use the ***minimum*** number of memory barrier instructions. List the locations of each barrier below (e.g., "Add **MEMBAR$_{RR}$** after `T1.1`").

T1	**T2**
T1.1: Store (a) ← 1 T1.2: Load r1 ← (a) T1.3: Load r3 ← (b)	T2.1: Store (b) ← 1 T2.2: Load r2 ← (b) T2.3: Load r4 ← (a)

Place a MEMBAR$_{WR}$ between T1.1 and either T1.2 or T1.3 for T1. Same for T2.

Note that MEMBAR$_{RR}$ is not helpful since it does not prevent reordering T1.3 and T1.1.

Question 5 (3 points)

Consider a shared-memory machine that executes the following four threads on four cores. Assume that memory location `a` contains initial value `0`.

T1	T2	T3	T4
Store (a) ← 1	Store (a) ← 2	Load r1 ← (a) Load r2 ← (a)	Load r3 ← (a) Load r4 ← (a)

If the machine implements the **TSO** consistency model, can it produce the following execution outcome (r1, r2, r3, r4) = (1, 2, 2, 1) ?

No. Stores from T1 and T2 should appear in the same order for T3 and T4.

Question 6 (3 points)

Ben Bitdiddle modifies the above TSO machine. The original machine has one thread per core. Ben implements multi-threading, making each core support 2 thread contexts. The threads running on the same core share a single committed store buffer.

This machine executes the four threads in Question 5. T1 and T3 run on Core 0, and T2 and T4 run on Core 1. Can this machine produce the following execution outcome (r1, r2, r3, r4) = (1, 2, 2, 1)?

Yes. The shared committed store buffer allows T3 to observe T1's store first, and allows T4 to observe T2's store first.

Question 7 (2 points)

Does the machine described in Question 6 still maintain TSO?

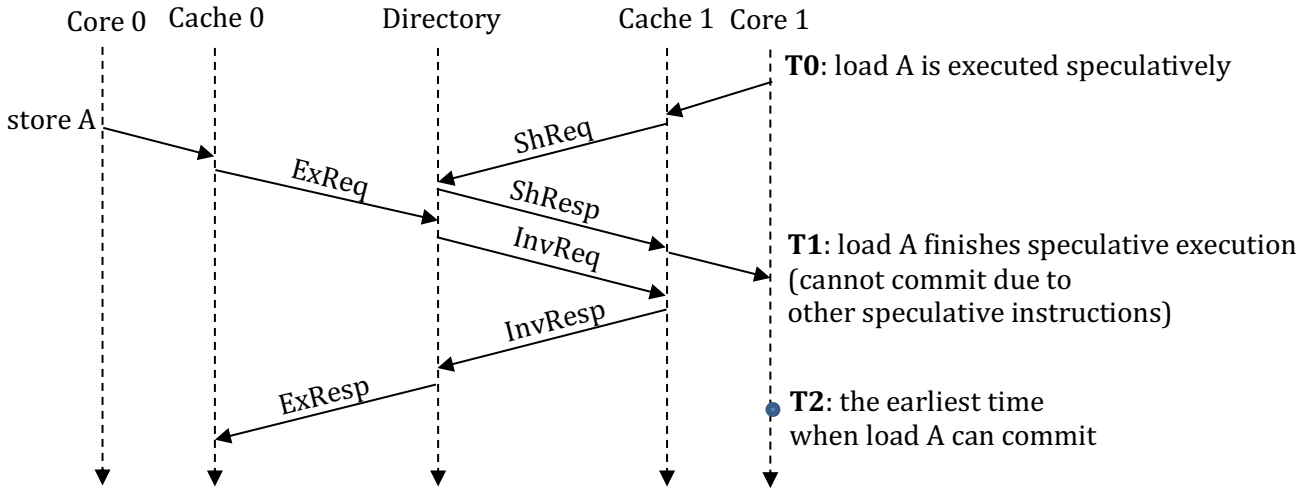
No. Compare the answers in Q5 and Q6.

Part D: Consistency with Speculative Execution (35 points)

Ben Bitdiddle is building a multi-core processor and he wants to implement the ***sequential consistency*** model. The processor supports out-of-order execution and speculative execution. Following the standard out-of-order execution mechanism, all the instructions running on a core, including load instructions, can be issued and executed out of program order, but are always committed in program order. Each core has a private cache. All the private caches are kept coherent using a *directory-based MSI* coherence protocol.

Ben adopts the SC scheme we explained in lecture: speculative loads can issue out of order. To check whether speculation is correct or not, each core monitors invalidation requests received by its private cache. If the core receives an invalidation request to a cache block accessed by a load instruction that has received its data from the cache but has not yet reached its commit point, the load is squashed and the core rolls back and re-executes from the load instruction. Otherwise, the load successfully commits. Note that stores are always non-speculative. In addition, for simplicity, while a store is waiting for a coherence response, other loads or stores are not allowed to execute. We call this implementation **Machine 0**.

Ben analyzes the following scenario on Machine 0. Initially neither cache 0 nor cache 1 has the data in address A. Core 1 speculatively executes the load instruction "load A" at T0, receives the data from the cache at T1. T2 is the earliest time when the load reaches the head of the ROB. Meanwhile, shortly after T0, Core 0 commits a store to the same location, and sends an invalidation request to Cache 1. The timeline of the execution of the cores and cache coherence transactions is shown below.



Question 1 (6 points)

Consider the above event sequence on Machine 0. Can the load commit successfully without re-execution (re-sending coherence messages)?

No. The accessed line has been invalidated.

Question 2 (6 points)

Does Machine 0 guarantee that all load instructions will eventually commit, i.e., a load cannot be prevented from committing indefinitely? If all loads can make forward progress, briefly explain why; otherwise, give a counterexample.

Hint: You could slightly modify the example (e.g., by adding more instructions) for analysis.

No. Consider a case where Core 0 repetitively issues stores to A, the load on Core 1 may never commit. (This may be called *starvation*.)

Ben changes the machine to create a new machine, called **Machine 1**. Machine 1 eliminates the speculation check logic from Machine 0, allowing a load to commit without re-execution even if the cache block accessed by the load is invalidated before its commit time. This allows a load to commit despite having read a value that may have been changed before its commit time.

Question 3 (5 points)

Does Machine 1 maintain cache coherence? Briefly explain why or why not.

Yes. Write propagation and write serialization are both preserved. It is just a matter of reordering some coherence transactions.

Question 4 (5 points)

Does Machine 1 implement sequential consistency? Briefly explain why or why not.

No. Core 1's load is effectively ordered earlier: it may precede some earlier loads and stores. SC does not allow any kind of such store-load or load-load reordering.

Question 5 (4 points)

Does Machine 1 implement TSO? Briefly explain why or why not.

No. Though stores are performed in order, loads can be reordered w.r.t. earlier loads. (TSO only allows loads to be reordered before earlier stores.) Such load-load reordering breaks TSO.

Ben modifies Machine 0 to produce a new design, **Machine 2**. Machine 2 keeps the speculation check logic from Machine 0, but relaxes it slightly: it allows a load to commit despite receiving an invalidation if the load was the oldest speculative memory instruction in program order when it started execution.

Question 6 (5 points)

Does Machine 2 implement sequential consistency? Briefly explain why or why not.

Yes. Sequential consistency is maintained because the load cannot be reordered to precede any previous (in program order) loads or stores.

Question 7 (4 points)

Does Machine 2 guarantee that all load instructions will eventually commit, i.e., a load cannot be prevented from committing indefinitely? If all loads can make forward progress, briefly explain why; otherwise, give a counterexample.

Yes. By ensuring that the earliest load can commit, forward progress is guaranteed.