Computer System Architecture
6.823 Quiz #2
November 12th, 2021

Name: _____SOLUTIONS_____

This is a closed book, closed notes exam.
80 Minutes
14 Pages (+2 Scratch)

Notes:

- Not all questions are of equal difficulty, so look over the entire exam and budget your time carefully.
- Please carefully state any assumptions you make.
- Show your work to receive full credit.
- Please write your name on every page in the quiz.
- You must not discuss a quiz's contents with other students who have not yet taken the quiz.
- Pages 15 and 16 are scratch pages. Use them if you need more space to answer one of the questions, or for rough work.

Part A	_____	15 Points
Part B	_____	30 Points
Part C	_____	25 Points
Part D	_____	30 Points
TOTAL	_____	**100 Points**

Part A: Multithreading (15 points)

Ben Bitdiddle is designing an out-of-order superscalar processor that supports simultaneous multithreading up to two threads. He is considering different design options for the processor's **committed store buffer**. The committed store buffer holds data for stores that have committed until they are written to the L1 cache in program order. An entry is allocated in the store buffer when a store instruction commits, and it is freed when the corresponding value is written to the L1 cache.

Ben's processor executes two threads, A and B, with the following characteristics:

- Thread A contains 1 store every 4 instructions. Each store takes on average 8 cycles between commit and writing to the L1 cache.
- Thread B contains 1 store every 5 instructions. Each store takes on average 20 cycles between commit and writing to the L1 cache.

Question 1 (8 points)

Ben first considers a *shared* store buffer design with 6 entries, where the two threads are both allowed to allocate entries on a single store buffer. The processor uses a *round-robin* fetch policy, which results in each thread committing 1 instruction per cycle. How many store buffer entries are used on average?

Recall Little's Law: Average number of entries used = Average throughput * Average Latency.

For Thread A, entries used = (1 IPC) * (1 store / 4 instr) * (8 cycles) = 2 entries
For Thread B, entries used = (1 IPC) * (1 store / 5 instr) * (20 cycles) = 4 entries

Thus, a total of 6 entries are used on average.

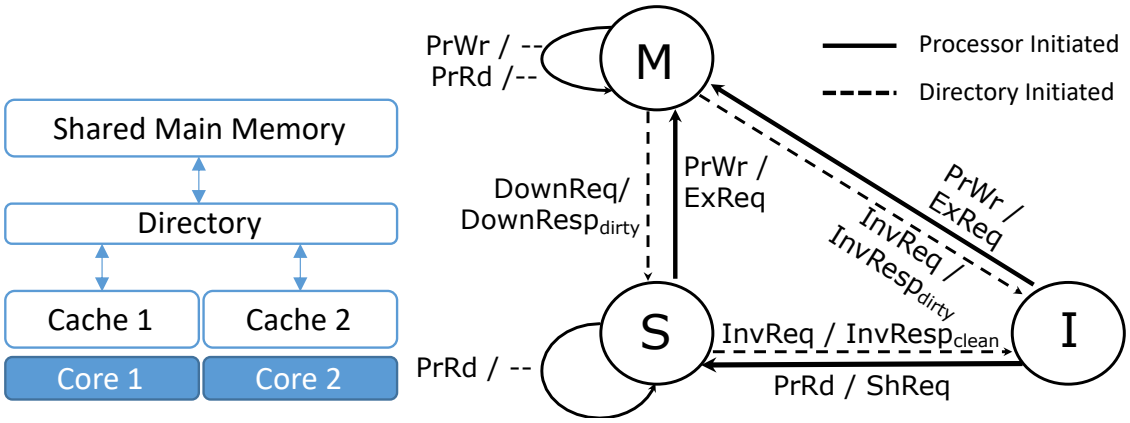
Question 2 (7 points)

Ben now considers a split store buffer design where each thread uses a separate 3-entry store buffer. By how much does the overall throughput decrease compared to the shared store buffer in Question 1?

Notice that we are using round-robin policy, so the processor will equalize the throughput to the lowest one amongst all threads. Thread A is unaffected, but thread B has only 3/4 of the needed entries, thus its throughput decreases to 3/4 IPC. Hence, total throughput is 1.5 IPC total.

Part B: Cache Coherence (30 points)

Ben Bitdiddle is designing a 2-core processor where each core has a private, local cache with shared main memory. The private caches have 4-byte lines and are kept coherent by a directory-based MSI Protocol. The diagrams below show the block diagram of the system (left) and the coherence protocol's processor-side state-transition diagram (right).



Ben wishes to add a **next-line prefetcher** to each core to reduce the overall number of cache misses in his programs. The prefetcher works by monitoring the cache misses out of the processor. When the processor misses on cache line X *and* the prefetcher predicts that the next cache line (at address X+4, since lines are 4 bytes long) will soon be accessed by the core, the prefetcher signals the processor to send a `ShReq` for the next cache line *in addition* to the original coherence request for cache line X.

Question 1 (5 points)

Consider the following sequence of accesses happening in chronological order. Assume that the coherence transaction initiated by a core's read/write finishes before the next core action, *including prefetches*.

Core 1: read A
Core 1: read A+4
Core 2: write A+4

Assume that all private caches are initially invalid, and Core 1's prefetcher signals the core to send a `ShReq` for line A+4 upon a cache miss to line A.

(a) Fill in the table below indicating how coherence states for the two cache lines A and A+4 change after each access.

Standard MSI Protocol				
	Core 1's Cache		Core 2's Cache	
Initial State	A: I	A+4: I	A: I	A+4: I
After Core 1 reads A	A: S	A+4: S	A: I	A+4: I
After Core 1 reads A+4	A: S	A+4: S	A: I	A+4: I
After Core 2 writes A+4	A: S	A+4: I	A: I	A+4: M

(b) How many coherence requests (`ShReq`, `ExReq`, `DownReq`, `InvReq`) were processed throughout the sequence of accesses?

4 requests: 2 ShReqs, 1 ExReq and 1 InvReq

Question 2 (5 points)

Now consider the following sequence:

Core 1: read A
Core 2: write A+4
Core 1: read A+4

Again, assume that all private caches are initially invalid, and Core 1's prefetcher signals the core to send a `ShReq` for line A+4 upon a cache miss to line A. How many coherence requests (`ShReq`, `ExReq`, `DownReq`, `InvReq`) were processed throughout this new sequence of accesses?

6 requests:
- 2 ShReqs by Core 1's first read
- 1 ExReq and 1 InvReq to A+4 initiated by Core 2
- 1 ShReq and 1 DownReq to A+4 initiated by Core 1

Ben wants to systematically characterize the effects of prefetches. He first classifies prefetches according to their *usefulness* to the requesting core:

- **Useful**: The prefetched line is referenced by the requesting core before it is evicted (either due to a conflict, or due to a write to the line from another core).
- **Useless**: The prefetched line is *not* referenced by the requesting core before it is evicted.

Ben further classifies useless prefetches into two sub-categories according to their *harmfulness* to the remote core:

- **Harmless**: A useless prefetch is initially classified as harmless unless it satisfies the harmfulness criteria specified below.
- **Harmful**: The prefetch downgraded a remote core's copy of the cache line, and that core subsequently wrote to the line before it was evicted.

Note that this means a prefetch can be categorized into one of three classes: Useful, Harmless (implying Useless), and Harmful (also implying Useless).

Question 3 (5 points)

Consider the sequences from Questions 1 and 2. How would you classify the two prefetches according to Ben's categorization: Useful, Harmless, or Harmful?

1: Useful since the line is referenced without any intervening invalidations or evictions.
2: Harmless since while it is Useless, it did not downgrade a remote copy.

Question 4 (5 points)

Describe a sequence of accesses (like those in Question 1 and 2) that cause a *Harmful* prefetch in Ben's system. All cache lines should start invalid at the beginning of your sequence, and the prefetched line should be in an invalid state at the end of the sequence.

How many extra coherence requests does the prefetch incur for your sequence compared to a 2-core system that does not implement prefetching?

Many possible answers here. One possible one is:

Core 2: write A+4
Core 1: read A <--- prefetch A+4 here
Core 2: write A+4

Ben modifies his coherence protocol to reduce the number of Harmful prefetches. Now, all `ShReq`s also contain a *prefetch bit* that is set if the `ShReq` was sent due to a prefetch from a core. Since `ShReq`s with the prefetch bit set are speculative, the directory decides how to process this request as follows: If no other core holds a copy of the line in the modified (M) state, the directory replies with a `ShResp` since the prefetch can never become Harmful. Otherwise, the directory `NACK`s the `ShReq`.

Question 5 (5 points)

Does Ben's modification remove all Harmful prefetches? Explain briefly.

Yes, it removes all harmful prefetches. For a prefetch to be harmful, it needs to downgrade a remote core's copy of the line. However, the new mechanism directly prevents this by ignoring all prefetch requests when any other core has the line in modified state.

Question 6 (5 points)

Describe a sequence of accesses for Ben's 2-core system where Ben's modification to the coherence protocol prevents a Useful prefetch from happening.

Core 2: write A+4
Core 1: read A <---- tries to prefetch here, but request is NACKed.
Core 1: read A+4

Notice here that the prefetch for line A+4 would've been useful since it is read immediately without any intervening invalidations or evictions.

Part C: Memory Consistency (25 points)

Consider a system with two processors. Assume the following:

- Each processor has a private cache, and *private caches are kept coherent*.
- Memory location `A` initially contains a value of 0.
- The contents of registers `R1` and `R2` for both processors alias to the same memory location `A`, i.e., `Reg[R1] == Reg[R2] == A`. In other words, *all* memory operations are to the same address, `A`, for both processors, but this is not necessarily known at issue time.

Consider the following memory accesses performed by each processor:

P1	P2
P1.1: ST 0(R1) ← 1 P1.2: ST 0(R2) ← 2	P2.1: LD Ra ← 0(R1) P2.2: LD Rb ← 0(R2)

Question 1 (17 points)

For the following questions, answer whether the final set of values `(Ra, Rb) = (2, 1)` are possible under the corresponding memory consistency model. *Note*: Because caches are kept coherent, all memory consistency models must enforce coherence invariants (write propagation and write serialization).
Note that for this to happen, either the stores or the loads need to be reordered w.r.t. program order.

a) **Sequential Consistency**: stores and loads of individual processors happen in the order specified by the program. (4 points)

No, as SC doesn't allow any out-of-program-order loads or stores.

b) **Total Store Order (TSO)**: stores can be reordered after later loads. (4 points)

TSO is irrelevant here as there's no store-to-load reordering that can happen.

c) **Partial Store Order (PSO)**: stores can be reordered after later loads and stores. (4 points)

No. Key point here is that to preserve coherence, stores have to be visible to P2 in the program order of P1, so PSO does not allow store-to-store reordering.

d) **Relaxed Memory Order (RMO)**: loads and stores can be reordered after later loads and stores. (5 points)
Yes. RMO allows load-load reordering.

Question 2 (8 points)

The RMO machine 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.

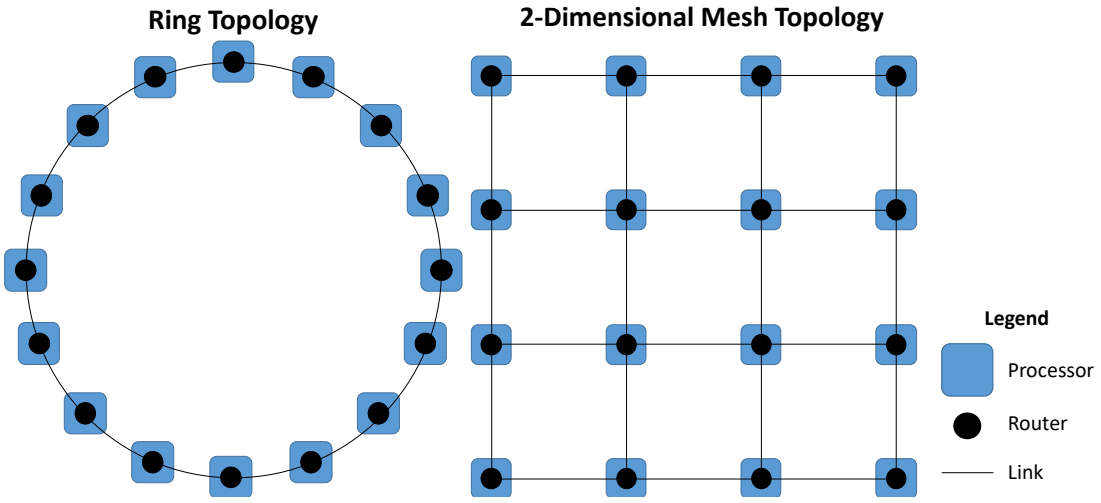
Using the **minimum number of memory barrier instructions**, rewrite **P1** and **P2** from Question 1 such that the **RMO machine produces the same outputs as the SC machine for the given code.**

P1	**P2**
P1.1: ST (A) ← 1	P2.1: LD Ra ← (A)
	MEMBAR$_{RR}$
P1.2: ST (A) ← 2	P2.2: LD Rb ← (A)

Part D: Networks (30 points)

Question 1 (8 points)

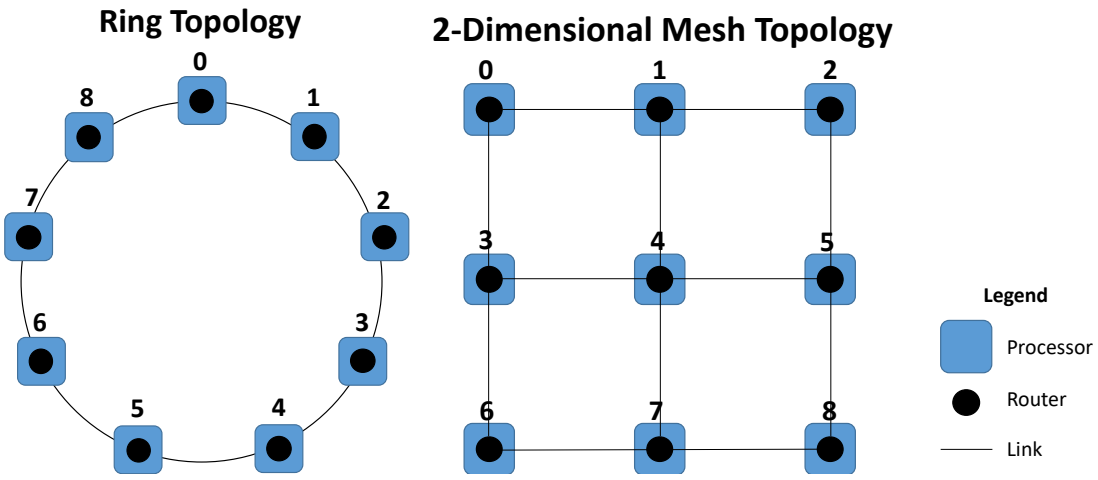
Consider the following two topologies: A ring and a 2-dimensional mesh.



Assuming that there are N nodes, calculate the following for each topology. Assume uniform random traffic for average distance calculation. For the average distance of the mesh, you can answer in terms of its asymptotic growth with respect to N (e.g., $O(N^3)$).

	Ring	2-D Mesh
Number of links	N	$2(N - \sqrt{N})$
Bisection Bandwidth	2	$\sqrt{N}$
Diameter	$N/2$	$2(\sqrt{N} - 1)$
Average Distance	$N/4$	$O(\sqrt{N})$

Now consider a 9-node network for the two topologies. Each node is assigned a unique ID as shown in the figure below:



Question 2 (4 points)

Consider a *neighbor* traffic pattern instead of the usual uniform random traffic. In this traffic pattern, **processor *i* sends all of its traffic to processor (*i* + *1) mod 9***.

What are the average distances of the two topologies under the neighbor traffic pattern?

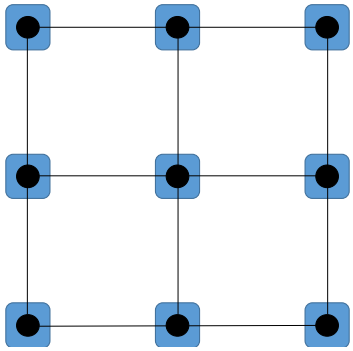
Ring: every neighbor is 1 hop away, so it's 1
Mesh: every neighbor is 1 hop away except: 2->3 and 5->6 which is 3 hops, and 8->0 which is 4 hops. Thus, average diameter = (1*6 + 3*2 + 4)/9 = 16/9

Question 3 (4 points)

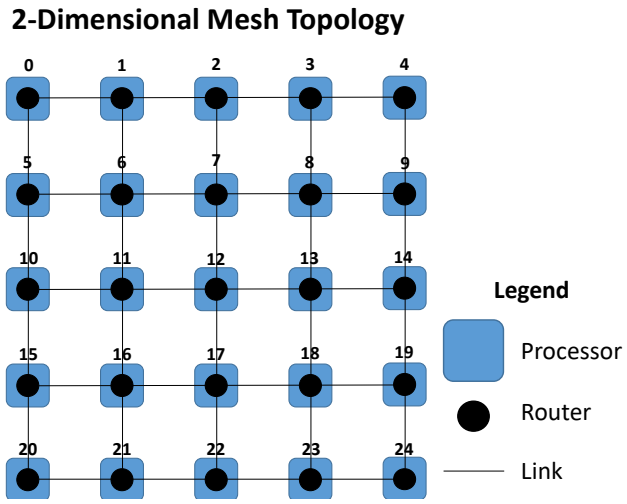
Can a relabeling of nodes lead to a lower average distance for the mesh topology with the neighbor traffic pattern? If possible, show the relabeling in the figure below. If not, briefly explain your reasoning.

Yes, multiple possible relabelings can result in lower average distance. One possible one: swap the labels of nodes 3 and 5.

2-Dimensional Mesh Topology



Now consider a 25-processor mesh network, as shown below:



In this system, each processor produces a single value, and all values must be summed up:

```
sum = v0 + v1+ v2 + ... + v24
```

where `vi` is the value produced by processor `i`.

To accomplish this, each core sends a packet containing its value `vi` to a single processor, called the *reduction node*. The reduction node computes the final `sum` by adding values as they arrive. Ben chooses processor 12 as the reduction node.

Assume the following:

- Each processor `i` sends a packet containing its value `vi` at cycle 0.
- Each packet takes 1 cycle to traverse a router. Assume unlimited buffering at each router.
- Each router-to-router and router-to-processor link represents two channels in opposite directions. Channel traversal is *instantaneous* i.e., no additional cycles are spent in traversing a channel.
- Only one packet can traverse a router-to-router link at a given cycle, whereas an unlimited number of packets can traverse a router-to-processor link at a given cycle. The processor is able to reduce all packets it receives in the same cycle.
- The mesh network routes packets via XY-routing (all horizontal links are traversed before vertical links).

Question 4 (4 points)

We define that a router is *congested* when multiple packets in its input ports are wish to be routed to the same output channel in a given cycle. Which router is congested for the most number of cycles during the reduction?

Routers 7 and 17 are the most congested since they need to route traffic from all nodes in their row and above/below, for a total of 10 packets.

Question 5 (5 points)

At what cycle does the last packet arrive at the reduction node?

The congested router will start sending packets at cycle 0, so the last packet arrives the *router* at the reduction node at cycle 10. We need 1 more cycle to traverse the last router (note that we define the processor as the reduction node), so it arrives at cycle 11.

Ben tries to reduce router congestion by having each router sum up the packets as they arrive. Each router is now augmented with an adder. If multiple packets wish to be routed to the same output channel in the same cycle, the router creates a new packet at the output channel in the next cycle that contains the summed values of those packets.

Question 6 (5 points)

Assume each processor `i` sends a packet containing its value `vi` at cycle 0. At what cycle does the last packet arrive at the reduction node with the modified routers?

Now it's simply the latency from the farthest processor to the reduction node, which is 5 cycles (5 routers to traverse).

Scratch Space

Use these extra pages if you run out of space or for your own personal notes. We will not grade these unless you tell us explicitly in the earlier pages.